Using machine learning for automatic classification of the layout quality of UML class diagrams

UNIVERSITÀ DI PISA

Armillotta Domenico

Supervisor: Prof. Cimino Mario
Alfeo Antonio Luca
Fruzzetti Chiara

Department of Engineering
University of Pisa

This dissertation is submitted for the degree of
MSc Artificial Intelligence and Data Engineering

Abstract

In this thesis project, an automated approach for evaluating UML class diagrams will be presented. Unified Modeling Language (UML) class diagrams are widely employed throughout the software design lifecycle to model Software Requirement Specifications during the development of any software. From an industrial standpoint, this tool could be utilized for automated quality assurance of class diagrams, such as integrating it into a DevOps toolchain as part of a quality monitoring process. For instance, automated feedback can be generated once a UML diagram is checked into the project repository. The primary approach employed involves advanced machine learning techniques, necessitating a training phase, particularly if this tool is to be used in an industrial context. The software will automatically assess the quality of a diagram using an image as input. In the future, the software can be extended to accept the exported XML from the program used to create UML diagrams. This project was carried out under the supervision of the University of Pisa and the company MBDA Italy.

Table of contents

Introduction 1

1 State of Art **5**

1.1 Paper 1 : Evaluating the layout quality of UML class diagrams using machine learning 6

1.1.1 Dataset 6

1.1.2 Metrics 7

1.1.3 Extraction Algorithm 8

1.1.4 Classifier 9

1.2 Paper 2 : Parsing Structural and Textual Information from UML Class . . . 9

1.2.1 Model used for feature extraction 10

1.2.2 Model used for text extraction 11

1.3 Paper 3 : Automatic classification of UML Class diagrams from images . . 11

1.3.1 Feature extraction 11

1.3.2 Feature extracted 12

1.3.3 Classifier 12

1.3.4 Result 13

1.4 Paper 4 : Extracting UML Models from Images 13

1.4.1 Image processing 13

1.4.2 Generate XML file 14

2 First model : segmentation **15**

2.1 Dataset and Labelling 15

2.2 Faster R-CNN architecture 17

2.2.1 How it works 17

2.2.2 Training 18

2.2.3 Model usage 19

2.2.4 Performance 20

2.3 YOLOv8 architecture 27

2.3.1 How it works . . . 27
2.3.2 Training . . . 28
2.3.3 Performance and Usage . . . 29
2.4 Alternative approach to segmentation . . . 30

3 Class Detection **33**
3.1 Class segmentation . . . 33
3.2 Feature extraction . . . 34

4 Arrow detection **37**
4.1 Version 1 : with Faster R-CNN . . . 37
4.1.1 Arrow extraction . . . 37
4.1.2 Arrows identification . . . 38
4.2 Version 2 : without Faster R-CNN . . . 42
4.2.1 Arrow extraction . . . 42
4.2.2 Arrow identification . . . 42
4.3 Feature extracted . . . 46
4.4 Tested versions . . . 47
4.4.1 Solution with OpenCV countours . . . 47
4.4.2 Solution with start/end point . . . 48
4.4.3 Solution with connected point . . . 49
4.4.4 Solution with Canny algorithm . . . 52
4.4.5 Solution with Harris corner + Canny . . . 52
4.4.6 Solution with start/end point + Canny . . . 55
4.4.7 Solution with HoughLinesP algorithm . . . 55

5 Cross line detection **59**
5.1 Cross extraction . . . 59
5.2 Cross identification . . . 62
5.3 Feature extracted . . . 63
5.4 Algorithm at work : special case . . . 63

6 Image info detection **67**
6.1 Feature Extracted . . . 67

7 Text detection **69**
7.1 Text extraction . . . 69
7.2 Pre processing . . . 70

7.3 Semantic processing 71
7.4 Detection of wrong method inside class 73

8 Second model: quality classifier 75
8.1 Dataset 75
8.2 Type of classifier 77
8.2.1 KNN 77
8.2.2 Random Forest 78
8.2.3 Decision Tree 80
8.2.4 Naive Bayes 83
8.2.5 Neural Network classifier 85
8.2.6 SMOTE 87
8.2.7 Ensemble Methods with three classifier and SMOTE 89
8.3 The best classifier 91
8.4 Correlation Analysis 92

9 Limitations of the application 99
9.1 First model limits : segmentation 99
9.2 Feature extraction limits 101
9.3 Classifier limits 102
9.3.1 Amount of data 102
9.3.2 Important feature 103
9.3.3 Miss classified image 104

10 Final Application 113
10.1 Composition of the two model 113
10.2 Overall performance 117
10.3 Flask application 119
10.4 Conclusion and future work 121
10.4.1 Conclusion 121
10.4.2 Future work 122

References 127

Introduction

The pipeline designed (fig . 1) to achieve the final goal consists of three different crucial phases:

1. Model creation for feature extraction.
2. Model creation for classification.
3. Deploying the final application.

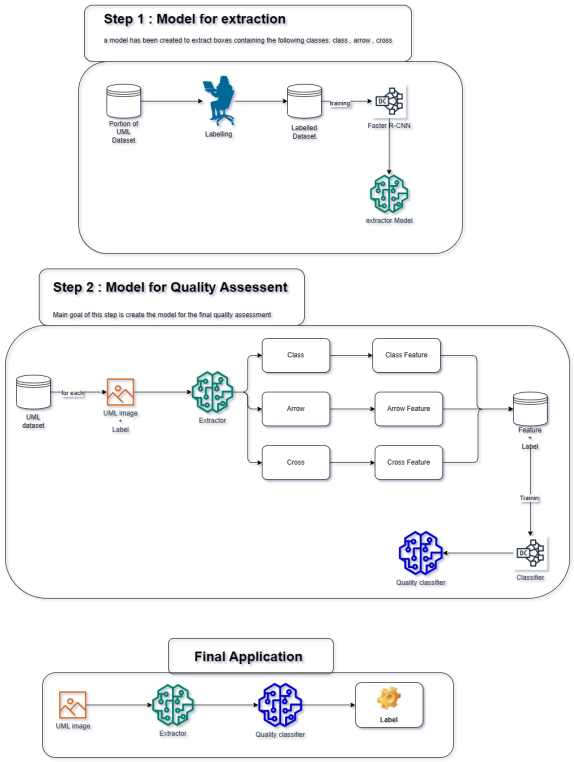

Fig. 1 pipeline

First phase : Model for extraction

In the first phase, a model is created for extracting useful information for the classification from UML diagram images. This phase is critical because, it was necessary to identify the relevant metrics to describe a good UML model.

The extracted metrics closely resemble those in Paper1 [3], to which additional metrics were added after identifying the primary applications of the projects that would be analyzed

in collaboration with the company. All metrics are described in detail in the upcoming section. Indeed, the metrics to be extracted will determine the future utility of the entire software, as incorrect metrics could be unrepresentative during quality assessment.

For the creation of this initial model, a Faster R-CNN architecture was used, one of the neural network architectures employed for object detection in images. Training was conducted on a dataset consisting of several thousand UML diagrams, which were appropriately labeled at every element and then globally rated.

Therefore, by the end of this phase, a model capable of extracting all the elements (arrow, class, cross) within a UML diagram image was obtained.

Second phase : Model for quality assessment

In the project's second phase, following the creation of the feature extractor model and the computation of metrics (25 metrics in total for each image), these metrics are utilized by the classifier responsible for assigning labels to the images.

To identify the classifier with the best performance, several tests were conducted. The considered classifiers included KNN, Naive Bayes, DNN (Deep Neural Network), Random Forest, and Decision Tree. Due to the limited quantity of training data, the classifiers had their hyperparameters suitably tuned and optimized using various techniques including data augmentation.

Third phase : Final application

The final application is tasked with integrating the two models into a comprehensive pipeline that takes an image as input and, after processing, generates a quality score for the UML diagram ranging from 1 to 5, based on its quality and a feedback to improve quality useful at the designer.

The final pipeline is designed to integrate both parts, namely the Faster R-CNN and the classification aspect. For the thesis project, two pipelines were created that interweave the various aspects encountered during development in different ways, highlighting different implementations with varying performance levels. The two pipelines are very similar in terms of technologies and solutions, but they differ in their implementation approach. Both leverage the power of the Faster R-CNN, algorithms like Canny edge detection, Hough Transformation, and Ramer–Douglas–Peucker for metric extraction, as well as various classifiers such as KNN, Random Forest, and DNN for the final classification.

Finally, the end application has been deployed using the Flask micro service to provide a better interface. The shown pipeline exhibits short execution times, which are influenced by the hardware on which the server runs.

Chapter 1

State of Art

Intro

In the field of software engineering, assessing the quality of Unified Modeling Language (UML) diagrams is a highly relevant problem. UML diagrams are widely used to visually represent the structure and behavior of software systems, making their correct interpretation essential. However, manually evaluating the quality of such diagrams can be a challenging and error-prone task.

Recently, artificial intelligence (AI) and computer vision have been applied to address this challenge. Several studies, such as "Evaluating the layout quality of UML class diagrams using machine learning" and "Parsing Structural and Textual Information from UML Class diagrams to assist in verification of requirement specifications," have explored the use of machine learning techniques to assess the layout quality of UML diagrams and to extract structural and textual information from UML class diagrams.

Furthermore, research like "Automatic classification of UML Class diagrams from images" and "Extracting UML Models from Images" has demonstrated the feasibility of extracting UML models from images using computer vision techniques. These approaches promise to automate the quality assessment process, increasing efficiency and reducing the likelihood of errors.

However, despite significant progress, many challenges remain. Research in this field is active and continuously evolving, with the goal of further improving the accuracy and efficiency of existing methods and developing new techniques to address unresolved issues.

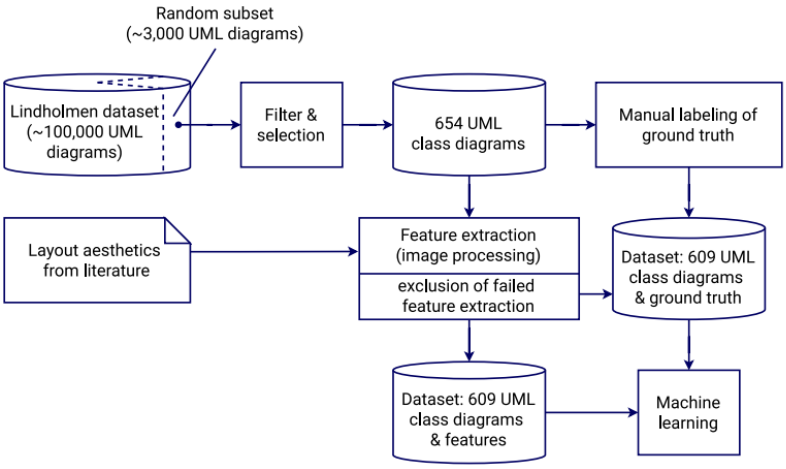

Fig. 1.1 Pipeline of paper

1.1 Paper 1 : Evaluating the layout quality of UML class diagrams using machine learning

In this paper [3], is presented an automated method for evaluating the layout quality of UML class diagrams. The approach used is machine learning techniques to build (linear) regression models based on features extracted from the class diagram images using image processing. As ground truth, is used a dataset of 600+ UML Class Diagrams for which experts manually label the quality of the layout.

The focal point of this paper is the metrics extracted and computed from various parts of the paper, including features derived from classes, arrows, and cross lines. These metrics are important because they manage to bring out the characteristics that distinguish a readable UML design from a non-readable one. This paper has been particularly important, especially for the metrics that served as inspiration for the work conducted in this thesis, and for the dataset, which has been elaborated upon extensively and made available by the paper's authors. In fig. 1.1 is it possible see the entire pipeline followed by the paper.

1.1.1 Dataset

The starting point for the research is the Lindholmen dataset of images created by Hebig et al. (2016). This dataset consists of almost 100,000 UML diagrams. This dataset was assembled through mining open source repositories on GitHub for images. From this dataset, all images that were not class diagrams, hand-drawn, not screenshots, duplicated diagrams, or too simple were discarded. After applying the above criteria as filters, a total of 654 diagrams (out of approximately 3,000 in the initial set) remained useful. The labeling was performed

manually by six experts in the areas of software modeling and HCI, and the label is a 5-point Likert scale: 1: Very Bad, 2: Bad, 3: OK, 4: Good, and 5: Very Good. A Likert scale is an ordinal scale.

1.1.2 Metrics

Class metrics

- Rectangle orthogonality. The orthogonality of rectangles is calculated by counting the number of distinct rectangle positions on the x and y axis, respectively
- Rectangle coverage
- Aspect ratio.
- Rectangle distribution. For this feature, the image is divided into four equal quadrants (obtained by drawing horizontal and vertical lines through the image's center point)
- Rectangle proximity. The distance from its center to the center of the other rectangles is calculated for each rectangle
- Rectangle size
- Rectangle size variation
- Number of rectangles

Arrow metrics

- Line bends. Lines are represented as straight segments, and the number of bends on a line is one less than its number of segments
- Line angles. The deviation angle from orthogonality
- Average line length
- Line length variation
- Longest line
- Shortest line
- Number of lines.

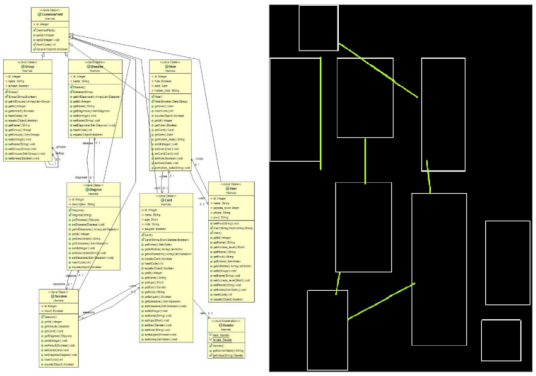

Fig. 1.2 Paper feature extraction

Cross metrics

- Line crossings. The number of line crossings in a diagram can be found by counting the intersections between the found lines.
- Crossing angles. For each line crossing, the crossing angle is calculated

1.1.3 Extraction Algorithm

Four common image processing algorithms are used for finding elements in the diagram images. Those are Canny edge detection Hough Transformation, Suzuki 85 and Ramer–Douglas–Peucker. Is it possible see the result in fig.1.2.

Canny edge detection is an image processing algorithm that identifies edges by blurring the image with a Gaussian filter, computing edge gradients and directions, and then thresholding to remove noise and retain high-intensity gradient edges.

Hough Transformation (HT) detects straight lines in images by representing edges with direction and distance from the image center, grouping edges with the same direction and distance into lines.

Suzuki 85 (S85) is a border-following algorithm used to identify contours in images by scanning pixels and marking all pixels on a border that satisfies specific conditions.

Ramer–Douglas–Peucker (RDP) approximates a polygon with a small number of points from a curve with many points by recursively removing less significant points. It is suitable for identifying shapes, like approximating a curve with four points to potentially represent a rectangle.

1.1.4 Classifier

The software tool Weka5 was used for the machine learning parts of this research. It is not an excellent tool compared to Python, which offers better and more customizable tools for the specific use case. The paper's results are excellent; in fact, the evaluation yielded a number on this scale: in 75.4% of the cases, the produced value was not more than 0.5 away from the ground truth. The paper, using Weka, does not provide the implementation of the classifier with the fine-tuned parameters of the model.

1.2 Paper 2 : Parsing Structural and Textual Information from UML Class

This paper [2] proposes an extension of Computer Vision and Neural Networks applications to parse hand-drawn UML class diagrams and extract requirement specifications for validating software requirement completeness. The objective is to efficiently convert hand-drawn class diagrams into their digital counterparts and analyze the contained requirements. Hand-drawn diagrams may contain errors and discrepancies compared to the original specifications due to human errors. This paper has been particularly important for the segmentation part. In fact, the Python library "detecto" was used to build a Faster R-CNN, from which inspiration was drawn after evaluating its performance. Even though the analyzed problem is different, reading and analyzing this paper proved to be useful. The part concerning the digitization of the detected graphs is not important for the thesis's objective, so it will not be explored. Only the extraction of text and classes will be analyzed. In fig. 1.3 is possible see the pipeline followed by the paper.

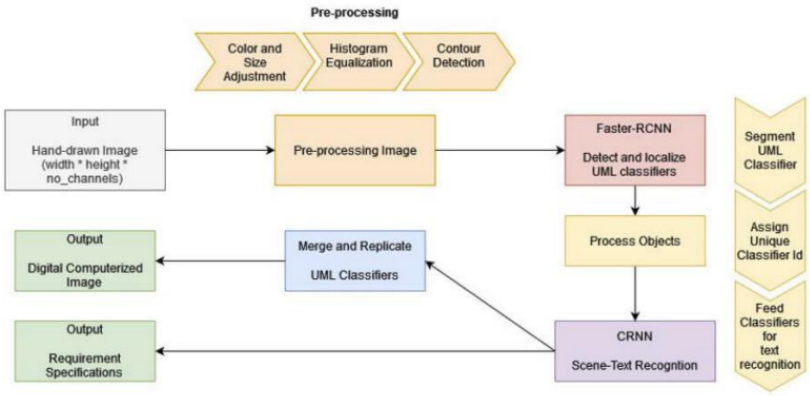

Fig. 1.3 Pipeline of paper

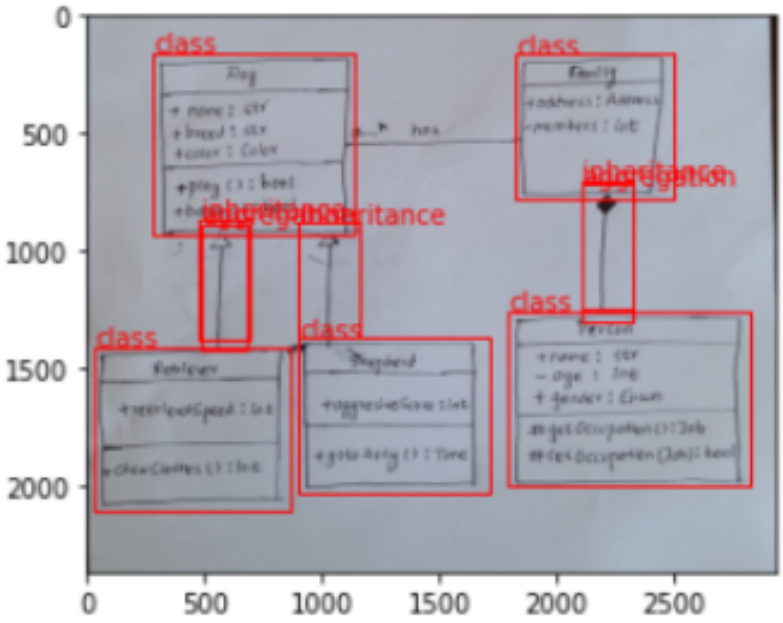

Fig. 1.4 Feature extraction with Faster R-CNN

1.2.1 Model used for feature extraction

This paper discusses the evolution of object localization techniques in Computer Vision. It begins with an introduction to CNNs combined with sliding window detectors for object detection but highlights the computational cost of exhaustive search.

The paper introduces the RCNN (Region-based Convolutional Neural Network) as a solution to this problem. RCNN consists of three modules: region proposal generation, feature extraction, and category-specific classification. It uses selective search to generate region proposals and then processes them using a CNN.

Fast-RCNN improves upon RCNN by feeding the entire image to the CNN to generate a feature map, allowing for more efficient processing of region proposals. ROI pooling is used to reshape the feature map.

Faster-RCNN further enhances speed and accuracy by introducing a Region Proposal Network (RPN) to predict region proposals, eliminating the need for selective search. The RPN module guides the Fast-RCNN module in identifying regions of interest. In fig. 1.4 is showen an example of feature extration from UML class diagram.

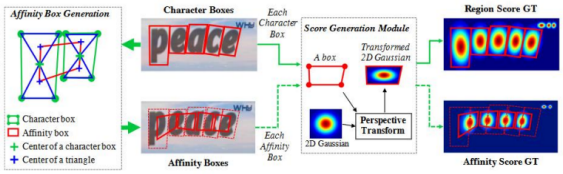

Fig. 1.5 CRAFT architecture

1.2.2 Model used for text extraction

The paper also discusses CRAFT (Character Region Awareness for Text Detection), a framework for text detection and in fig. 1.5 is represented the architecture. CRAFT detects individual character regions and links them to form text instances using weakly supervised training.

CRNN (Convolutional Recurrent Neural Network) is presented as an efficient method for scene text recognition, combining hierarchical feature maps with bi-directional Long Short-Term Memory and dense neural networks.

1.3 Paper 3 : Automatic classification of UML Class diagrams from images

The paper [4] introduces an automated system designed to classify UML class diagram images in two classes : no-uml and uml. The study proposes 23 image features and explores their effectiveness in classifying UML class diagram images. The performance of these features is assessed based on their Information Gain Attribute Evaluation scores. The study further evaluates the specificity and sensitivity of six classification algorithms using a dataset containing 1300 images. The results indicate that 19 out of the 23 introduced features play a significant role in predicting the classification of UML class diagram images. The research problem specifically focuses on identifying suitable features and classification algorithms to distinguish UML class diagram images from non-UML class diagram images. Additionally, a comparison is made between various classification algorithms to determine the most suitable one for the task.

1.3.1 Feature extraction

Shape and line extraction is carried out using three external algorithms: Hough transform (HT); Suzuki85 (S85); and Ramer–Douglas–Peucker (RDP) . The contours that S85 finds are used to find various shapes and are subsequently broken down into straight lines. Using

the algorithm in conjunction with HT leads to better detection of lines. The lines are then processed, so that horizontal and vertical lines, that are on the same axes and represent the same line, are joined together into a single line. Rectangles that are not caught by using S85 are then extracted by finding horizontal lines that are parallel and in the same position on the x-axis, and have the same two vertical lines intersecting them on each end. RDP is used to find different types of polygon: rectangles, rhombuses, triangles and ellipses. So the algorithms used for feature extraction are very similar between this and the first paper analyzed.

1.3.2 Feature extracted

In this case, the extracted metrics are identical to those in the first analyzed paper, except for the following additional metrics:

- Rectangles horizontally/vertically aligned
- Average horizontal/vertical line size.
- Parent rectangles in parent rectangles.
- Noise : Detected lines that are outside of rectangles, divided by the number of all detected lines.
- Colour frequency : Three most frequent colours in the image are found. percentage

1.3.3 Classifier

The paper tried to select the most appropriate classification algorithm involved choosing algorithms that encompass diverse approaches to classification. The six selected classification algorithms are as follows:

- Decision Table (DT)
- Random Forest (RF)
- Support Vector Machine (SVM)
- Logistic Regression (LR)
- REP-Tree (RT)
- J48 Decision Tree (J48)

1.3.4 Result

In summary, LR excels in the task of eliminating non-UML CD images and is therefore considered the most suitable classification algorithm for our classifier, utilizing the mentioned extraction features , achieving a 91.4% correct classification rate for non-UML CD images.

1.4 Paper 4 : Extracting UML Models from Images

The paper [1] focuses on the phase of extracting UML diagrams from images, without evaluating the characteristics or quality of the extracted diagrams. The paper introduces the Img2UML tool, designed to extract UML Class models from images, making them compatible with CASE tools for further analysis. The tool exports these UML Class models into XMI files, which can be conveniently accessed using CASE tools like StarUML. The Img2UML tool, which automates the extraction of UML Class models from images without requiring manual reconstruction. The analysis of this paper has focused on the feature extraction part, even though it was not useful for implementation since the algorithmic implementation is not specified in the paper.

1.4.1 Image processing

The paper outlines a four-step process for image processing:

1. Detecting Classes in the image: The initial step involves identifying classes within the image. This is achieved by detecting rectangles in the image using the Aforge.NET Framework image processing library. Image quality is enhanced through various filters, such as Gaussian Sharpen, Grayscale, and Threshold.

2. Recognizing text in the classes: Rectangles representing classes can have three parts containing text: class name, attribute names, and function names. The system identifies these parts by detecting two or three rectangles within larger rectangles. Optical character recognition (OCR) using Microsoft Office Document Imaging (MODI) is employed for automatic text recognition.

3. Detecting Relationships: Dependencies between classes are depicted through connecting lines in class diagrams. Various styles of connecting lines exist, such as straight, hooked, diagonal, curved, solid, and dotted. The tool can detect straight connections but not curved ones. The paper assumes that each line connects two classes, without specifying the type of relationship.

4. Detecting UML Class model symbols: This stage focuses on detecting the types of relationships between previously identified classes. UML Class models involve four relationship types: associations, generalization, dependency, and realization. Additionally, association relations include four subtypes: association, direct association, aggregation, and composition. Recognition of six UML class model symbols is necessary to determine seven relationship types. Shape recognition algorithms from the Aforge.NET Framework image processing library are employed for symbol recognition, along with geometric-based operations.

1.4.2 Generate XML file

Once classes and relationships have been identified and recognized in the image, the next step involves extracting and organizing all model information. The XMI (XML Metadata Interchange) format is the preferred choice for this purpose. The paper does not reference the type of code utilized for the transformation into XML.

Chapter 2

First model : segmentation

Intro

The goal of segmentation is to partition the various classes of objects present within UML diagrams as accurately as possible. In our specific case, the different classes are "classes," "arrows," and "intersections." In the literature there are a lot of technique for segmentation , are been tested the main approach : YOLO and Faster R-CNN , to choose the best one. For the Faster R-CNN, the Python library "Detecto" was used. It is a Python library build on top of PyTorch ,that allows the construction of computer vision and object detection models. Inferences can be performed on images and static videos, transfer learning on custom datasets, and model serialization to files. Detecto is based on PyTorch, enabling easy model transfer between the two libraries. Detecto simplifies the process of initializing models, applying transfer learning on custom datasets as in our case with a UML dataset.

2.1 Dataset and Labelling

The dataset used for training the models, which was later labeled, is the same as the one used in Paper 1 analyzed in the state of the art. It is derived from scraping UML diagrams extracted from open GitHub repositories and then filtered by experts who removed all diagrams with low quality, originating from screenshots or hand-drawn ones. So, as a result, there is a dataset for this training of approximately 650 diagrams, which may seem few, but since these diagrams are filled with classes, arrows, and intersecting lines, they contain thousands of objects from which the model can learn during training. The dataset being not huge is divided in 80% train , 10% validation set and 10% test set. Even though MBDA made efforts to retrieve UML diagrams from some older projects, they were unable to provide a dataset for

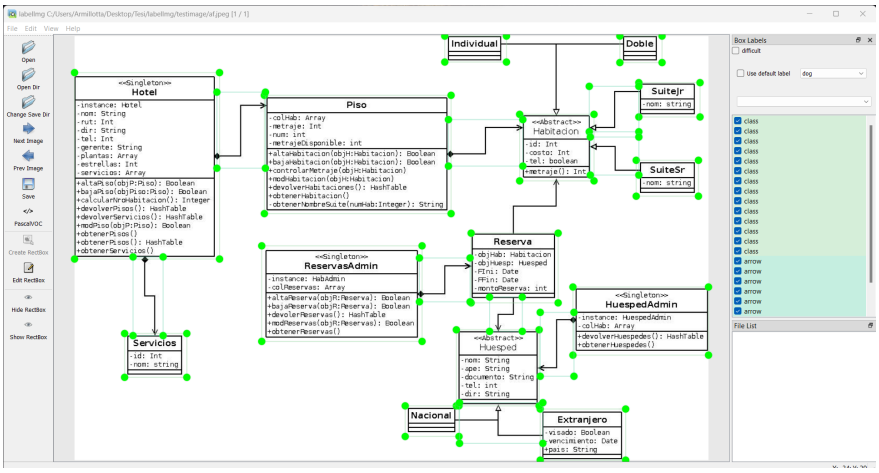

Fig. 2.1 ImageLabl software

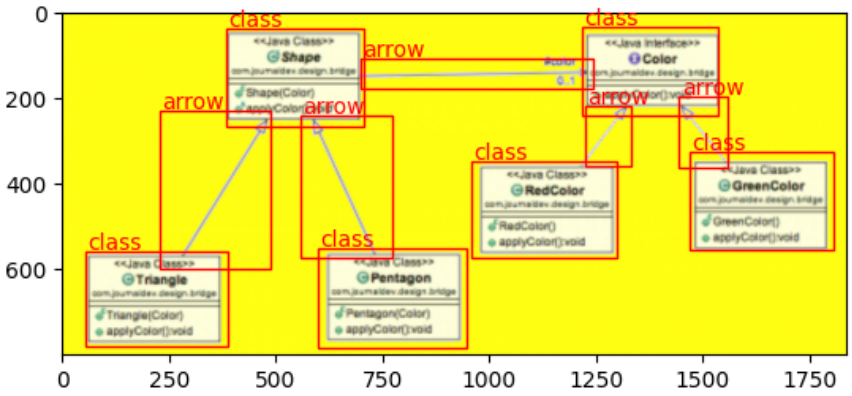

Fig. 2.2 Uml labelled

training or testing. This was due to the dataset containing highly sensitive information that could only be accessed by employees. As labeling is crucial for model training, it was carried out meticulously manually using the open-source program LabelImg . Boxes for each class were manually drawn on each UML diagram image so that the program could generate an XML file for each image containing all the necessary information, which will subsequently be used for model training. The process is shown in fig. 2.1, with the classes listed on the left, can be observed in the image. In the labeling process, all the images comprising the dataset were analyzed, which amounted to approximately 650 diagrams, it took about 25 hours to create the label for all the images. Each image contained dozens of objects, allowing the model to learn to recognize patterns as we can see in fig. 2.2.

2.2 Faster R-CNN architecture

2.2.1 How it works

The architecture chosen for creating the model responsible for partitioning the image is Faster R-CNN. Faster R-CNN, an acronym for "Faster Region-based Convolutional Neural Network," is an advanced architecture for object detection in images. It is one of the most powerful and accurate architectures for object detection tasks and was introduced by Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun in 2015. The primary goal of Faster R-CNN is to combine region-based object detection with the ability to automatically generate proposals for regions of interest (RoI) within the image. In fig. 2.3 it is possible to see the architecture. More in details the main concept behind every Faster R-CNN:

1. Backbone CNN: The basic architecture of Faster R-CNN starts with a pretrained Convolutional Neural Network (CNN), such as ResNet or VGG, serving as a feature extractor. This CNN is responsible for extracting both low and high-level features from the input image, which are utilized to understand image characteristics like edges, textures, and structures.

2. Region Proposal Network (RPN): The central part of Faster R-CNN is the Region Proposal Network (RPN). The RPN is a convolutional neural network that receives the features extracted from the input image through the CNN. Its primary function is to generate a set of region proposals that may contain objects. These proposals are generated by considering a grid of positions and sizes in the output of the CNN and evaluating the likelihood of each region containing an object. The candidate RoIs are then classified as either "object-containing" or "non-object-containing" and regularized to refine their positions.

3. RoI Pooling: After the RoIs have been generated by the RPN, these regions are extracted from the original CNN's feature maps through a process known as RoI pooling. This process ensures that all RoIs have the same size so that they can be fed into a fully connected network for classification and regression.

4. Classification and Regression: The extracted RoIs are passed through two separate branches of the network: one for classification and one for regression. The classification branch is responsible for assigning a class label to the candidate RoIs (i.e., determining what type of object is present in the RoI), while the regression branch is used to refine the positions of the RoIs relative to the actual object boundaries.

5. Non-maximum Suppression (NMS): After classification and regression, the non-maximum suppression phase is performed to remove overlaps among the candidate RoIs and retain only the best proposals. This step ensures that each object in the image is represented by a single RoI.

YOLO in the literature is the best alternative to Faster R-CNN , have the same purpose but different architecture. The Faster R-CNN was chosen as the final object detector over YOLO. Its performance was slightly better, but it offered greater customizability in terms of implementation. YOLO tests will be discussed in the following chapter.

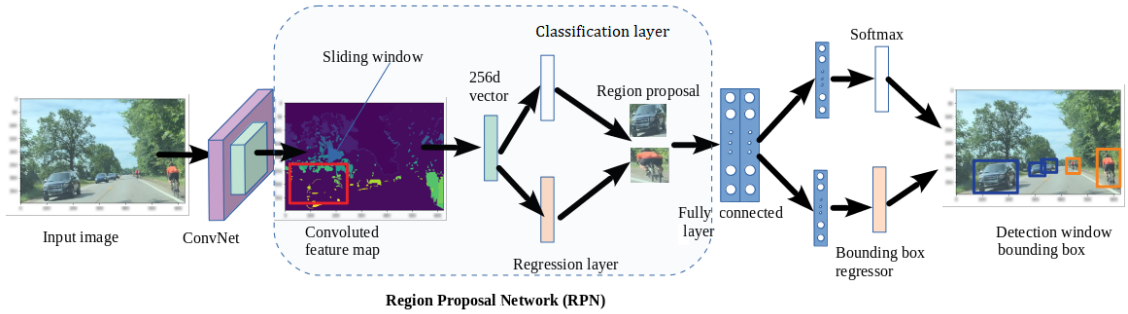

Fig. 2.3 Faster R-CNN architecture

2.2.2 Training

During the training phase, the Faster R-CNN model is trained with all the manually labeled data. Naturally, not the entire data-set is used; it was split to include a Test Set and a Validation Set. The labels and the corresponding bounding box information were provided in the VOC (Visual Object Classes) format, which was then transformed into CSV format for training. During this phase, a compatibility issue between the model and the XML of the images was identified. Indeed, the labeling software assigned an XML with a local path and a name that could disrupt the training conducted with PyTorch. Therefore, it was necessary to create a script to address the issues related to the path and verify that the names were correct to be able to run the training also in cloud hardware. After running several epochs, the trained model is obtained as the output. To try to reduce over-fitting, the training set was also augmented using the following code:

```
custom_transforms = transforms.Compose([
# Convert the input image data to a PIL (Python Imaging Library)
transforms.ToPILImage(),
# Resize the image
```

```
    transforms.Resize(800),
    # Perform a random horizontal flip with a 10% probability
    transforms.RandomHorizontalFlip(p=0.1),
    # Perform a random vertical flip with a 10% probability
    transforms.RandomVerticalFlip(p=0.1),
    # Rotate the image by a random angle between -10 and 10 degrees
    transforms.RandomRotation(10),
    # Adjust the color saturation of the image by adding a random value between -0.
    # Adjust the color saturation of the image
    transforms.ColorJitter(saturation=0.3),
    # Randomly convert the image to grayscale
    transforms.RandomGrayscale(p=0.1),
    # Convert the PIL image to a PyTorch tensor
    transforms.ToTensor(),
    utils.normalize_transform(),
])
```

Augmentation is typically applied when training lacks a sufficient number of instances and helps reduce overfitting in deep learning models by generating new data through spatial transformations. In this case, numerous experiments were conducted with various hyperparameters to find the best-trained model, evaluated based on performance and loss. Another problem that was encountered during training was the intermittent use of Google Colab's GPUs. Due to restrictions on RAM and usage time, there were also limitations on the experiments that could be conducted. In fact, the Detecto library model appears to load all the images needed for the step into RAM, which is not optimal for memory management. It would have been much better if tensor representations of the image had been used to avoid the risk of out-of-memory issues during execution. To try to reduce the memory usage , the batch size and the image size was decreased.

The loss curve shown in the fig. 2.4 indicates a stop in the optimal condition of the best model trained around the 6 epoch. In fact, the loss then continues to rise, which is a classic sign of model over-fitting.

2.2.3 Model usage

The trained model is now usable and can identify the following labels: "arrow," "class," and "cross," which are the main components used to calculate metrics. When the model is used without methods for filtering the detected bounding boxes, the result is an image with many

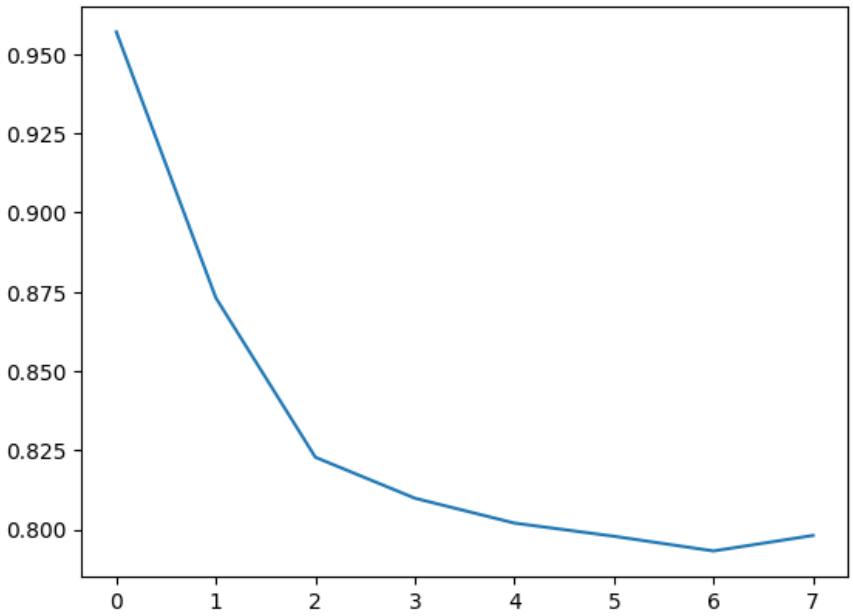

Fig. 2.4 Loss of the model

overlapping and/or low-accuracy bounding boxes, as can be observed in the fig. 2.5. The filtering phase is performed by the individual detection stages because each class of detected object may have different detection thresholds. The thresholds that can be fine-tuned are the Accuracy Threshold and the IoU (Intersection over Union) Threshold.

- Accuracy Threshold: This threshold refers to the minimum confidence level that the model must reach to consider a prediction as accurate.
- IoU Threshold: The IoU (Intersection over Union) is a measure of overlap between two bounding boxes. It is a value that determines how much overlap is required for one bounding box to be considered a duplicate, resulting in the elimination of one of them.

Here is an example of filtering with fine-tuned threshold values, showing a reduction in the number of classes and overlaps as in fig. 2.6. In the final model, hyper parameter tuning was performed for each type of identified object, resulting in different values to achieve optimal recognition of the various objects.

2.2.4 Performance

During the training phase, we trained 5 models responsible for segmenting UML image data. We fine-tuned these models by adjusting their training parameters and then tested

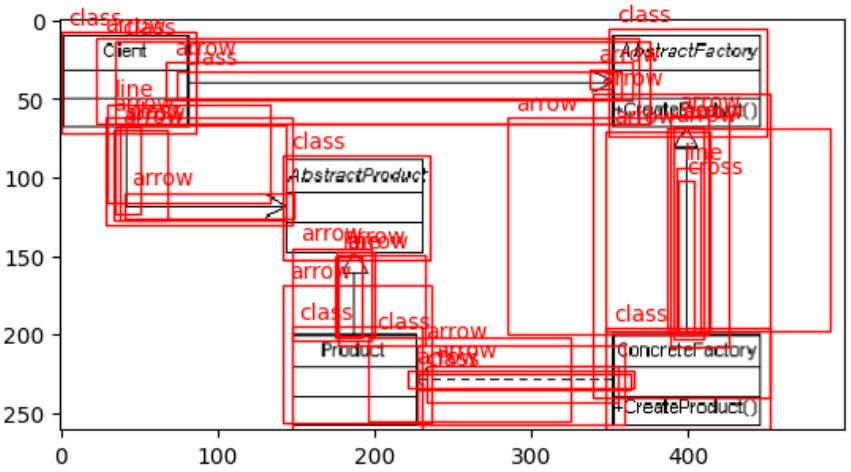

Fig. 2.5 Unfiltered UML scheme

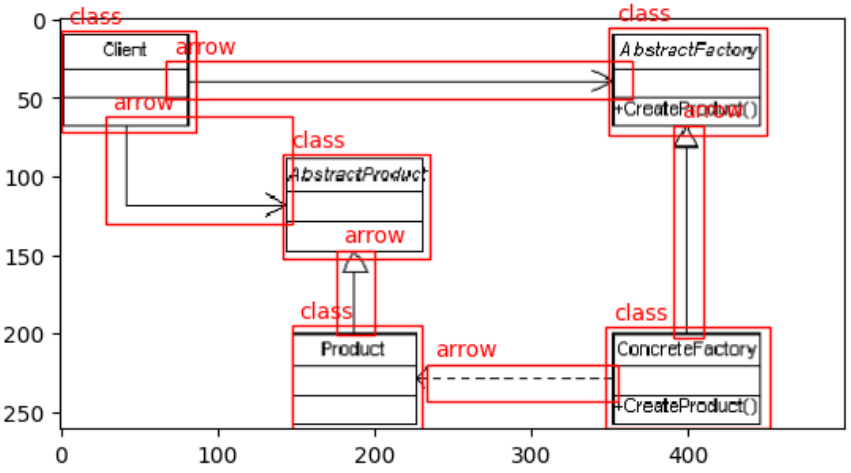

Fig. 2.6 Filtered UML scheme

their performance. All models were evaluated using the same test dataset to enable a more accurate comparison.

Here are the details of the models:

1. **Limited Train**: This model was trained on a smaller dataset compared to the others, roughly half of the entire training dataset, with minimal image variations during augmentation. Training images were resized to a standard size of 800x800.

2. **Full Limited Augmented**: This model was trained on the entire training dataset, but the images were resized to a standard size of 600x600. Augmentation was performed as described in the previous paragraph, including random flip, random grayscale, and random rotation.

3. **Full FullSize Augmented**: Similar to training 2, but the only difference is that the images were not resized and were kept in their native dimensions.

4. **Full FullSize Augmented** (Reduced Randomization): This model was trained similarly to the second model, but this time, randomization within the augmentation process was reduced.

5. **Full Limited Augmentation Crop**: Training was conducted on the entire dataset with images resized to 800x800 pixels. In addition to the previous augmentation techniques, this model also included random cropping. However, it caused issues as it led to training on incorrect bounding boxes.

6. **Full Limited without Augmentation**: Training was conducted on the entire dataset with images resized to 800x800 pixels , but without augmentation.

Here is an overview of the performance of the various models tested:

Model	detected_class	detected_arrow	detected_cross
Limited	770/847	565/844	32/105
Full Limited Augmented	732/847	321/844	20/105
Full size Augmented 1	767/847	466/844	21/105
Full size Augmented 2	759/847	449/844	29/105
Full Limited with Crop	0	0	0
Full no Augmented	789/847	720/844	97/105

This table summarizes the performance on the test set of various segmenters trained as previously described. The test set comprises approximately 66 images, each containing multiple classes—arrows and crosses, totaling 847 classes, including 844 arrows and 105

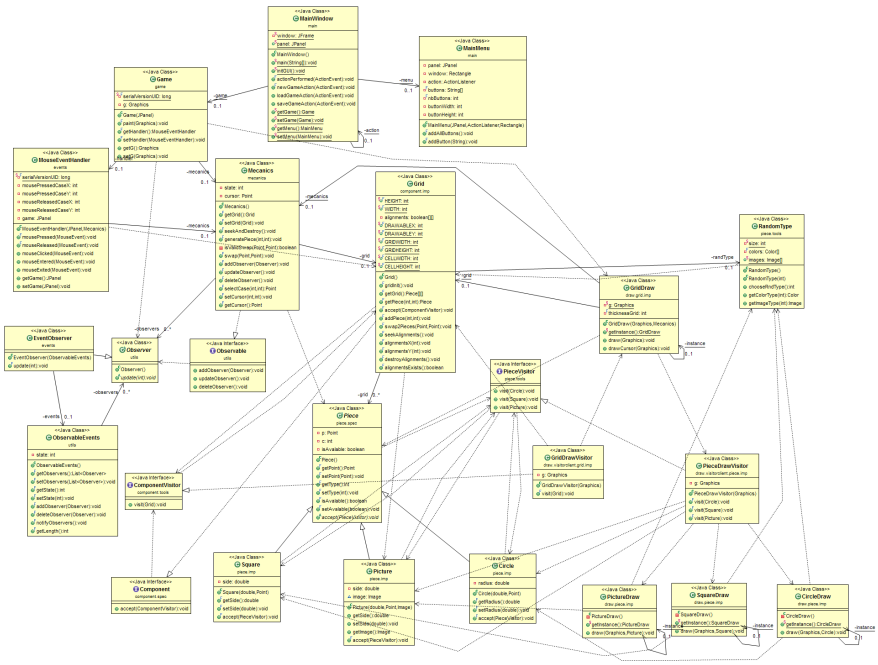

Fig. 2.7 Example of UML with confused arrows

crosses. The segmenters were evaluated based on their ability to correctly identify the number of objects belonging to different classes in each image. Notably, the segmenters trained with the entire training set excelled in performance.

These quality metrics are summary measures. Indeed, the final model's accuracy will be determined by the combined use of the classifier.

To calculate precision and recall, the values of True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN) would have been needed. Therefore, evaluating the percentage overlap for each identified object in every image would have been required. However, this was not conducted because the models' performance varied significantly, and accuracy at this stage does not represent the final model's precision, as this is merely an intermediate point.

All models performed well in class detection, as it's a relatively straightforward task, and there weren't significant issues. The first problems arose in arrow detection, which challenged the models. This is because in simpler and more straightforward examples, all models perform well, but in more complex cases where lines intersect and overlap confusingly, identification becomes almost impossible. We have an example in the fig. 2.7 where even the human eye would struggle to distinguish the lines, especially when the image is resized. The same applies to identifying crossing lines labeled as the "cross" class, which tend to be more difficult, especially when dashed lines often intersect. Performance are not so good like in the class example but the results were affected by the limited training data

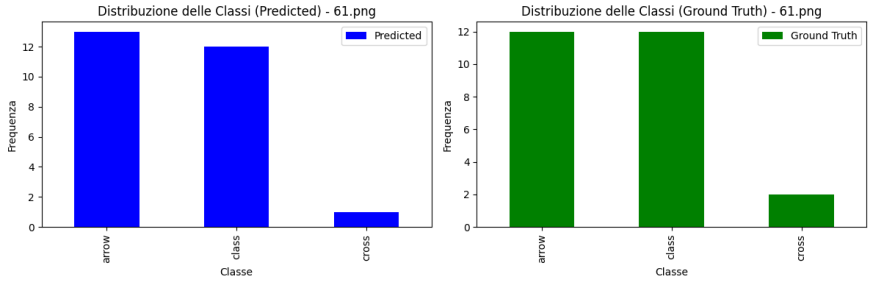

Fig. 2.8 Example of model segmentation

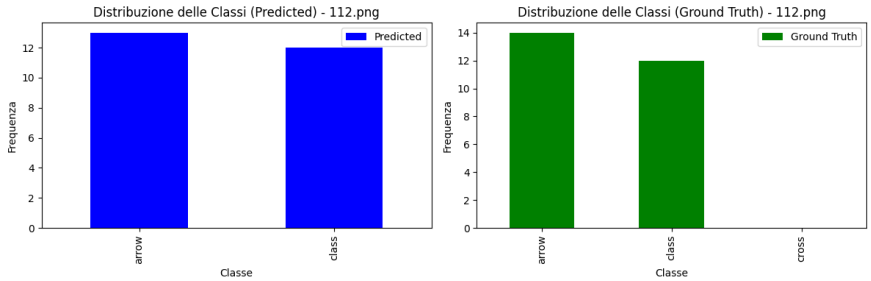

Fig. 2.9 Example simple image

available, which requires a larger data-set for more complex shapes. In fig. 2.8 is possible to see that the model in that image was able to identify also the "cross".

The choice of a Faster R-CNN architecture was made because it is more precise than a YOLO-type architecture. However, further investigations will be conducted to compare the two architectures in terms of accuracy.

The performance image highlights that the model's performance with the "crop" augmentation parameter is zero. This is because, due to cropping, the bounding boxes were incorrect, causing the model to learn incorrect information.

From the selected classifier for the final model, more precise metrics regarding errors were extracted. In the figure, you can see the precision of each class across the entire dataset. More specifically, in figures 2.9 , 2.10 and fig. 2.11,2.12 it's possible to observe that some images have very high precision while others have low precision, which can be attributed to the complexity of the UML diagrams being classified.

From the examples, it appears that in all cases, the classes are detected, but when it comes to complex images, only the clearer arrows are detected.

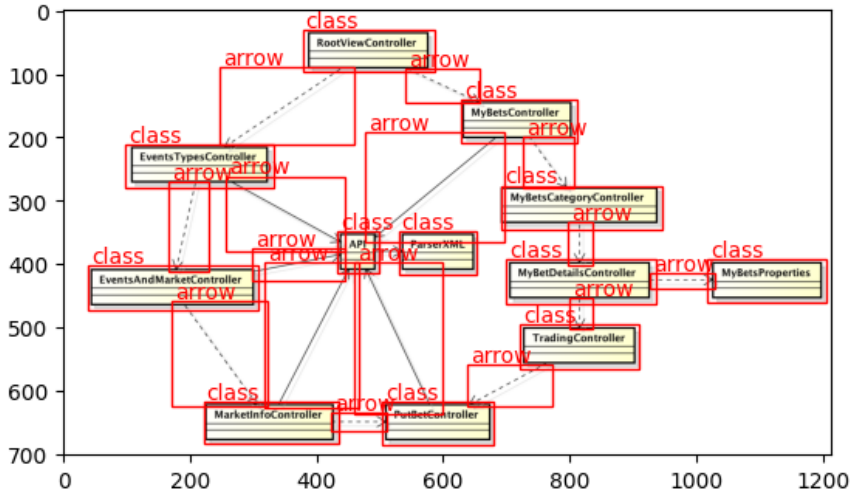

Fig. 2.10 Example simple image

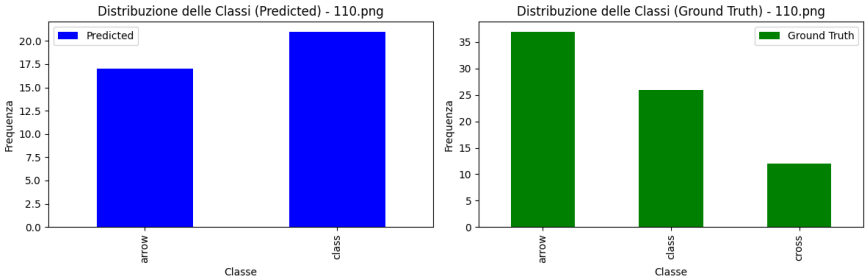

Fig. 2.11 Example complex image

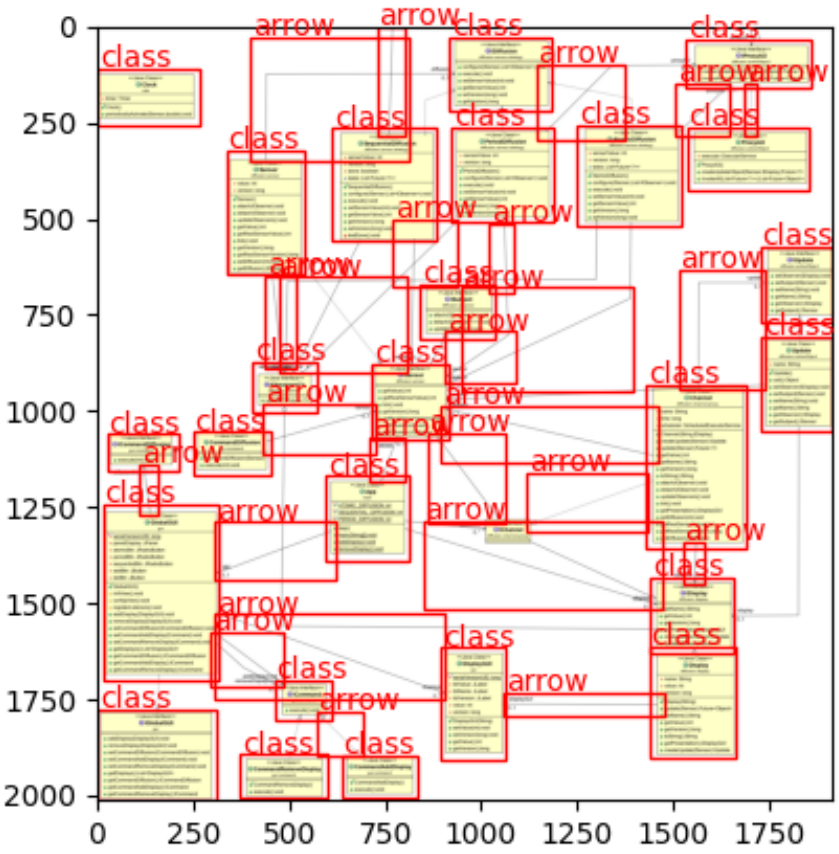

Fig. 2.12 Example complex image

2.3 YOLOv8 architecture

2.3.1 How it works

YOLO, or "You Only Look Once," is a deep learning-based object detection framework that allows for real-time object detection in images or videos. It was developed by Joseph Redmon in 2016. It is an evolved version of YOLO (You Only Look Once) designed to enhance detection performance and execution speed.

YOLOv8 is the latest model in the YOLO family, introduced in 2022 by Ultralytics. YOLOv8 is built upon the YOLOv5 framework and incorporates several architectural and developer experience improvements.

The main components of YOLOv8 include:

1. Backbones: YOLOv8 utilizes various deep neural network architectures such as Darknet, CSPDarknet, YOLOv4, YOLOv4-tiny, and others as backbones for feature extraction from images. These backbones are responsible for representing images in terms of features relevant to object detection.

2. Neck: The "neck" of YOLOv8 is a component that connects the backbone to the final detection module. It may include layers such as PANet (Path Aggregation Network) or SAM (Spatial Attention Module) to enhance detection accuracy.

3. Detection Head: This is the core of the detection model. The detection module consists of specialized neural networks that process the features extracted from the "neck" and generate predictions for objects in the image. It also produces bounding box coordinates, object classifications, and associated confidences.

4. Anchor Boxes: Anchor boxes are used to predict the sizes and positions of bounding boxes around objects in the image. They are used in the detection module to establish the appropriate bounding box dimensions.

5. Non-Maximum Suppression (NMS): After the model generates multiple object predictions, NMS is a post-processing step that eliminates duplicates and retains only the bounding box with the highest confidence when there is significant overlap between them.

6. Post-Processing: YOLOv8 may also include additional post-processing steps such as bounding box refinement or handling objects that span multiple bounding boxes.

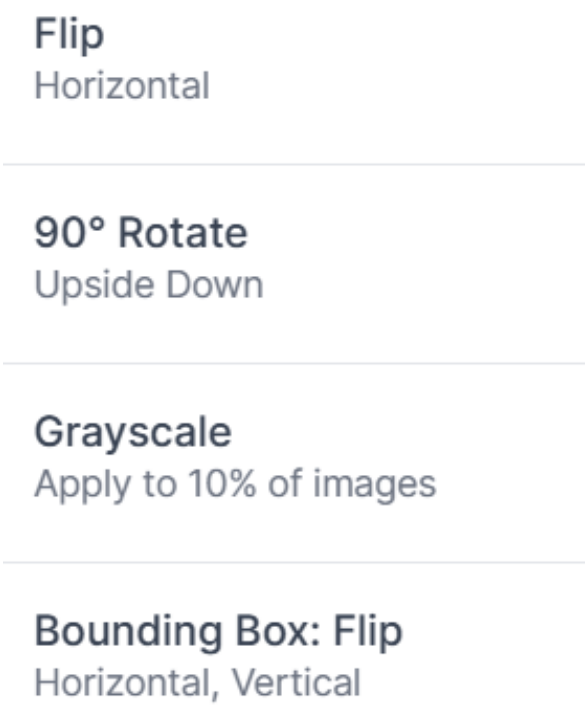

Fig. 2.13 Data Augmentation in YOLO

2.3.2 Training

Before conducting training with YOLO, it was necessary to use scripts to convert the annotations from VOC Pascal format, which were incompatible with YOLO due to a different bounding box annotation format. The training was conducted by applying data augmentation to increase the number of images. The operations depicted in Figure 2.13 constitute the data augmentation pipeline, which includes flip, rotation, grayscale, and bounding box flip. With the application of data augmentation, the dataset's image count was effectively doubled. Subsequently, YOLOv8 was trained over 25 epochs.

The average GPU RAM usage was approximately 8GB, with training taking approximately an hour using Tesla T4 GPUs. As evident from Figure 2.14, the training progressed satisfactorily. The loss for various detected objects consistently decreased with the number of epochs, while precision and recall increased as epochs advanced. The performance metrics were influenced by the dataset's limited size and imbalance, as illustrated in Figure 2.15. Indeed, certain categories, notably the "cross," are underrepresented, with few instances.

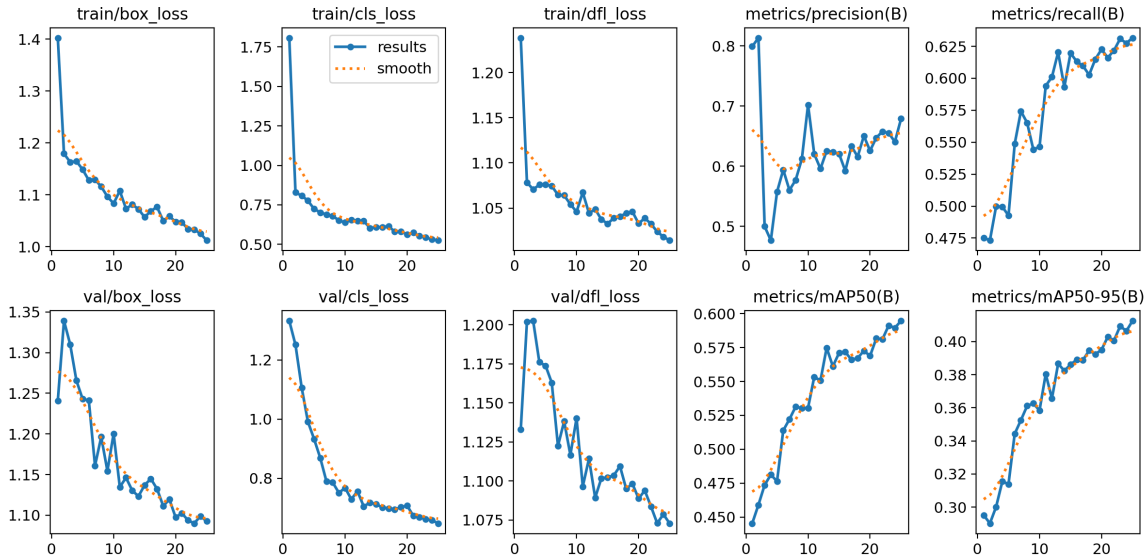

Fig. 2.14 YOLO training result

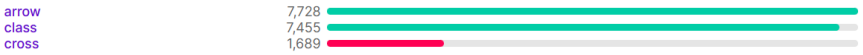

Fig. 2.15 YOLO dataset object

2.3.3 Performance and Usage

As anticipated from the theory, YOLO's performance yielded slightly inferior results compared to Faster RCNN, as depicted in Figure 2.16. For each identified class, the accuracy varies, primarily because these classes are abundant in the dataset and possess easily distinguishable shapes, rendering them relatively straightforward.

Conversely, arrows exhibit a moderate precision, even with their occasionally complex or confusing shapes, and their accuracy is respectable. However, the situation becomes more challenging for crosses, given their underrepresentation in the dataset.

The prediction speed is notably high, but it is highly dependent on the GPU responsible for executing the task.

Class	Images	Instances	Box(P	R	mAP50
all	61	1890	0.663	0.636	0.591
arrow	61	788	0.642	0.778	0.714
class	61	751	0.932	0.952	0.917
cross	61	351	0.414	0.179	0.142

Fig. 2.16 YOLO performance

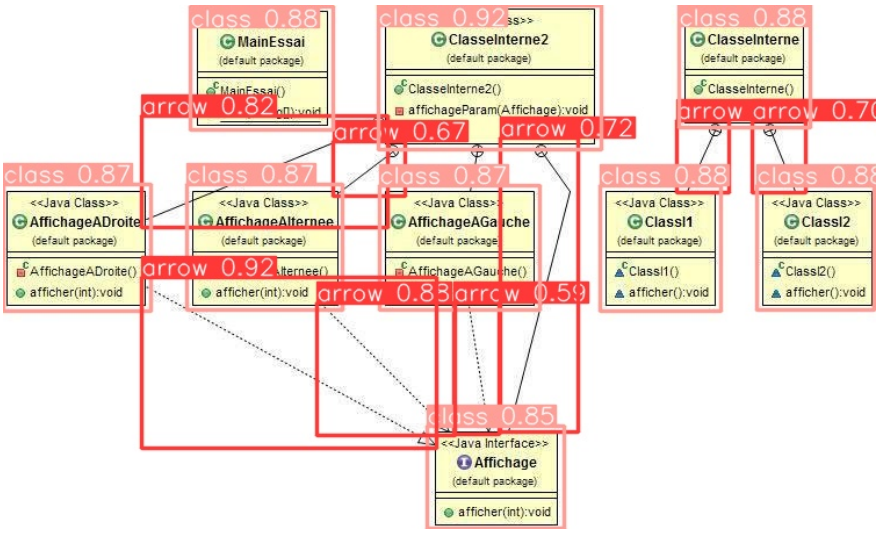

Fig. 2.17 YOLO example 1

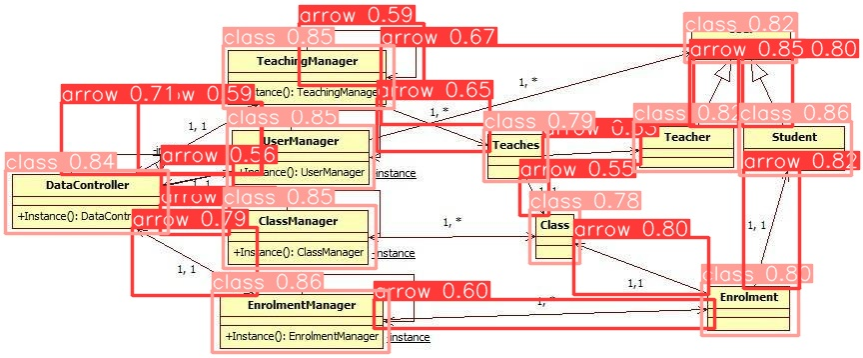

Fig. 2.18 YOLO example 2

The utilization process follows a conventional approach: the model 'models.pt' is loaded, and the segmenter is used, which will return the image's bounding boxes, along with their associated class and confidence percentage.

Below are some examples in Figure 2.17, Figure 2.18, and Figure 2.19. It is evident that in some cases, objects are misclassified, such as some simple arrows and crosses.

2.4 Alternative approach to segmentation

An alternative approach to segmentation, as an alternative to Faster R-CNN or YOLO, which utilizes bounded boxes, is the concept of semantic segmentation through pixels. This involves labeling individual pixels, enabling the identification and separation of objects or regions of interest within the image. This could potentially enhance the accuracy of identifying arrows

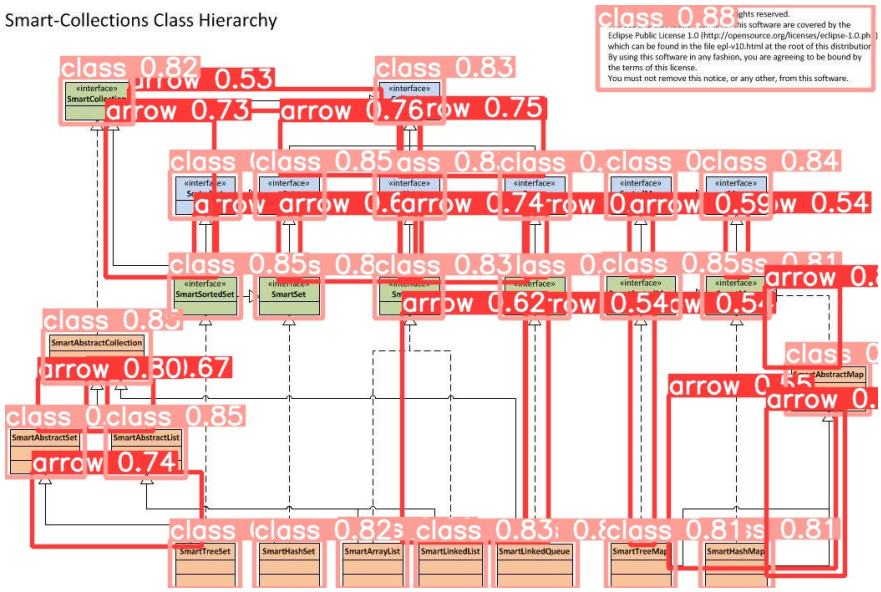

Fig. 2.19 YOLO example 3

and crosses, as individual pixels can achieve a finer granularity compared to bounded boxes, especially in complex images filled with intersecting lines within intricate and confusing scenarios.

Deep neural networks, such as U-Net or FCN, are often employed for semantic segmentation. U-Net is characterized by its U-shaped architecture and is trained using a dataset in which every pixel in the input image is labeled with the corresponding semantic class. These labels are typically encoded as binary masks, where each pixel in the mask is labeled as belonging to a specific class or not. During training, the network endeavors to approximate these masks for the training images.

The most common tools for dataset creation include the VGG Image Annotator (VIA): VIA is an open-source annotation software developed by the Visual Geometry Group (VGG) at the University of Oxford. It is versatile and can be used for pixel-wise annotation in images. In Figure 2.20, there is an example of annotation.

This method has not been tested due to time constraints, as estimating approximately a hundred hours of work for dataset creation would be disproportionate to the thesis workload. Nonetheless, it remains a valid approach, worth testing even with a small dataset at hand.

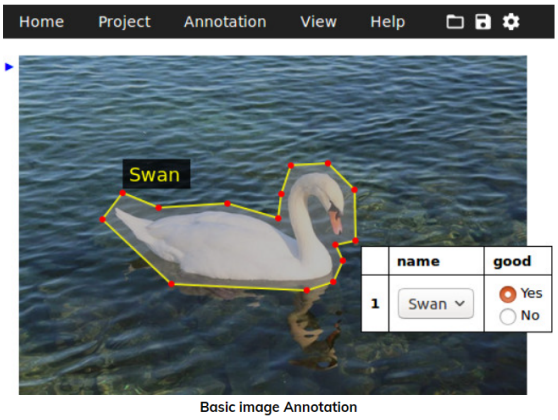

Fig. 2.20 Pixels labelling for U-Net

Chapter 3

Class Detection

Intro

In Unified Modeling Language (UML) diagrams, classes play a crucial role because they represent one of the fundamental concepts in object-oriented modeling and provide an organized structure for representing the components of a software system. They greatly influence the quality of an entire schema, so due to their criticality, it is necessary to extract the relevant features with particular attention.

For the segmentation and class detection part, we used a model specifically designed for extraction, and then we filtered the bounded boxes using fine-tuned parameters for the classes. The metrics we extracted and computed come from a thorough analysis and are partly taken from Paper 1, which focused heavily on the qualitative aspect that determines the quality of a UML schema.

In the end, the results and performance of this section are quite good, mainly because the model benefits from the simplicity of the characteristic class shapes.

3.1 Class segmentation

The segmentation was done using a Faster R-CNN, but we specified the optimal parameters for it. Once the model finds the bounded boxes in the image of the classes, they often overlap or are imprecise. Therefore, we need to go through a filtering phase, and the accuracy threshold for the model in this case is set to 0.70, while the threshold for IoU (Intersection over Union) is set to 0.2. Remember that IoU measures the common area between two bounding boxes and helps identify duplicate detected classes and model errors.

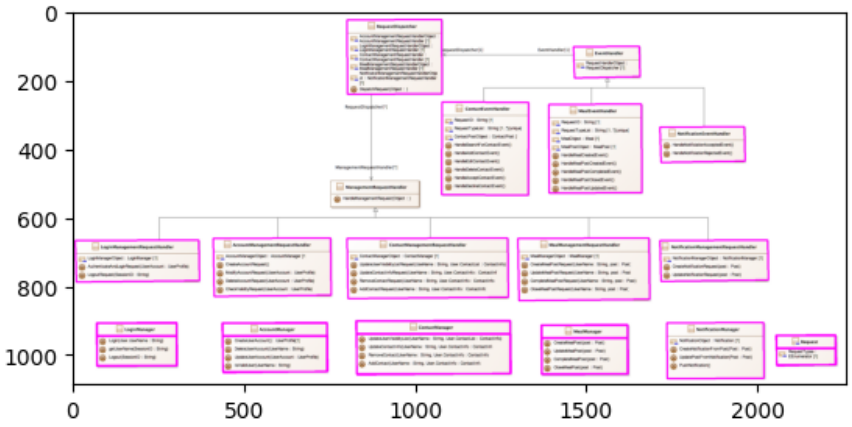

Fig. 3.1 Missed class with alternative algorithm

So, once the filtering phase is completed, we have the coordinates corresponding to the class boxes from which we extract the images. These segmented images will then be evaluated individually and used for processing the metrics that the model uses to assess the quality of UML diagrams. In the fig. 3.2, you can see the detected classes from which various classes are extracted, as shown in the example of the fig. 3.3. The high precision extends to the accuracy of metric calculations. Other techniques for class segmentation were also explored in addition to Faster R-CNN, and while some showed excellent performance, they did not perform well in all situations. In Figure 3.1, we can see the identification based on the Canny + HoughP algorithm, and in this case, one class was missed, which then affected the calculation of metrics relevant to the classifier. This latter method was faster and less computationally expensive than RCNN but was ultimately discarded due to its lower quality performance.

The performances are relative to the model in use , in general for the class segmentation was noted a good precision and recall in all of trained models. The segmentation of classes proves to be highly accurate even under adverse background conditions or image complexity. The processing time depends significantly on the hardware on which the Faster R-CNN runs, with GPUs resulting in faster processing.

3.2 Feature extraction

All the images related to the extracted classes are processed individually, and metrics are extracted. The following metrics aim to highlight the features that help determine the quality of a UML diagram and were inspired by Paper1. Here's a more detailed breakdown of the metrics and how they were developed:

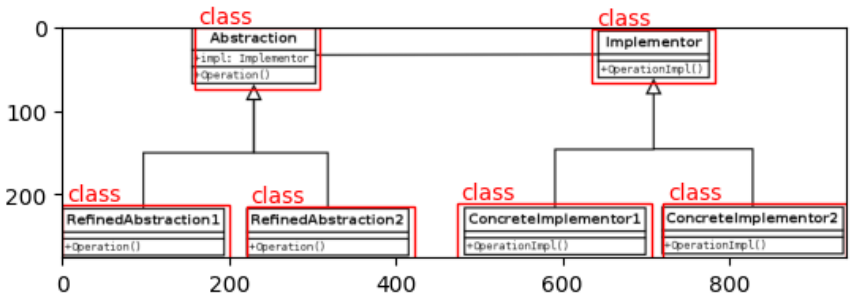

Fig. 3.2 Faster R-CNN in action for the class

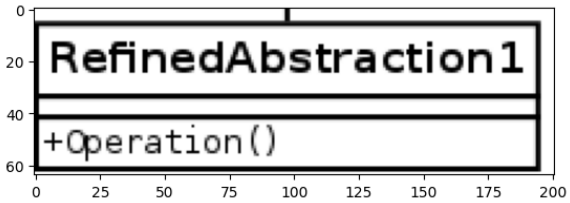

Fig. 3.3 Example of class detected

- **Rectangle coverage**: This feature represents how much of the total diagram area is covered by rectangle (which represent classes).This gives a ratio between 0 and 1
- **Aspect ratio**: This feature is calculated by dividing the width of the rectangle by the height of the rectangle.
- **Rectangle size variation**: this feature measures how much the rectangle sizes vary by calculating the standard deviation of the sizes of the rectangles in the scheme.
- **Rectangle size average** : This feature is the average area of all found rectangles normalized to the size of schema.
- **Number of rectangles**: this feature reflects the number of rectangles detected by image processing. This corresponds directly to the number of classes
- **Rectangle proximity**: the distance from its center to the center of the other rectangles is calculated for each rectangle. The average of this distance over all rectangles is used as the feature. The formula shows the definition of this feature:
$$\frac{\sum^{rectangles} shortestDistanceToOtherRectangle(rectangle_i)}{rectangle}$$
- **Rectangle distribution**: For this feature, the image is divided into four equal quadrants (obtained by drawing horizontal and vertical lines through the image's center point).

For each rectangle, the area of each of the four sections that it covers is calculated. These rectangle areas are summed up for the four sections, respectively, which gives each a value for how much of it is covered by rectangles. The variance of these values indicates whether all areas of the image are used. The formula shows the definition of this feature:

$var(\{rectangleCoverage(quadrant_i) \mid 1 \leq i \leq 4\})$

- **Ratio class/arrow**: the ratio between the number of class and the number of the arrow detected in the schema. Help to understand the complexity of the model.

Chapter 4

Arrow detection

Intro

In UML diagrams, the role of arrows is of particular significance as they identify relationships between various classes, which can take various forms. Therefore, studying them and extracting the appropriate metrics is crucial for the creation of a robust final classifier.

Recognizing this importance, prior to reaching the final implementation, numerous solutions and implementations were evaluated, as evident in the "Tested versions" section of this chapter, to maximize the accuracy of identification.

Many of the algorithms employed in this process have also been utilized in the papers examined in the State of Art section.

4.1 Version 1 : with Faster R-CNN

4.1.1 Arrow extraction

The extraction of arrows is dependent on the model using the Faster R-CNN architecture, which is known for its precision compared to the alternative YOLO.

As mentioned in the section about creating the extraction model, each specific extractor, in this case for arrows, has threshold parameters that adjust the sensitivity and precision of the extractor. In this case, the Accuracy has been set to 0.70, while the IoU (Intersection over Union) is set to 0.2.

Arrows are a fundamental part of UML diagrams and signify relationships between different classes. However, sometimes their identification is challenging due to the complexity and lack of clarity in the diagram. An example can be seen in the figure where the arrows are difficult to identify, but the model tries its best to identify the more visible arrows.

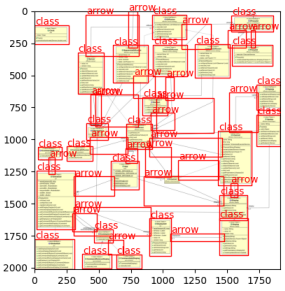

Fig. 4.1 Example of complex arrow segmentation

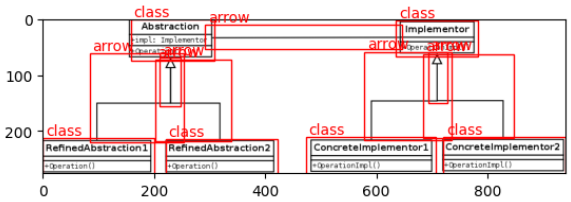

Fig. 4.2 Example of simple arrow segmentation

The performance is consistent with what was described in the segmentation model section, averaging around 55%, but with variations depending on the complexity of the analyzed image.

In general, the cropped box around the arrow is quite accurate, but there are often elements considered outliers, such as nearby classes or parts of other arrows, which will be removed in the preprocessing step discussed in detail in the next section.

In fig. 4.13 and fig. 4.14, is possible to see examples of more or less complex segmentation, and in fig. 4.3, there's an example of a segmented arrow.

4.1.2 Arrows identification

After the model segments and obtains the images containing the arrow, we arrive at the following crucial step, which must be performed for each segmented arrow. There are various problems that have been addressed, and we will analyze them one by one. Furthermore, this implementation is the result of numerous tests and errors, which will be described later.

The first step is to apply a Gaussian filter to eliminate obvious outliers or points.

The next step is to highlight the lines using the OpenCV HoughLinesP function, which is a more efficient probabilistic implementation of the Hough Transform for line detection and directly returns the endpoints of the detected lines.

First problem: As seen in Fig. 4.4, the Hough algorithm often identifies multiple lines stacked on top of each other for a single line, affecting metrics such as length. Consequently,

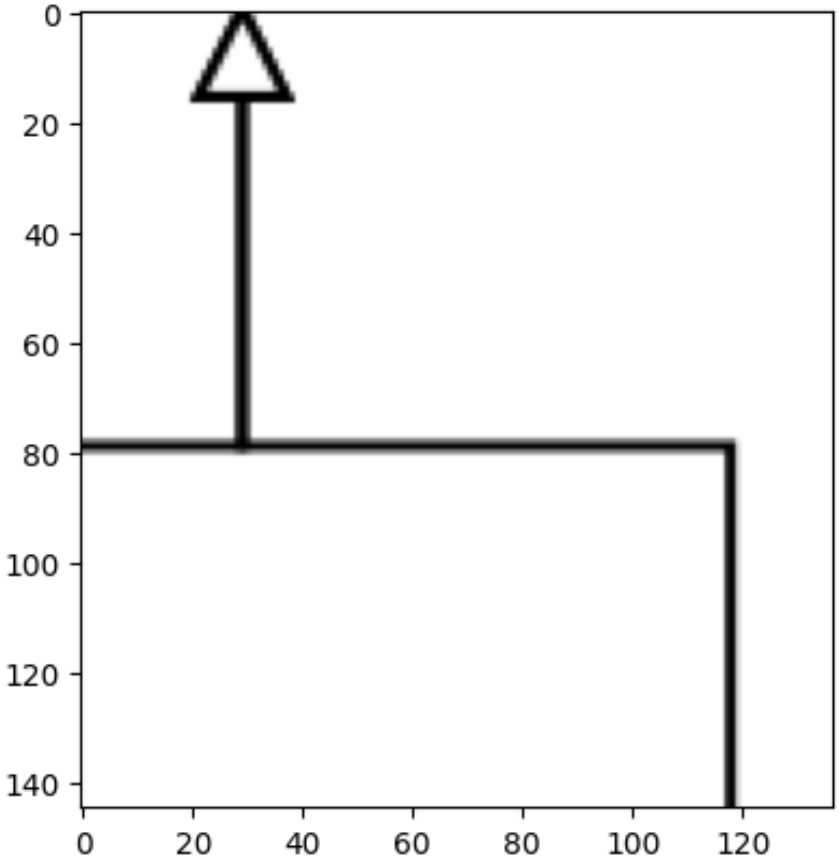

Fig. 4.3 Arrow segmented

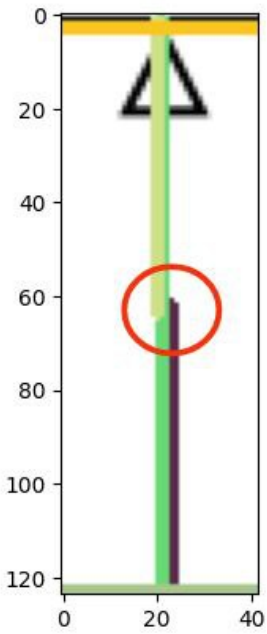

Fig. 4.4 Arrow after Hough function

you can observe a bundle of parallel lines on the same line, of which only one is correct. To address this problem, the lines are filtered by analyzing a circular neighborhood identified at the center of each bundle of lines. From this neighborhood, only the longest line is selected, effectively filtering out unnecessary lines. You can see the result after applying the filter in Figure 4.5. **Second problem**: Initially, the Hough function also selected small lines, even identifying as lines the labels of classes where only a portion appeared. To deselect those parts, it was sufficient to set a minimum segment length threshold equal to 5% of the shorter side of the image region containing the arrow.

Third problem: Another issue dealt with the arrows composed of multiple broken lines. In this case, the Hough algorithm identified them as separate single lines. To merge them, it was decided to create a "continuous" segment by analyzing all the endpoints of the individual segments and determining if there were other points from different segments within a 10-pixel neighborhood. If positive, the lines were considered "continuous"; otherwise, they were not. At the end of this computation, one or more "continuous" segments remained, and only the longest one was retained for feature analysis. Is possible to see the result in fig. 4.6.

After these steps, all the other parameters used for feature calculation can be computed. For each arrow, this includes the number of segments and the angle of each arrow.

In summary, after segmentation, each image is processed by eliminating outliers with the Gaussian filter. Each segment is analyzed, and multiple lines are removed. Then, continuous lines are calculated and the longest one is retained. Afterward, the metrics can be computed.

Fig. 4.5 Arrow after first filter

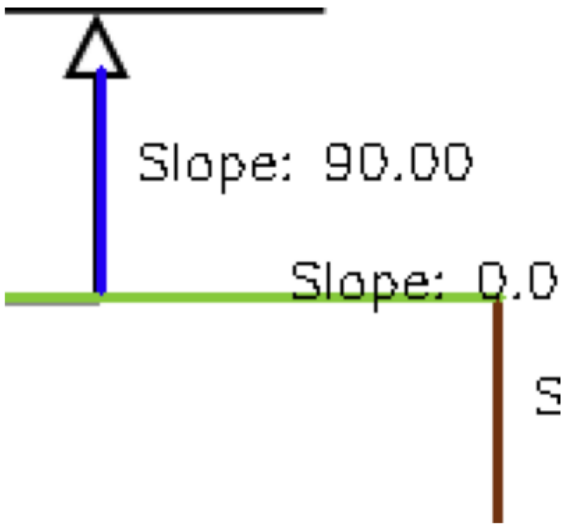

Fig. 4.6 Continuous arrow

4.2 Version 2 : without Faster R-CNN

After creating the model, it was observed that class identification worked very well, while arrow identification struggled in images composed of dozens or hundreds of overlapping lines. In fact, in some cases, the bounding boxes were unable to precisely identify a single line because they identified multiple lines simultaneously in complex situations. Since arrows are present in large numbers in each image, thus having a significant impact on the final result, a decision was made to intervene with an approach that can more finely identify the lines without the involvement of box identification using R-CNN.

The ultimate objective of this phase remains the identification of lines for the computation of the metrics required by the classifier.

4.2.1 Arrow extraction

In this initial phase, to ensure that the algorithms we will use for arrow identification do not become confused by the presence of lines, we aim to isolate the lines and remove any type of noise from the image, in this case, the classes.

Utilizing the R-CNN model with high accuracy for class identification, we employ it to erase the classes from the image, effectively rendering the area covered by the classes the same color as the background. This is achieved through filtering, retaining only the arrows that we will analyze for metrics.

We transition from original image in Figure 4.7 to Figure 4.8, where it is evident how all the identified classes are removed with high precision. Even in cases of very complex images, as the classes are consistently eliminated with high precision, the results were consistently excellent. An attempt was made to use a model not based on the Faster R-CNN for class identification and elimination, but it did not achieve the same level of precision and left more class residues after removal, especially in complex images. In Figure 4.9, you can observe an instance where a class is missed. Furthermore, even if we were to decide to eliminate the classes in Figure 4.10, you can notice that some noise remains due to the imperfect identification of the class boundaries.

4.2.2 Arrow identification

Once we have obtained the filtered image containing only the arrows for processing, we can initiate the line identification algorithm. The steps for line identification are as follows:

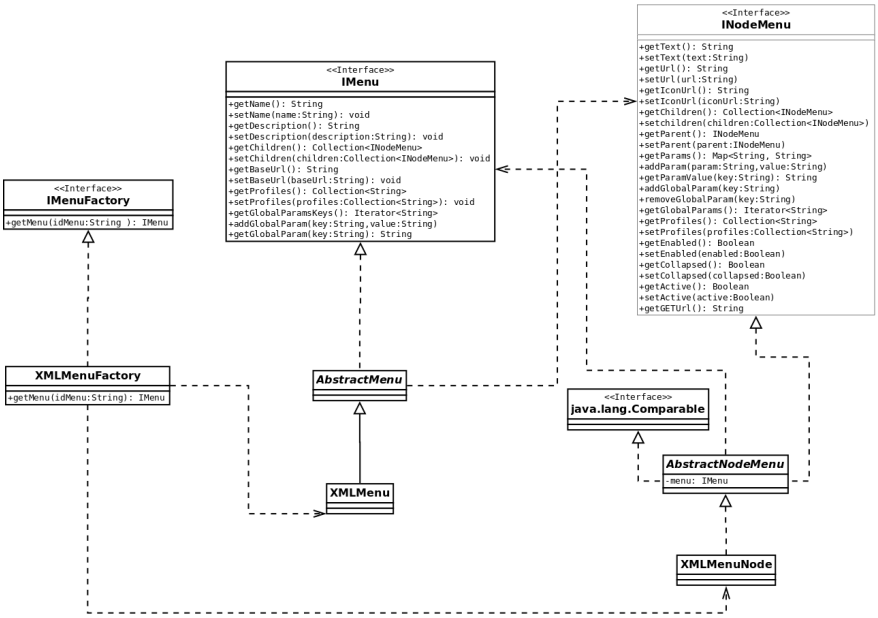

Fig. 4.7 Original image

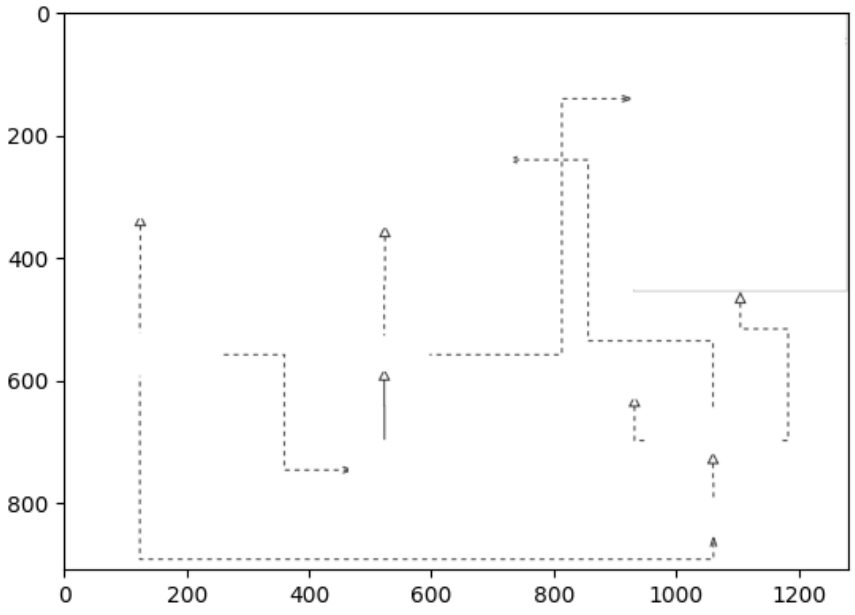

Fig. 4.8 Filtered image

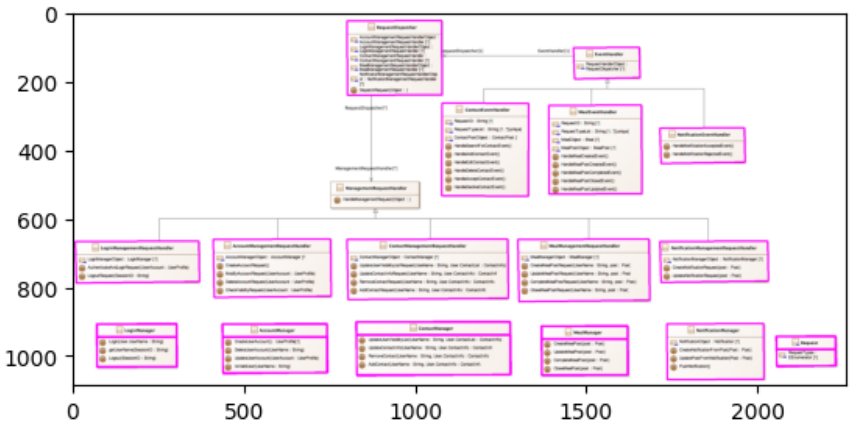

Fig. 4.9 Identification with missed class

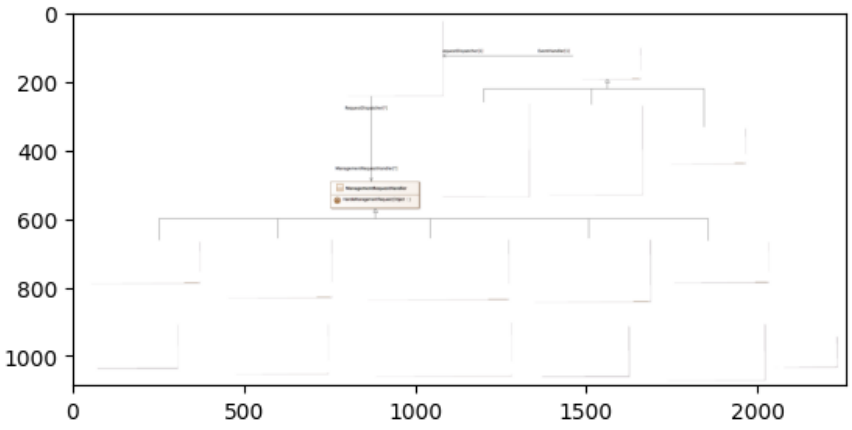

Fig. 4.10 Noisy image after deleting class

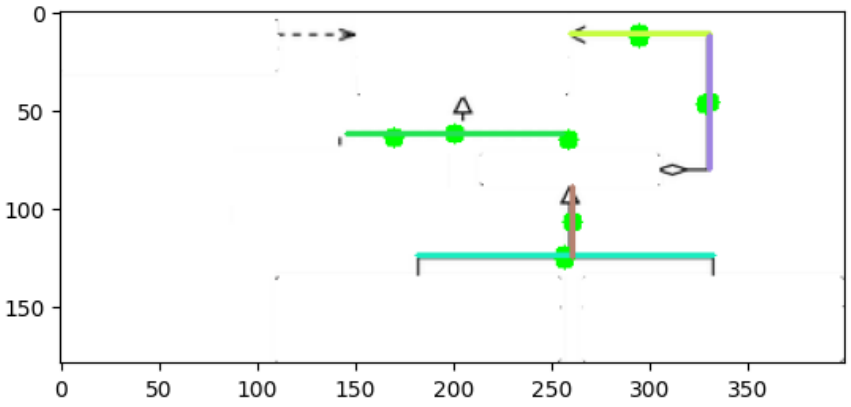

Fig. 4.11 Arrow identification simple image

1. Canny Edge Detection: The first phase, the application of Canny, is useful for detecting edges in the image. Canny is an edge detection algorithm that identifies points in the image where there are significant changes in pixel intensity.

2. Hough Transform: After obtaining the edge image, the Hough transform algorithm is employed to detect lines in this image. The Hough transform is a technique that can be used to detect lines. All parameters have been fine-tuned for our specific type of images that we need to analyze.

3. Line Filtering: This phase is necessary to filter the lines and achieve a more precise identification of segments. The Hough algorithm often makes identification errors. For each true segment, the algorithm selects it multiple times by overlaying the lines found, thus affecting the metrics, which consider a line multiple times. To address this issue, a workaround has been devised. A circle is drawn for each segment (in Figure 4.11, in green), and among the lines passing through them, only the longest one is retained. This filtering step helps eliminate multiple lines resulting from Hough's error.

All these identified lines are then used for computing the described metrics. The algorithm performs very well even in cases where there are numerous lines, as shown in Figure 4.12. However, this pipeline identifies "bends" rather than "arrows." Therefore, to count the number of arrows in the diagram, a clustering algorithm, specifically DBSCAN (Density-Based Spatial Clustering of Applications with Noise), is used. DBSCAN is an algorithm employed to identify groups of points in a data space based on point density.

In general, with this approach, an increase in feature extraction speed has been observed. This is because it does not require the use of specific hardware such as GPUs, as is the case with the R-CNN.

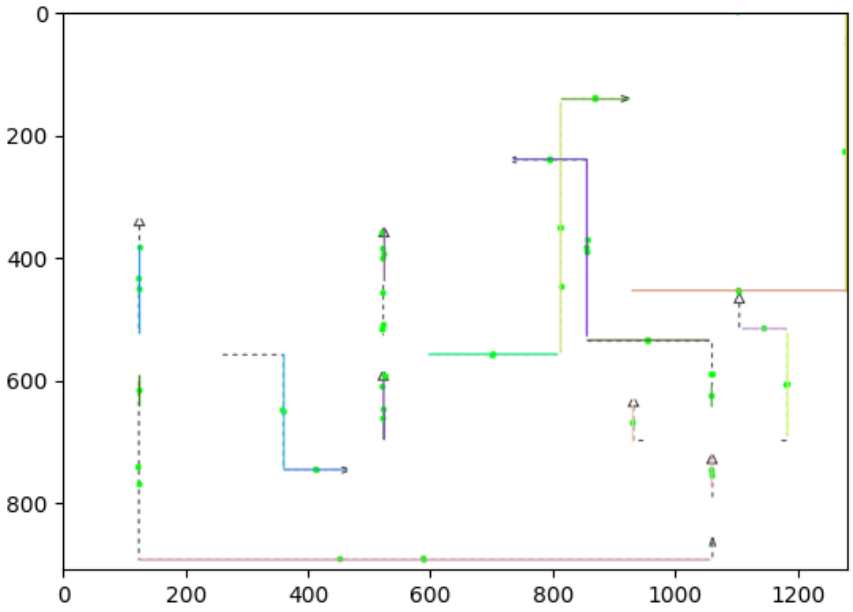

Fig. 4.12 Complex line identification

4.3 Feature extracted

For each UML diagram, all arrows are first detected and segmented. Then, each image undergoes processing using the algorithms described in the preceding chapter with the ultimate goal of extracting the following metrics. These metrics will be used for training and utilizing the UML diagram quality classifier.

The described metrics have been developed, drawing inspiration from Paper 1, which is detailed in the state-of-the-art chapter. Here's a detailed list of the calculated metrics:

- **Line angles** : The deviation angle from orthogonality, i.e., how far from being horizontal or vertical, is calculated for each line. If a line is more than 45◦ degrees from being horizontal, it is less than 45◦from being vertical and vice versa. The formula shows the definition of this feature:
  $\frac{\sum_{i=1}^{\#bends} angle(line_i)}{\#lines}$
- **Line orthogonality**: A line is orthogonal if it is exactly horizontal or vertical. A margin of error of 1◦ is used to allow for small deviations due to the quality of the image and the image processing. The number of orthogonal lines is divided by the total number of lines to get a ratio between 0 and 1. The formula shows the definition of this feature:

$$\frac{\sum_{i=1}^{\#lines} \begin{Bmatrix} 1 & if angle(line_i) < 1^\circ \\ 0 & otherwise \end{Bmatrix}}{\#lines}$$

- **Line bends**: Lines are represented as straight segments, and the number of bends on a line is one less than its number of segments. The formula shows the definition of this feature:
$\frac{\sum_{i=1}^{\#bends} bends(line_i)}{\#lines}$
- **Average line length**: This feature is the average length of all found lines.
- **Line length variation** : This feature measures how much the line lengths vary by calculating the standard deviation of the lengths.
- **Longest line** :This feature is the length of the longest found line.
- **Shortest line** : This feature is the length of the shortest found line.
- **Number of arrows**: This feature reflects the number of lines detected by image processing.

4.4 Tested versions

Before arriving at the current solution, many implementations were considered, which I will briefly explain, utilizing diverse techniques and algorithms.

4.4.1 Solution with OpenCV countours

In this version, after segmenting the various arrows, the outlines were highlighted using the "findContours" function, which is part of the OpenCV computer vision library. Before applying this function, a thresholding operation was performed to obtain a binary image. The "findContours()" function in OpenCV utilizes the Suzuki and Abe algorithm for topological structural analysis of digitized binary images along the edge. This algorithm is described in the document "Topological Structural Analysis of Digitized Binary Images by Border Following" by Suzuki, S., and Abe, K., published in 1985. It works by scanning the image line by line, pixel by pixel, and identifying connected components. A connected component is a set of adjacent pixels that share the same intensity or color. This algorithm offers several advantages, such as efficiency, as its complexity is linear, making it suitable for real-time applications and large-scale image processing tasks. The issue encountered in this

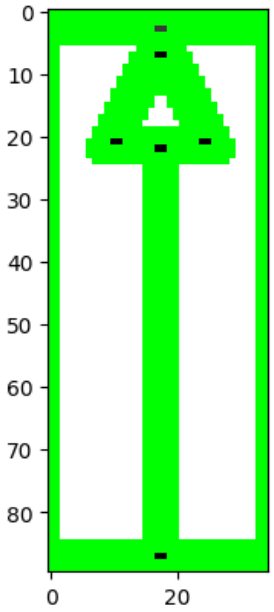

Fig. 4.13 simple arrow

version was the inability to distinguish the outline of the arrow from the outliers, mainly because the arrow's outline is unique, and the operation could not be performed effectively. Furthermore, on the same edge, multiple layers of contours often overlapped. Therefore, using this methodology distorted the true length of the analyzed arrow, prompting the search for an alternative approach. In fig. 4.13 and fig. 4.14 is possible see the algorithm at work with two types of arrows.

4.4.2 Solution with start/end point

In the following version, for each arrow identified by a bounding box, the starting and ending points of the arrow are found using an algorithm that identifies the farthest points and connects them to create a new artificial arrow, used for calculating subsequent metrics. Before identifying these points, the image was processed by adjusting the contrast to make the points more evident. The result can be observed in fig. 4.15 and fig. 4.16. Multiple issues were encountered in this implementation. One of these issues was related to calculating the arrow's length, as there was a significant error for images containing arrows composed of broken lines, which appeared identical to oblique arrows. An attempt was made to address this problem, but the only solution found required a second algorithm to find the contours between the two points, leading to a problem similar to the one in the initially discarded implementation. Another problem detected was the presence of outliers outside the arrow or

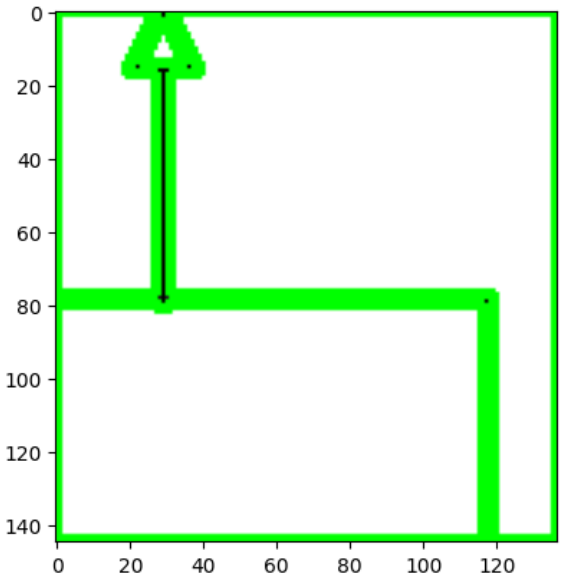

Fig. 4.14 complex arrow

pieces resulting from false segmentation belonging to other elements. The farthest points turned out to be distorted and not consistent with the arrow.

4.4.3 Solution with connected point

This implementation primarily utilizes the "connectedComponentsWithStats" function from OpenCV to analyze images containing arrows. This function is typically used for analyzing connected components in binary images. The algorithm is an algorithmic application of graph theory used to determine the connectivity of blob-like regions in a binary image. Once connected elements are found, the process proceeds with filtering. The issue detected was that the algorithm created a single line connecting all points, even those not directly connected. Therefore, in this step, the entire line was removed. As can be observed from the fig. 4.17 and fig. 4.18 , related to this implementation, calculating the length of the lines that were supposed to overlap the arrows proved ineffective because, in addition to being distorted, there was also a varying density of different lines among similar arrows, resulting in variable and non-fixed errors. Unexpectedly, excellent behavior was noted concerning outliers or noise.

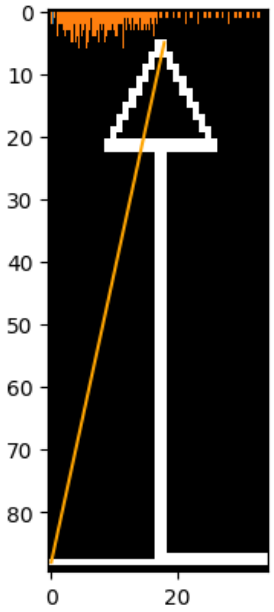

Fig. 4.15 simple arrow

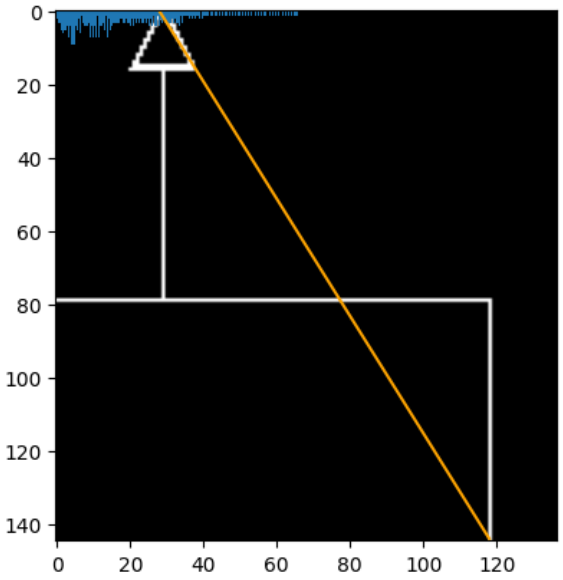

Fig. 4.16 complex arrow

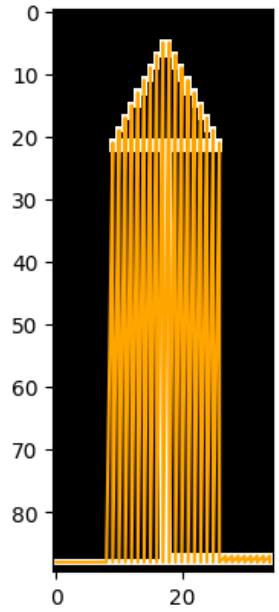

Fig. 4.17 simple arrow

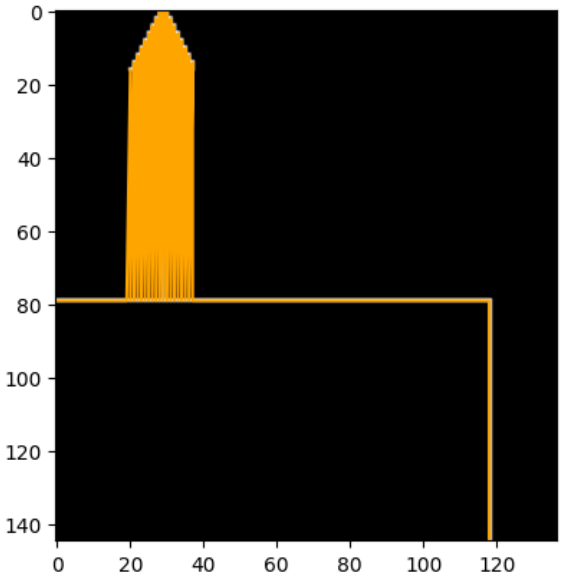

Fig. 4.18 complex arrow

4.4.4 Solution with Canny algorithm

The following implementation presents a variant that uses the Canny algorithm, employed in an attempt to highlight all the edges of the arrows. cv2.Canny() uses gradient detection and hysteresis to detect edges in an image, producing a binary image where points represent detected edges. The Canny edge detection algorithm consists of four main steps:

1. Reduce noise using Gaussian Smoothing.
2. Compute the image gradient using the Sobel filter.
3. Apply Non-Maximum Suppression to keep only local maxima.
4. Finally, apply Hysteresis thresholding using two threshold values, Tupper and Tlower, allowing the sensitivity of edge detection to be adjusted according to specific application requirements.

Going into more detail:

The Sobel filter, sometimes called the Sobel-Feldman operator, is used in image processing, particularly within edge detection algorithms, where it creates an image that emphasizes edges. It was developed by Irwin Sobel and Gary M. Feldman, colleagues at the Stanford Artificial Intelligence Laboratory (SAIL).

Hysteresis Thresholding helps distinguish true edges from potential ones. If a pixel has an intensity (or gradient, in the case of edge detection) above the upper threshold, it is immediately marked as an edge. If a pixel has an intensity below the lower threshold, it is immediately discarded. If the pixel intensity falls between the two thresholds, it is marked as an edge only if connected to a pixel above the upper threshold.

The results of the following algorithm can be observed in the images. However, an issue was encountered in the calculation of metrics because it is highly sensitive to outlier lines. If the segmentation is not precise, pieces of classes and/or text remain within the image, which posed a problem in calculating the length.

Regarding the double contour of an arrow (inner/outer), it has a minimal effect on distance calculation because all arrows will have the same type of deviation, which the model does not consider, even though it does not reflect the "true" length of the arrow.

4.4.5 Solution with Harris corner + Canny

The goal of this implementation was to use the Harris algorithm for corner detection, which identifies the sensitive points in the images. Once the corners were identified, the previous

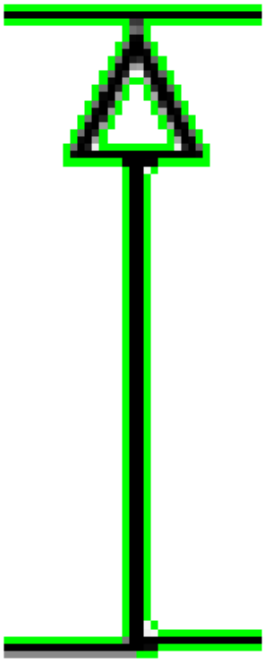

Fig. 4.19 simple arrow

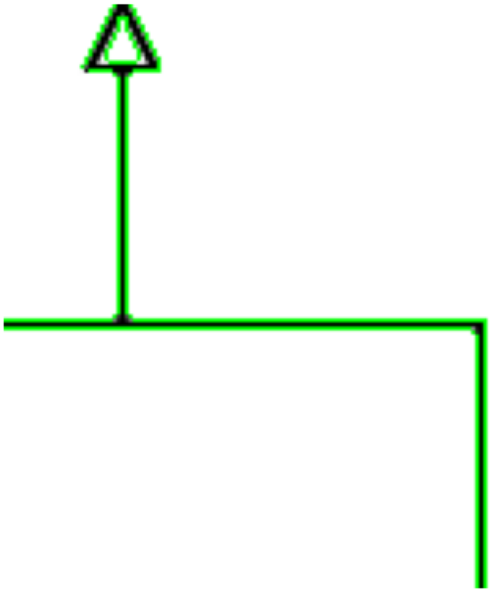

Fig. 4.20 complex arrow

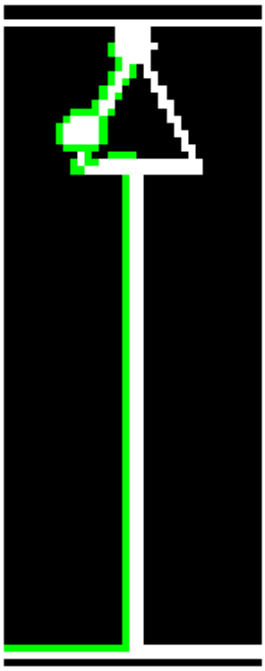

Fig. 4.21 simple arrow

Canny algorithm could be used to highlight the edges, eliminating and addressing the problem of outliers and the double edge.

Let's briefly analyze how Harris corner detection works:

Harris Corner Detection is a corner detection algorithm developed by Chris Harris and Mike Stephens in their 1988 paper "A Combined Corner and Edge Detector" . The underlying idea of the algorithm is that corners are regions of the image with a significant intensity variation in all directions. The algorithm calculates the intensity difference for a displacement of (u, v) in all directions and seeks to maximize this function for corner detection. This is done using Taylor expansion and some mathematical manipulations to obtain a final equation that can be used to calculate a score for each image window. A window with a score above a certain threshold is considered a "corner."

The problem encountered this time was the miss-detection of corners in several analyzed images, leading to incorrect highlighting of arrows. However, in cases where the corners were correctly detected, the algorithm accurately calculated the path and length of the arrow. As can be seen from the images, the detected corners are often incorrect, resulting in a miss-detection of edges as well as is it possible to see in fig. 4.21 and fig. 4.22.

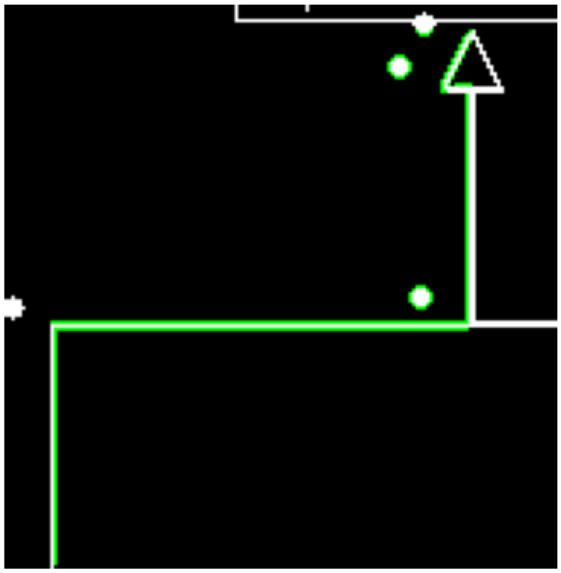

Fig. 4.22 complex arrow

4.4.6 Solution with start/end point + Canny

In this implementation, an attempt was made to combine the approaches that had yielded the best results so far, namely Canny and the approach that highlights the start and end points. Therefore, this version is divided into two phases, after processing the image by adjusting the contrast and highlighting the edges.

In the first phase, the starting and ending points of the arrow are determined by finding the farthest points from each other. In the second phase, the image is processed to find the edges using the Canny algorithm, delimited by the points found earlier.

The problem highlighted by the combination of these two implementations is that sometimes the Canny algorithm only highlights the outer edge, while other times it highlights both the outer and inner edges. This caused a significant random error that affected various metrics, including arrow length.

In fig. 4.23 and fig. 4.24 , related to this implementation, is it possible to observe that some edges are highlighted only externally, while others are highlighted both internally and externally.

4.4.7 Solution with HoughLinesP algorithm

In this implementation, an attempt is made to make the best use of the method offered by OpenCV called HoughLinesP. In short, this function: It's an implementation of the Hough Transform algorithm for detecting lines in an image. This function is a more efficient and

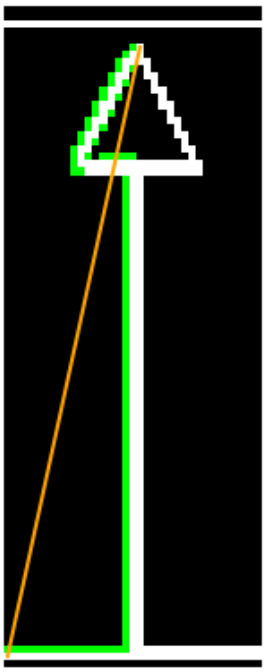

Fig. 4.23 simple arrow

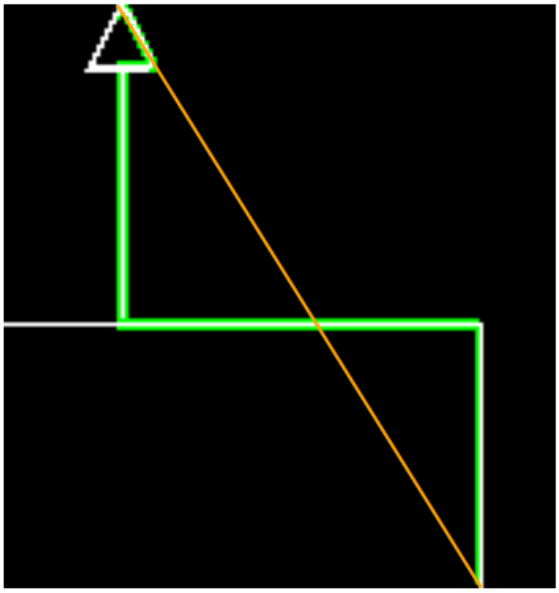

Fig. 4.24 complex arrow

probabilistic version of the Hough Transform algorithm (OpenCV: Hough Line Transform). Here's how the algorithm works:

1. First, the image is pre-processed for edge detection.
2. The algorithm creates a 2D accumulator to keep track of the intersections between curves from each point in the image. The rows of the accumulator represent the parameter ρ (the perpendicular distance from the origin to the line), and the columns represent the parameter θ (the angle formed by this perpendicular line and the horizontal axis measured counterclockwise).
3. For each point in the image, the algorithm calculates the family of lines passing through that point and increments the value in the corresponding cells at (ρ, θ) in the accumulator.
4. If the number of intersections exceeds a certain threshold, the algorithm declares that there is a line with the parameters of the intersection point.
5. The function returns the endpoints of the detected lines.

Once the working principle is understood, it was used and implemented. However, a problem was detected that skewed the calculated metrics, causing some issues. The main problem is that it highlights the same line multiple times as in fig. 4.25, which causes problems in length calculation. Moreover, this approach also detects outlier lines resulting from not very precise segmentation. Another issue to address is lines formed by multiple broken lines as in fig. 4.26.

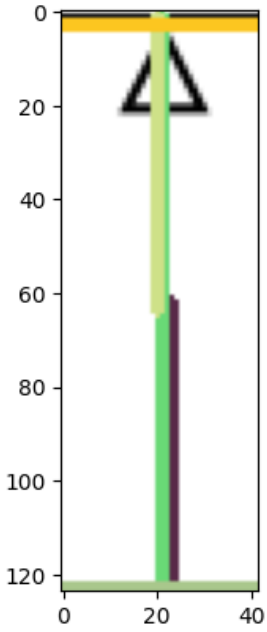

Fig. 4.25 simple arrow

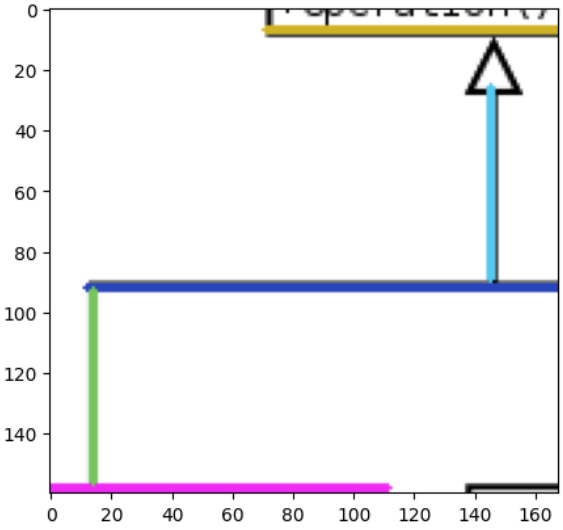

Fig. 4.26 complex arrow

Chapter 5

Cross line detection

Intro

The intersecting lines in UML diagrams play a crucial role in assessing their quality. There are two types of cross lines: those resulting from disorder, where lines intersect randomly and have non-orthogonal angles between them, and those deliberately intersecting to create more order and improve readability, in which case the lines are orthogonal.

Therefore, the characteristics of intersecting lines were analyzed and extracted because they are fundamental in classification. Generally, the pattern that can be observed is that as the number of disorganized crosses increases, the UML tends to become less readable.

To analyze these types of lines, a pipeline similar to the previous ones was used. It begins with an initial phase of segmentation using the Faster RCNN, followed by a subsequent phase of more accurate line identification. Finally, the actual evaluation metrics are extracted. These metrics will then be utilized by the final classifier.

5.1 Cross extraction

For this particular feature, we also rely on the model trained with the Faster RCNN architecture, capable of identifying cross lines. However, in this specific case, the model encounters greater difficulty for several reasons.

One of the primary challenges is due to the model being trained with "cross" as a minority class. Cross lines are much rarer compared to classes or arrows, as they are characteristics of a few complex UML diagrams. This indicates underfitting of the model for that particular class.

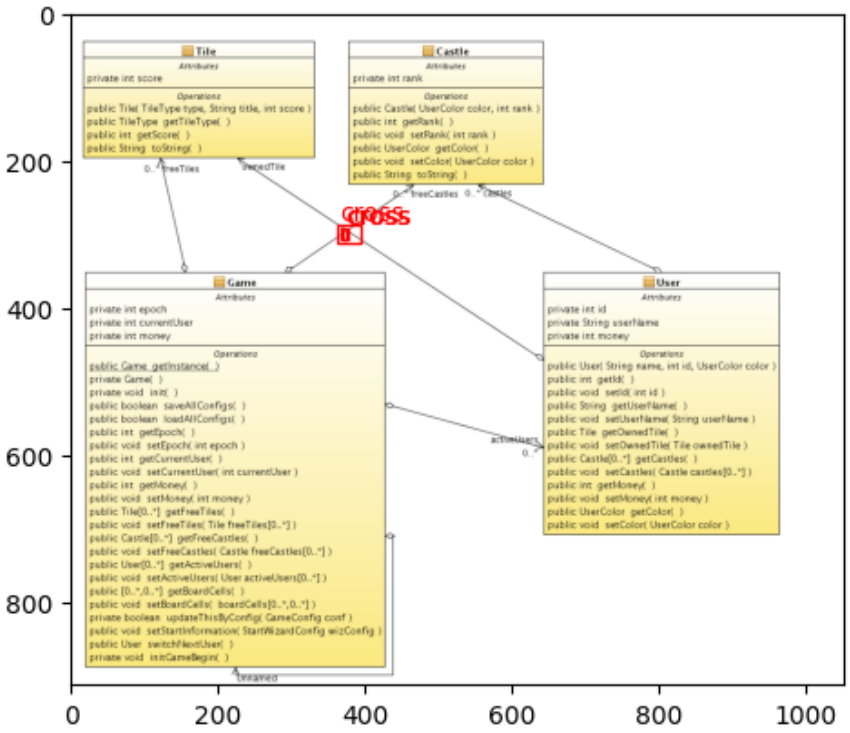

Fig. 5.1 Cross identification before filtering

Additionally, the model may occasionally confuse other elements that are not considered "crosses." For example, it might misidentify the area near a class that connects with an arrow, creating a shape resembling a cross along the class's border. Another example could be the uppercase letter "T," which creates a cross-like shape falling into the "ordered" category.

The segmentation performance is better than the previous model because fine-tuning was performed on the thresholds, and attempts were made to increase training by implementing augmentation. After fine-tuning, the model's thresholds were determined, with an accuracy value of 0.12 and an IoU value of 0.00 , the result is the cross identification in fig. 5.1. After identifying the boxes, they were filtered by selecting based on IoU metrics , in this case setted to 0.00 to have largest cross box that not intersect with other cross box , as is possible to see in fig. 5.2. This was done to accurately choose the image portion containing the cross (fig . 5.3). This step was added to the pipeline because the accuracy of the threshold parameter is low for identifying all crosses, and a filter is necessary before the processing phase.

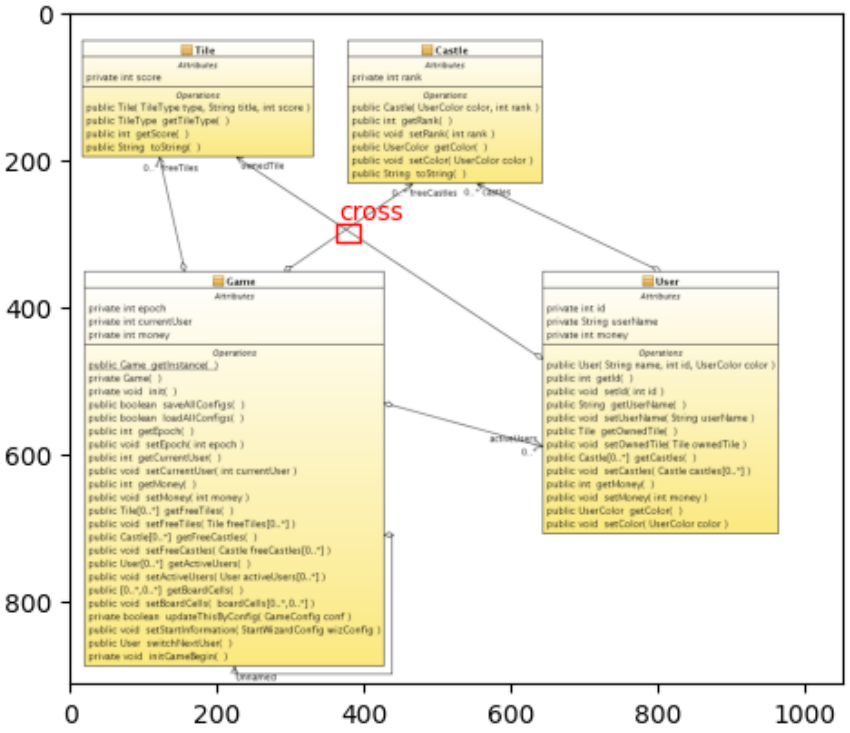

Fig. 5.2 Cross identification after filtering

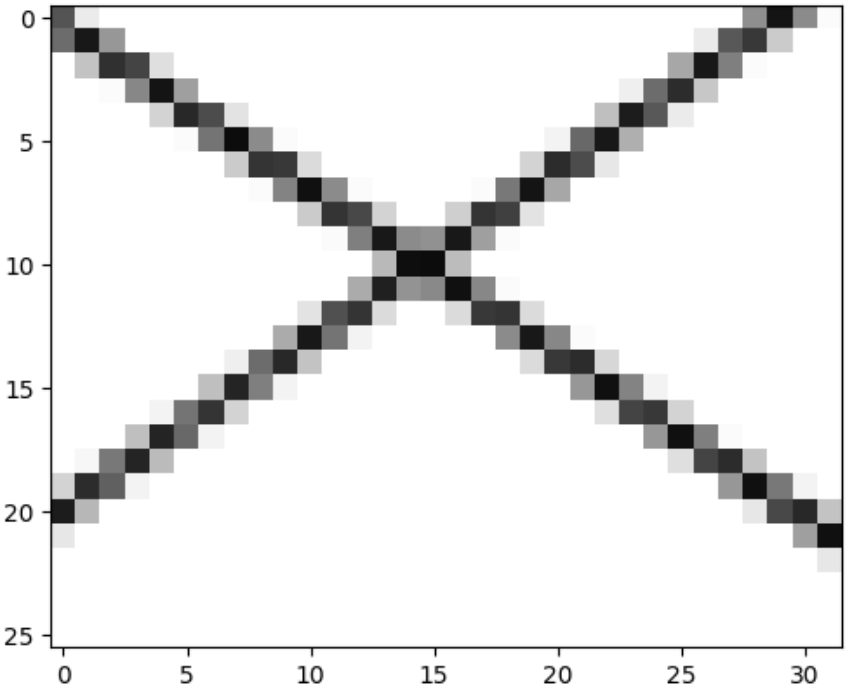

Fig. 5.3 Filtered detected cross

5.2 Cross identification

Once the image portion containing the "cross" of interest has been segmented, a more precise object highlighting is required. To achieve this, noise is initially reduced using a Gaussian filter. Subsequently, the HoughLinesP function from OpenCV is employed, which is an efficient implementation of the Probabilistic Hough Transform for line detection. This very function was used in arrow detection and was also mentioned in the papers analyzed in the State of the Art for line detection.

In the fig . 5.4, an example is shown from which useful information can be extracted, such as the angle. This angle is crucial for discriminating between two major categories of "crosses": orthogonal and non-orthogonal. The former indicates an organized UML diagram with deliberately combined arrows to enhance readability, while the latter results from disorder, where angles are random and non-orthogonal, signifying confusion in the diagram.

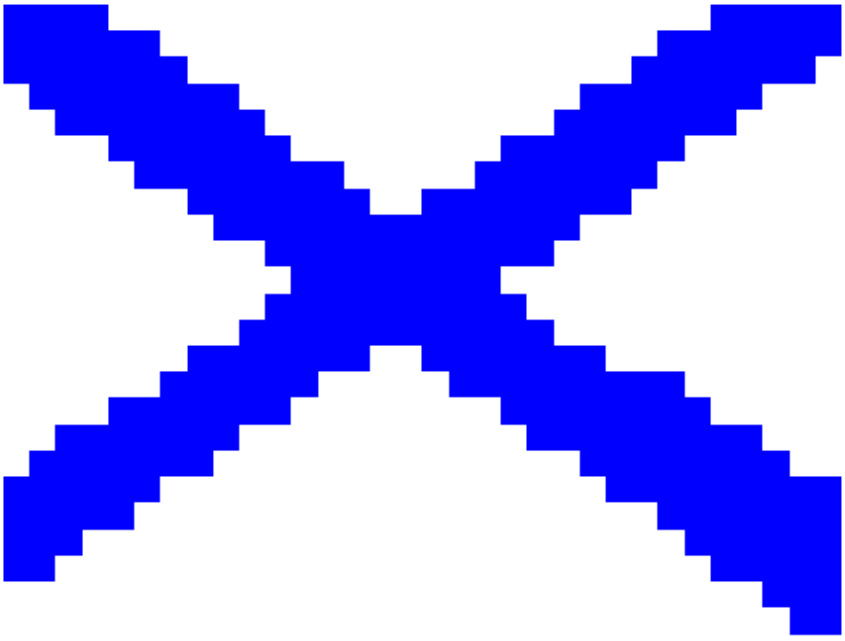

Fig. 5.4 Cross identified with HoughLinesP algorithm

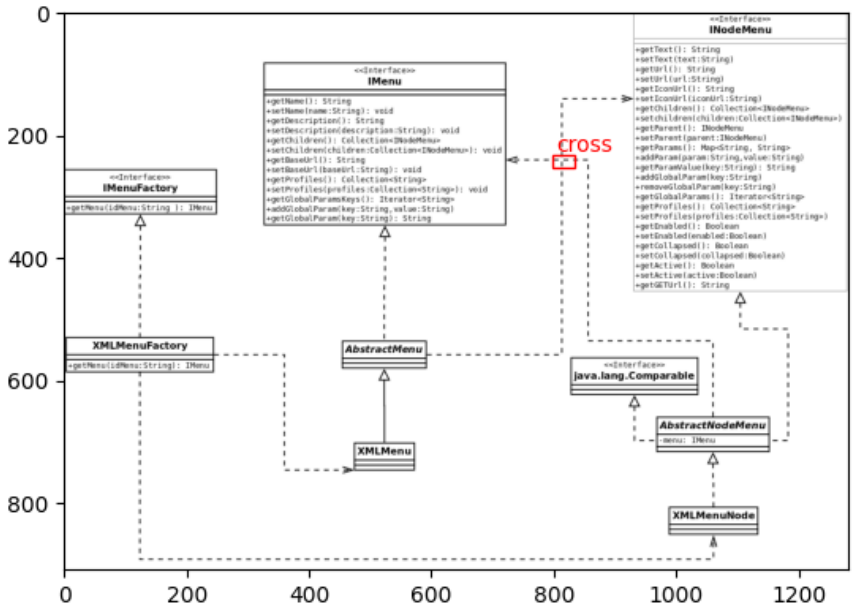

Fig. 5.5 Cross identification with dotted line

5.3 Feature extracted

The evaluation metrics extracted from this crucial element, such as the "crosses," were inspired by Paper1 analyzed in the State of the Art. After identifying, filtering, and selecting the lines, the following metrics are computed for each image:

- **Line crossings**: The number of line crossings in a diagram.
- **Crossing angles**: For each line crossing, the crossing angle is calculated — this is a value in the range from 0○ to 90○. The average crossing angle is then calculated as the feature. The formula shows the definition of this feature:
$\frac{\sum_{i=1}^{\#crossings} angle(rossing_i)}{\#crossings}$

5.4 Algorithm at work : special case

Case dotted line :

One of the potential issues could have been the identification of lines forming a cross but with dotted lines. This is because there is not a true intersection of the lines. However, the software is able to identify and segment even this type of lines, as can be seen in Figure 5.5 and figure 5.6. Thanks to this robustness, it is possible to calculate all the metrics even for this type of intersections. Once these elements are identified, the processing by HoughLinesP

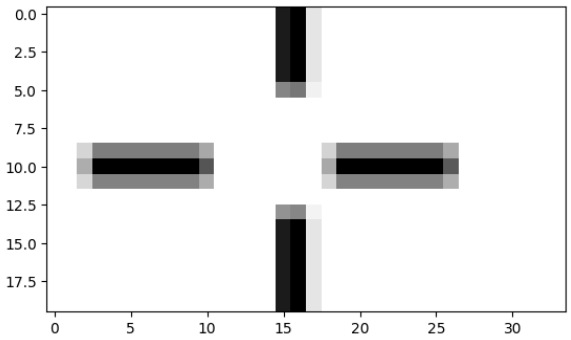

Fig. 5.6 Segmented dotted cross

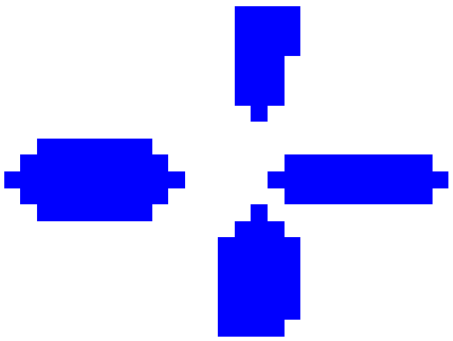

Fig. 5.7 Processed segmented cross

begins, which selects the segments as shown in Figure 5.7 and computes the slope for each line for the computation of the metrics required by the classifier.

Case multiple line :

In this instance, the image is more complex. Indeed, there are multiple lines intersecting, combining dotted lines with continuous lines. In this scenario, the algorithm performs reasonably well, successfully identifying multiple intersection points, for which it will subsequently compute the metrics for the crosses. Figure 5.8 displays the processed image.

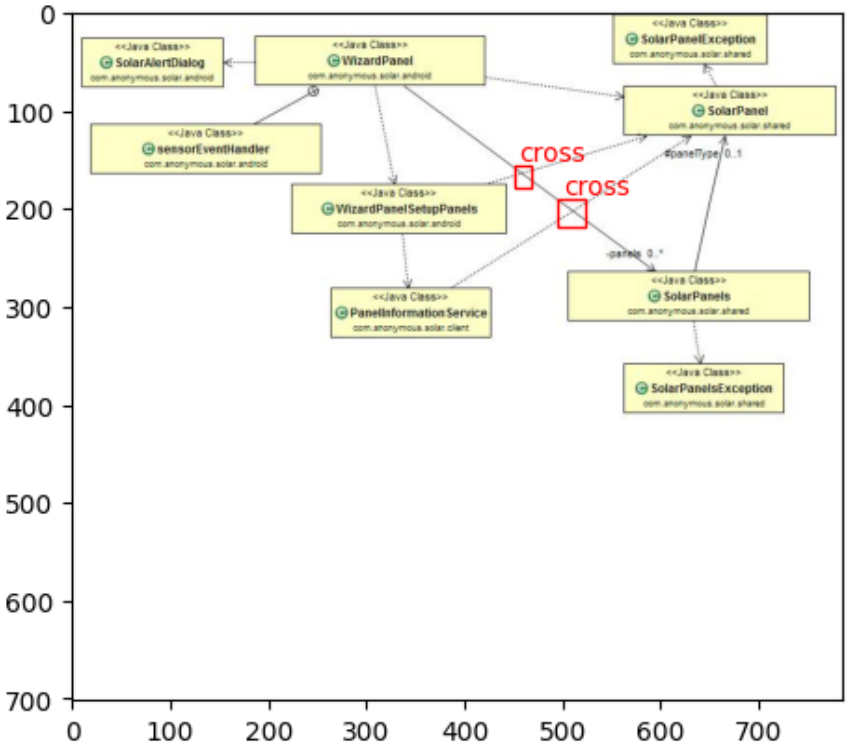

Fig. 5.8 Multiple cross identification

Chapter 6

Image info detection

Intro

In addition to the specific features of UML diagrams, other more general characteristics of the images were considered. This is because an image might be so blurred that it becomes impossible to read the content, or in another scenario, the image might have very high contrast, making it difficult to read the text (e.g., electric blue background with white text). While a text extraction algorithm might not encounter any difficulties, our human eyes certainly would.

6.1 Feature Extracted

The following metrics described below aim to capture the overall characteristics of the image, which, in combination with all the other specific metrics related to the UML domain, will be useful for the final classifier in image evaluation.All parameters were normalized to allow comparison between different image size. Below are the extracted metrics:

- **Histogram** : the histogram of an image represents the distribution of pixel values in that image. In a gray scale image, pixel values range from 0 (black) to 255 (white). It is useful as a metric because sometimes different colors are used, for instance, to distinguish various parts of a project, such as in the JAVA model-view-control pattern. In fig. 6.1 an example of histogram , the peaks correspond to different image color. This feature, being composed of an array of 255 elements, has been processed to yield two smaller integer features, namely the number of colors (n_color) and the average peak intensity (avg_intensity).
- **Image ratio** : the aspect ratio of an image is the ratio of its width to its height.

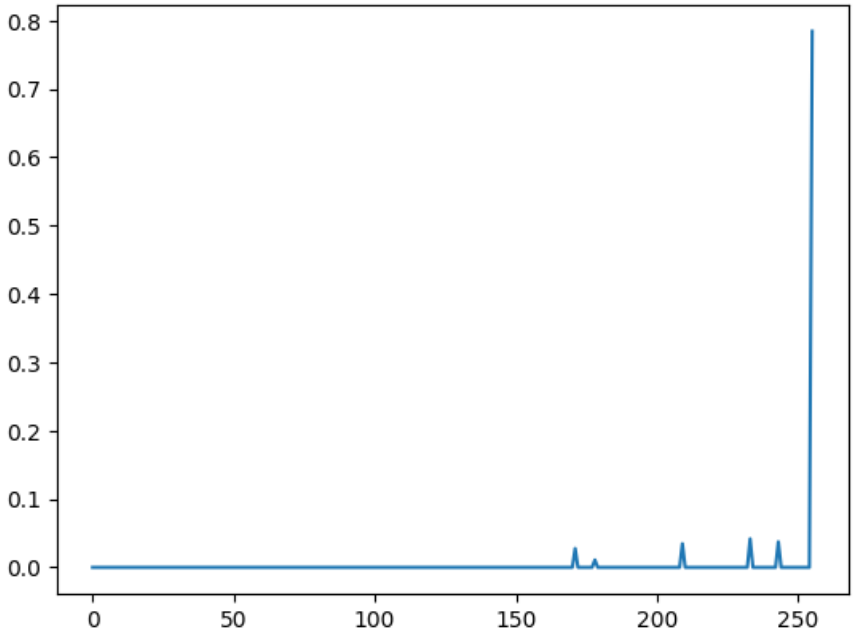

Fig. 6.1 Histogram of multi color image

- **Contrast** : the contrast of an image refers to the difference in color and brightness between different points in the image. High contrast could potentially have a negative impact on the image evaluation.
- **Image blur** : A metric for image blur often involves using the Variance of Laplacian method. If the normalized blur value obtained from the `get_blur` method is low, it indicates that the image is likely sharper.
- **Complexity** : this feature is the sum between the number of class , number of arrows and number of cross detected in the image. Hight complexity could potentially have negative impact on the image readability.

Chapter 7

Text detection

Intro

In the evaluation process of a UML diagram, a crucial aspect is the text contained within the various classes. These classes represent the context around which the entire diagram revolves. Therefore, with the help of the text, semantic coherence can be assessed, and any non-relevant classes can be highlighted. To accomplish this, the process needs to be divided into three phases: the first being text extraction, the second involving preprocessing, and the third phase focusing on context elaboration and extraction.

7.1 Text extraction

This phase is a critical point, so achieving high accuracy is of paramount importance. For this task, we preferred to use a pre-trained model called easyOCR, which, given the image of the class previously segmented by the model, extracts and returns the contained text.

EasyOCR is a Python library for utilizing a ready-to-use OCR model. With this library, you don't need to worry about the preprocessing and modeling phase. EasyOCR is built using Python's deep learning library and PyTorch, and having a GPU can accelerate the entire detection process. The detection part uses the CRAFT algorithm, and the recognition model is CRNN. It comprises three main components: feature extraction (currently using ResNet), sequence labeling (LSTM), and decoding (CTC). CRAFT stands for Character Region Awareness for Text Detection. It is a text detection algorithm that efficiently explores each character region and character affinities.

A pre-trained model with the English language was chosen for various reasons, including implementation speed and avoiding the need to train a model.

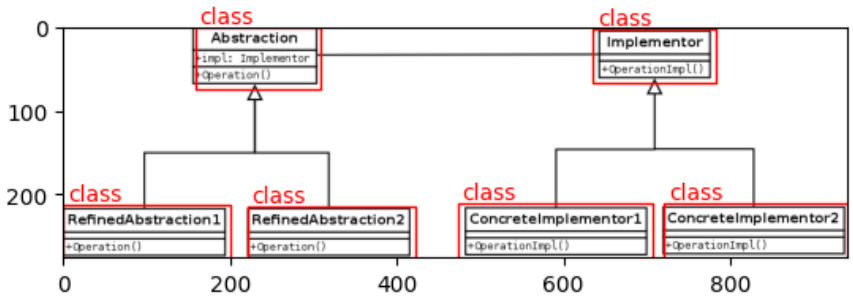

Fig. 7.1 Class detected

7.2 Pre processing

A fundamental part is the type of pre-processing that is performed, and in this case, the used pipeline comprises several stages.

1. Cleanup phase: In this phase, all characters that are not readable and not ASCII, punctuation, unnecessary spaces, and emojis are removed.

2. Separation phase: This phase is specific to the task we are trying to solve, which is the analysis of UML classes. According to software engineering principles, class or method names are assigned following camel case or snake case. In these cases, we want to separate the words to make them distinct.

3. Auto-correction phase: All extracted words are corrected using an automatic corrector to prevent issues like mis-detection in the extraction phase or mis-separation in the separation phase. To achieve this, a pre-trained model called gensim was used, primarily trained on Wikipedia texts, which tend to be more technical and informative.

4. Stemming: By using the Porter Stemming algorithm, an attempt is made to reduce words to their base forms or roots (stems). This process is useful for text normalization and reducing word variations to their primary form.

After this phase, all pre-processed words are ready for the subsequent macro phase. After several tests, it has been found that the error up to this point is negligible. The resulting pipeline is then shown , in fig. 7.1 the classes are segmented , from fig. 7.2 the textual content is extracted , then it is multi-step procesed and you have the results shown in fig. 7.3.

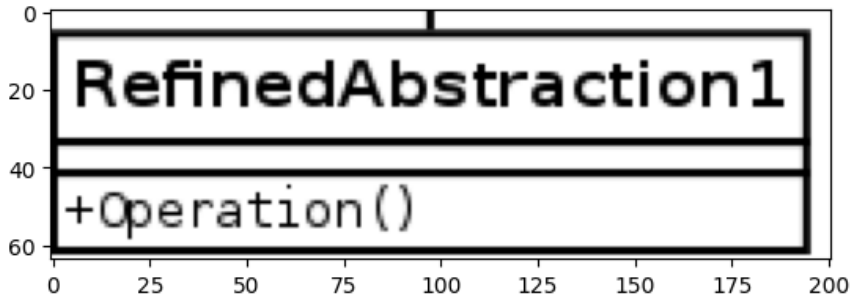

Fig. 7.2 Class segmented

```
['abstraction', 'operation', 'Refined']

[ 'abstract ' , 'oper ' , 'refin ' ]
```

Fig. 7.3 Text extracted

7.3 Semantic processing

To evaluate the semantic correlation between words, there are various techniques and metrics that can be utilized. These techniques can be applied to both word embedding models like Word2Vec, GloVe, and FastText, as well as transformer-based language models like BERT, GPT-3, and others. Before implementing one of these methods, the pros and cons of the two approaches were carefully considered.

Word Embedding (Word2Vec, GloVe, FastText):

pros:

1. Computational Efficiency: Word embedding models tend to be more computationally efficient compared to Transformer models. This makes them more suitable for applications with limited resources.

2. Ease of Use: They are easy to implement and use. They can be trained on relatively small text corpora and provide vector representations of words that can be readily used in NLP applications.

3. Interpretability: Vector representations obtained from word embedding models are often more interpretable, as each dimension of the vector can represent a specific semantic or syntactic feature.

cons:

1. Limitations in Long Sequences: Word embedding models treat words as independent units and may not necessarily capture long-term dependencies between words in a sequence. This can be a limitation for tasks requiring a deep understanding of word relationships, such as machine translation.

2. Limited Flexibility: Word embedding models are primarily designed for representing individual words and do not directly handle the processing of variable-length text sequences.

Transformers (es. BERT, GPT-3):

pros:

1. Text Sequence Processing: Transformer models are designed to handle variable-length text sequences and are particularly effective at capturing complex relationships between words within a sentence or document.

2. Contextual Representations: Transformer models capture the surrounding context of each word, allowing them to create richer and dynamic word representations.

3. State-of-the-Art in Many NLP Tasks: Transformer models like BERT and GPT-3 have demonstrated exceptional performance across a wide range of NLP tasks, including machine translation, text classification, question answering, text generation, and more.

cons:

1. High Computational Complexity: Transformer models are known to require significant computational power and large amounts of training data. This can make them less accessible for developers with limited resources.

2. Lack of Interpretability: Vector representations generated by Transformer models are complex and challenging to interpret. It is not immediately clear what each dimension of the vector represents.

3. Model Size: Successful Transformer models like GPT-3 can be massive in terms of parameters, making it difficult to deploy on devices with limited resources.

Given these pros and cons, the implementation with LAMA2 was attempted, which is the second generation of Meta's open-source large-scale language model. This model has been made freely available for research and commercial use. Unfortunately, the required hardware was too expensive, despite its excellent performance. Therefore, a lighter solution was chosen, namely Word2Vec, which belongs to the word embedding family.

Word2Vec is a Natural Language Processing (NLP) technique published in 2013. The word2vec algorithm uses a neural network model to learn associations between words from a large corpus of text. Once trained, this model can detect synonymous words or suggest additional words for a partial sentence. As the name suggests, word2vec represents each distinct word with a specific list of numbers called a vector. These vectors are carefully chosen to capture the semantic and syntactic qualities of words. In this way, a simple mathematical function (cosine similarity) can indicate the level of semantic similarity between the words represented by those vectors.

The value of cosine similarity between all the words in the UML is a useful metric during training to assign the UML label to the model because it helps determine if the context is coherent. However, it is not used for training due to the computational power required. Evaluating each image would take more than a few minutes, which is unfeasible given such a large dataset, both during training and deployment.

In the image, the semantic distance of an entire UML diagram is represented using Euclidean distance, allowing it to be plotted in fig. 7.4.

7.4 Detection of wrong method inside class

A very interesting implementable application would have been the detection of methods with incorrect names using outlier detection algorithms like the isolation forest. This could have been done with the help of the distance found by the transformer model or Word2Vec. However, the problem in this case lies with the dataset used. Since it was scraped from GitHub and lacks a predominant theme, outlier detection wouldn't be effective.

In a corporate environment, on the other hand, with access to older and validated projects, it would have been possible to identify a more commonly used programming language and train the detection model with this data.

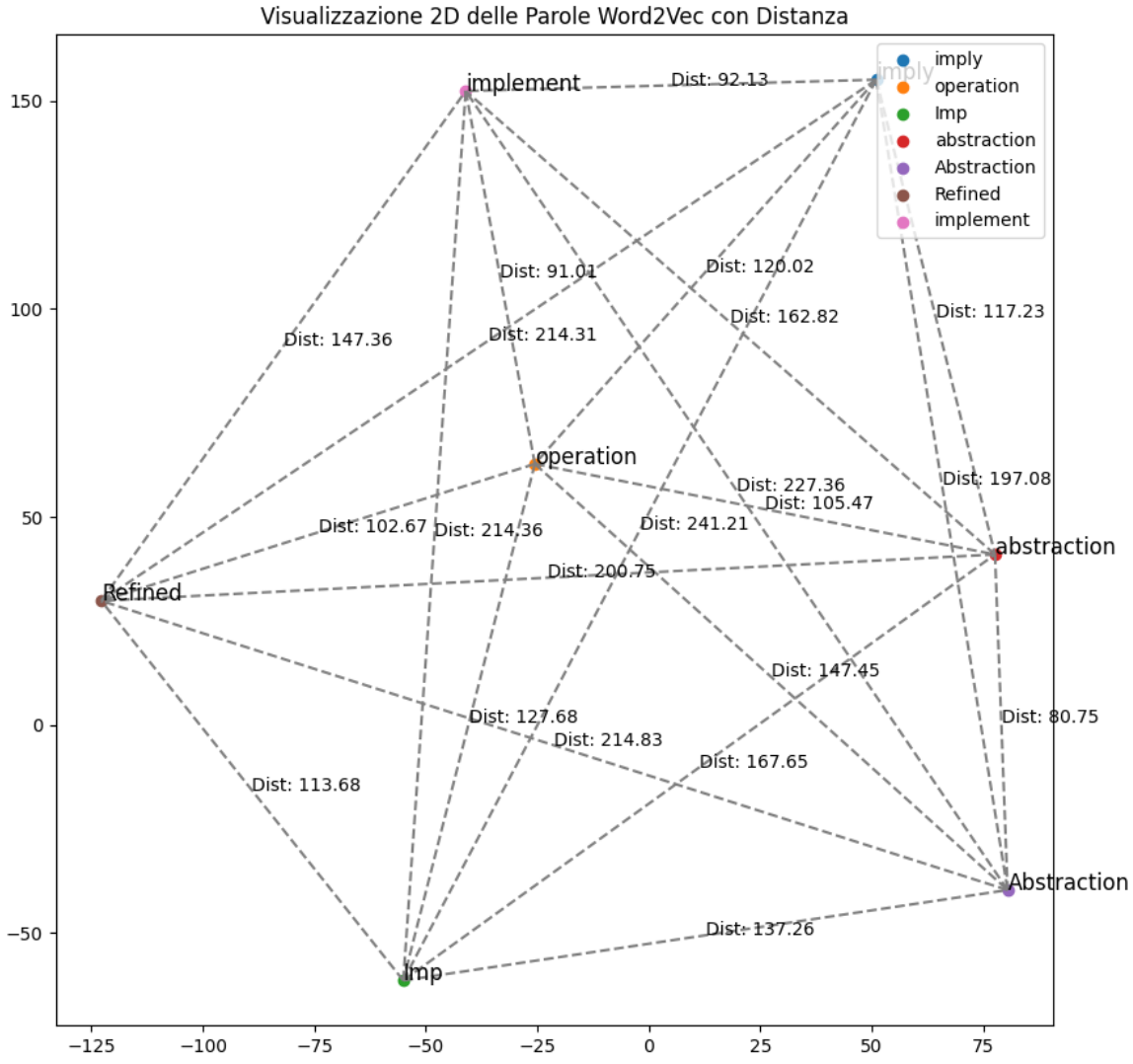

Fig. 7.4 Text distance

Chapter 8

Second model: quality classifier

Intro

The second model that has been trained is a classifier, and it will output the class to which the schema belongs, according to 5 labels ranging from 0 to 5, based on the quality of the schema. Training is performed on the data that has been extracted from the first model, including all metrics derived from various components of the schemas. To recap, the image initially segmented by the first model, Faster R-CNN, is then processed and filtered, and all metrics and features are extracted. These features are then fed into the classifier, which will provide the class of membership as output. Various classic machine learning classifiers were tested to select the best one based on performance. After an initial training phase, the model is usable in the application following image processing as described.

8.1 Dataset

The dataset used results from the processing of all the images from the first model that are associated with quality labels. These labels were obtained from Paper1, where six experts in the field of software modeling manually assigned scores ranging from 0 to 5 based on quality. The dataset was divided into 80% for training, 10% for validation, and 10% for testing, following standard practice, and comprises a total of 663 images.

The metrics used for training are those described in the extraction process by the first classifier, totaling 21 metrics. These metrics encompass aspects related to arrows, crosses, classes, and the image as a whole.

Textual features were excluded due to the following issue: since the dataset was scraped from GitHub, it did not have a specific theme but rather exhibited a wide diversity, ranging

from UML diagrams for financial management software to UML diagrams for pizzeria software. Consequently, it was challenging to identify a specific language or context, rendering the metrics not directly linked to the quality of the schema. This situation differs when creating a dataset with a specific theme, such as in a corporate environment, where more or less significant patterns related to quality can emerge.

In some classifiers, when specified there will be multiple results. This is because, for the feature extraction and dataset creation, two different pipelines were employed. One pipeline utilizes the Faster R-CNN for arrow identification, while the other pipeline solely relies on line identification algorithms. Therefore, concurrently with the search for the best classifier, efforts are made to identify the most reliable and robust feature extractor for the final implementation.

Here are the **arrows metrics** used for training:

- Number of lines
- Average line length
- Line length variation
- Longest line
- Shortest line
- Line bends
- Line angles
- Line orthogonality

Here are the **cross metrics** used for training:

- Number of line crossings
- Crossing angles

Here are the **class metrics** used for training:

- Rectangle coverage
- Average Aspect ratio
- Rectangle size variation
- Number of class

- Rectangle size
- Rectangle proximity
- Rectangle distribution

Here are the **overall image metrics** used for training:

- Image Histogram
- Image aspect ratio
- Image contrast
- Image blur

8.2 Type of classifier

8.2.1 KNN

The k-Nearest Neighbors (KNN) classifier is a non-parametric and instance-based machine learning algorithm. Its fundamental mechanism is predicated on the concept of proximity and similarity within the feature space.

In essence, when presented with a new data point for classification, the KNN algorithm identifies the k nearest neighbors to this point from the training dataset. Neighbors are determined using a distance metric, in this case the Euclidean distance.

The choice of the parameter k, representing the number of neighbors to consider, significantly impacts the algorithm's performance. For this reason, hyperparameter tuning was performed by trying all k values from 1 to 20 and selecting the best classifier. In the case of this classifier, the best value for k is 16, as can be seen from the graphs in Figure 8.1.

KNN directly utilizes the training data at prediction time, making it particularly useful for situations where the underlying data distribution is not well-defined Following the identification of the k nearest neighbors, the algorithm assigns the class label to the new data point based on majority voting among its neighbors. In a classification context, this involves selecting the class label that is most frequently present among the k neighbors. The performance metrics are as follows, and it should be noted that the performance metrics are weighted based on the class:

With pipeline R-CNN:

Accuracy = 0.48

Recall = 0.44

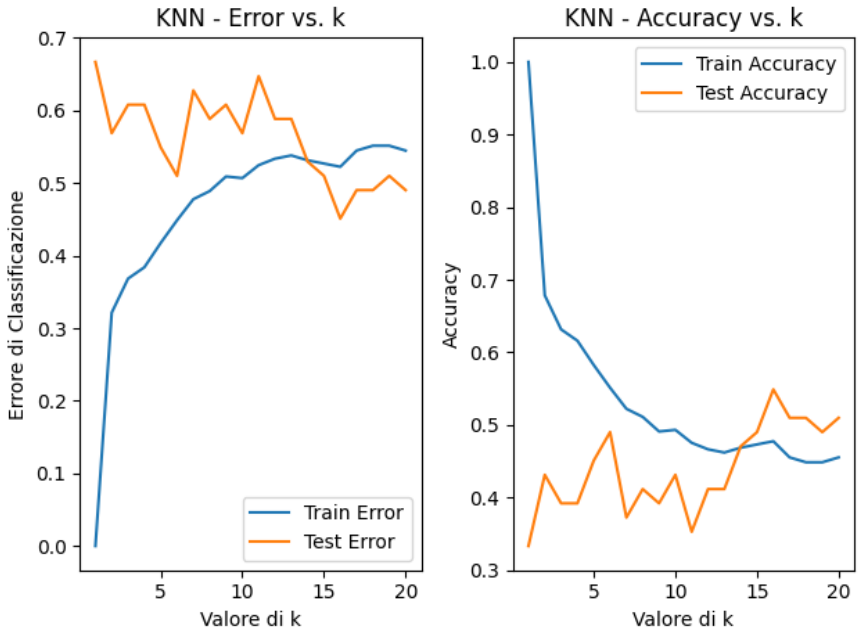

Fig. 8.1 Plot to choose the correct K in KNN

F1 = 0.42

Deviation = 0.84

Another very important metric that has been taken into consideration is the average deviation between predicted classes and true classes on a scale from 0 to 5. This is done to assess how much the classifier deviates from the correct classification.

This value indicates that when the classifier makes an error, on average, it deviates by 1 class. Therefore, it doesn't confuse between good and bad patterns but rather makes errors in the subtleties of goodness.

Here is the confusion matrix fig 8.2.

With pipeline line detection :

Accuracy = 0.30

Recall = 0.35

F1 = 0.31

Deviation = 1.02

8.2.2 Random Forest

The Random Forest classifier is an ensemble machine learning technique that combines the predictive power of multiple decision trees to make accurate predictions. It operates through the following key steps:

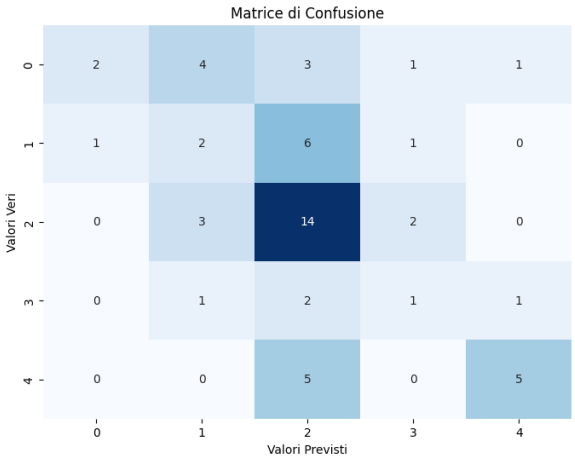

Fig. 8.2 KNN confusion matrix

1. Ensemble of Decision Trees: Random Forest creates a collection of decision trees during the training phase. Each tree is constructed using a different random subset of the training data, a process known as bootstrapping.

2. Voting or Averaging: When it comes to making predictions, each tree in the ensemble provides its own output. For classification tasks, these outputs represent class labels.

3. Random Forest then combines these outputs through a majority vote (in classification) to arrive at a final prediction.

The classifier have the hyper parameter `n_estimators` and determines the number of decision trees that will be created in the ensemble . the parameter has a significant impact on the performance and behavior of a Random Forest model.

The parameter has a significant impact on the performance and behavior of a Random Forest model. For this reason, hyper parameter tuning was performed by trying various combinations and selecting the best classifier, as shown in Figure 8.3. In the case of this classifier, the best value is 150. The overall performance of this classifier is as follows, and it should be noted that the performance metrics are weighted based on the class:

Wth pipeline R-CNN:
Accuracy = 0.45
Recall = 0.43
F1 = 0.42
Deviation = 0.80

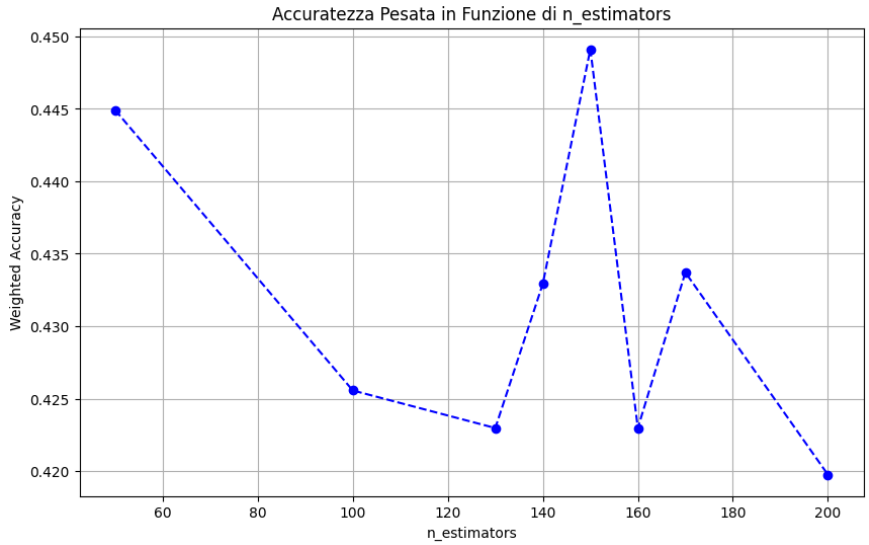

Fig. 8.3 Hyper parameter tuning for Random Forest

This value indicates that when the classifier makes an error, on average, it deviates by 1 class. Therefore, it doesn't confuse between good and bad patterns but rather makes errors in the subtleties of goodness. Here is the confusion matrix fig 8.4.

With pipeline line detection :

Accuracy = 0.55

Recall = 0.53

F1 = 0.53

Deviation = 0.59

Figure 8.5 show the selection for the second pipeline of hyperparameter n_estimators and in the case of this classifier, the best value is 100.

8.2.3 Decision Tree

The Decision Tree classifier is a machine learning algorithm used for classification. the classifier divides the dataset into increasingly homogeneous subsets by applying decision rules based on input features. These rules are used to make predictions or classifications for new data. So operates by recursively partitioning the dataset into subsets based on the values of input features. Here's a concise overview of how it works:

- At the top of the tree is the root node, representing the entire dataset. It selects the feature that best separates the data into distinct subsets based on minimize impurity.

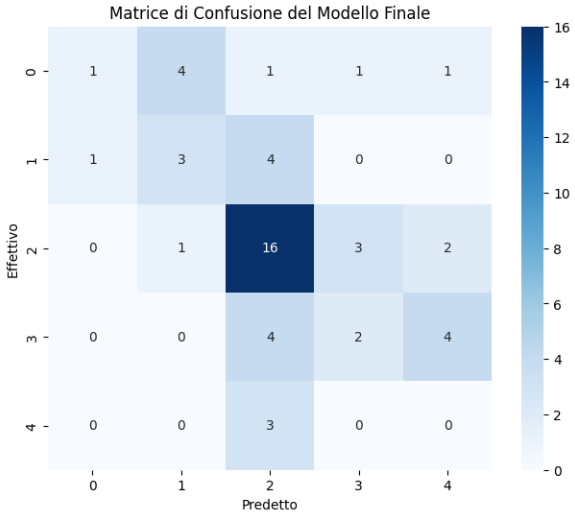

Fig. 8.4 Confusion matrix Random Forest

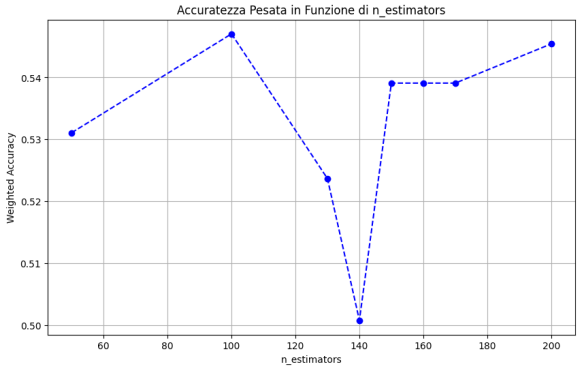

Fig. 8.5 Random Forest of pipeline 2

- Internal Nodes: As the tree grows, internal nodes represent feature-value combinations that split the data into subgroups. The splitting continues until a stopping criterion is met, predefined tree depth or the purity of the subsets.
- Leaf Nodes: The terminal nodes of the tree, known as leaf nodes, contain class labels .
- Decision Rules: The path from the root node to a leaf node represents a series of decision rules. When a new data point is to be classified or predicted, it follows these rules by traversing the tree from the root to a leaf, leading to a final classification.

The classifier has various hyper parameters which has a significant impact on the performance and behavior of the Decision Tree. For this reason, a grid search was performed to optimize the following hyper parameters:

- max_depth: This hyperparameter determines the maximum depth to which the decision tree is allowed to grow during training.
- min_samples_split: This hyperparameter represents the minimum number of samples required to split a node into child nodes during the construction of the decision tree.
- min_samples_leaf: This hyperparameter sets the minimum number of samples required to form a leaf node at the end of a branch. It determines the smallest size of a node that can make predictions.
- max_features: It determines the maximum number of features that the decision tree considers when making a split.

The overall performance of this classifier is as follows, and it should be noted that the performance metrics are weighted based on the class:

With pipeline R-CNN:

Accuracy = 0.45

Recall = 0.43

F1 = 0.42

Deviation = 1.0

This value indicates that when the classifier makes an error, on average, it deviates by 1 class. Therefore, it doesn't confuse between good and bad patterns but rather makes errors in the subtleties of goodness. Here is the confusion matrix fig 8.6.

With pipeline line detection :

Accuracy = 0.47

Recall = 0.52

F1 = 0.48

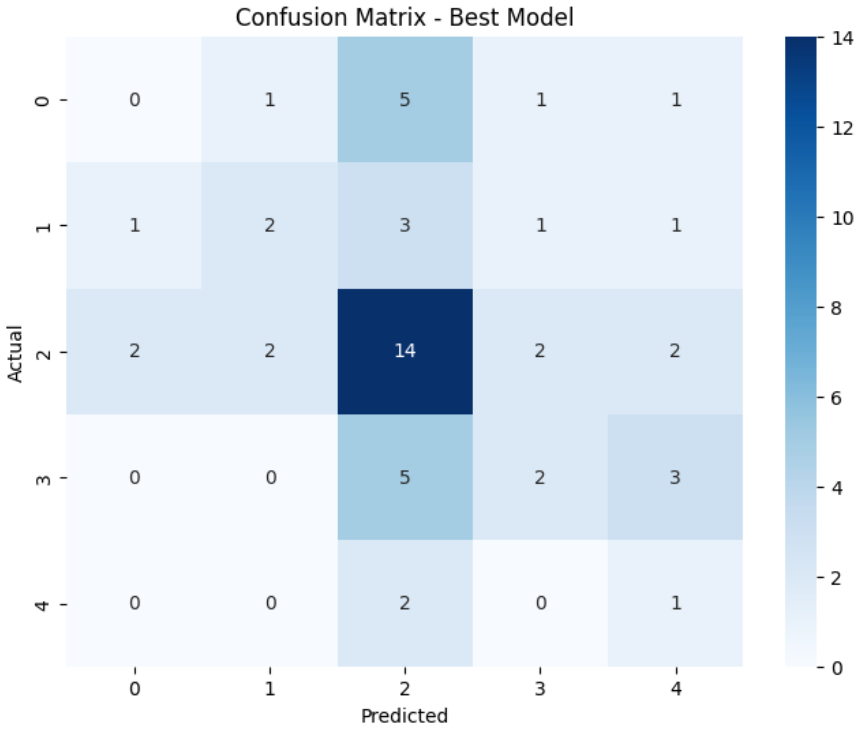

Fig. 8.6 Confusion Matrix for decision tree classifier

8.2.4 Naive Bayes

The Gaussian Naive Bayes classifier is a probabilistic machine learning algorithm used primarily for classification tasks. It operates based on the principles of Bayes' theorem and a simplifying assumption of feature independence, which is why it is called "Naive ". Despite its simplistic assumptions, Naive Bayes can be surprisingly effective for a wide range of classification tasks, especially when dealing with text or high-dimensional data. Here's a concise explanation of how it works:

- Probability and Bayes' Theorem: At its core, the Naive Bayes classifier calculates the probability of a data point belonging to a particular class. It uses Bayes' theorem, which relates conditional probabilities. Specifically, it calculates the probability of a class given some observed features.

- Feature Independence Assumption: The "Naive" aspect of Naive Bayes is the assumption of feature independence. It assumes that the features (variables) used to describe the data are all independent of each other.

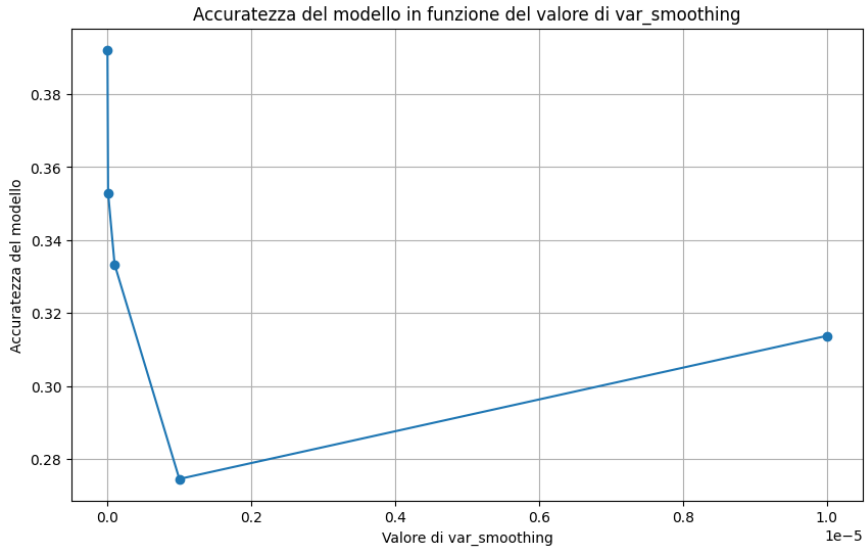

Fig. 8.7 Hyper parameter tuning for Naive Bayes classifier

- Calculating Class Probabilities: To classify a new data point, the algorithm calculates the probability of it belonging to each possible class. It does this by multiplying the probabilities of each feature occurring in that class, based on the training data.

In the provided code, `var_smoothing` is a hyper parameter for the Gaussian Naive Bayes classifier. It determines the amount of smoothing applied to the variances of the features when modeling the Gaussian distributions for each class. and have a large impact depending on the chosen value , this is why is performed an hyper parameter tuning on a range of value to choose the best classifier. In fig. 8.7 is shown the various test and their performance.

The overall performance of this classifier is as follows, and it should be noted that the performance metrics are weighted based on the class:

With pipeline R-CNN:

Accuracy = 0.38

Recall = 0.38

F1 = 0.41

Deviation = 0.92

This value indicates that when the classifier makes an error, on average, it deviates by 1 class. Therefore, it doesn't confuse between good and bad patterns but rather makes errors in the subtleties of goodness. Here is the confusion matrix fig 8.8.

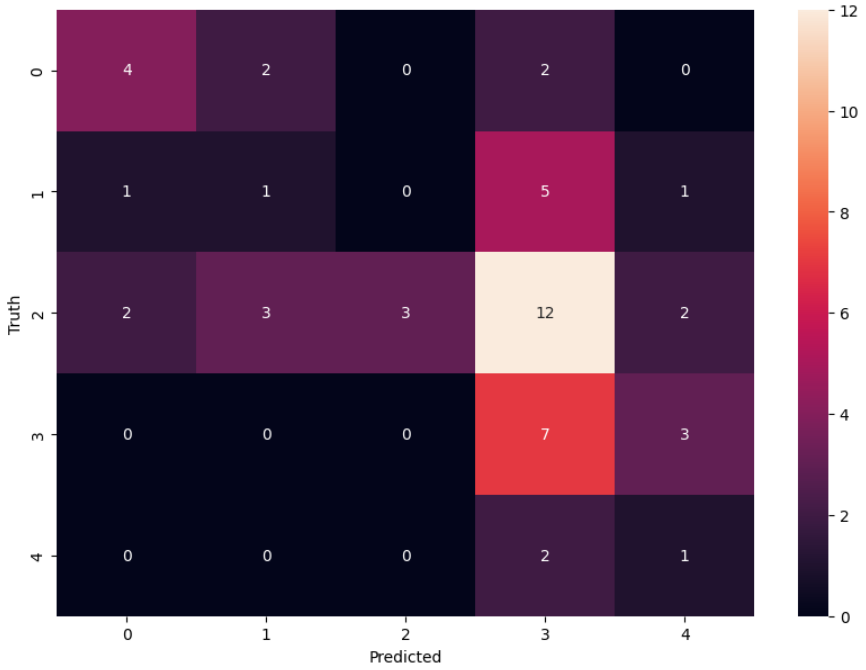

Fig. 8.8 Naive Bayes confusion matrix

8.2.5 Neural Network classifier

The Deep Neural Network (DNN) classifier is a machine learning model that mimics the structure and function of the human brain's neural networks. It consists of multiple layers of interconnected artificial neurons. Each layer processes and transforms the input data to make predictions or classifications. Here's a brief explanation of how it works:

- Neural Network Layers: A DNN classifier typically comprises an input layer, one or more hidden layers, and an output layer. The input layer receives the features of the data, while the hidden layers perform complex transformations, and the output layer produces the final predictions or classifications.
- Activation Functions: Each neuron within a layer applies an activation function to its input.
- Weights and Connections: The connections between neurons are associated with weights. During training, these weights are adjusted to minimize the difference between the predicted outputs and the actual target values. This process is typically achieved through backpropagation and gradient descent algorithms.
- Loss Function: The loss function quantifies the difference between the predicted and actual values.

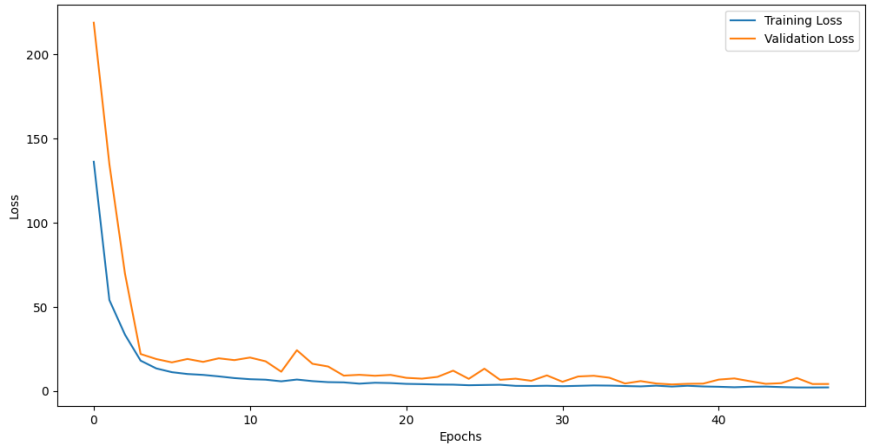

Fig. 8.9 DNN classifier training

- Early Stopping: Early stopping is a technique used during training to prevent overfitting. It monitors a specified metric (e.g., validation loss) and stops training if that metric doesn't improve anymore.

In the case of the created DNN, a framework called Keras, built on the foundations of TensorFlow, was used. The constructed network is simple, consisting of only two layers of hidden neurons with 16 and 8 neurons, and a final layer with 5 neurons, corresponding to the number of classes to classify.

The architecture was intentionally kept simple to avoid underfitting due to the limited training data. Fine-tuning was performed, with the chosen number of neurons being the best configuration.

Early stopping halted the training at the 48th epoch. Figure 8.9 illustrates the loss trend during training.

The overall performance of this classifier is as follows,are not so god as other classifier because generally dnn needs more training ,and it should be noted that the performance metrics are weighted based on the class:

With pipeline R-CNN:

Accuracy = 0.22

Deviation = 1.22

This value indicates that when the classifier makes an error, on average, it deviates by 1 class. Therefore, it doesn't confuse between good and bad patterns but rather makes errors in the subtleties of goodness. Here is the confusion matrix fig 8.10.

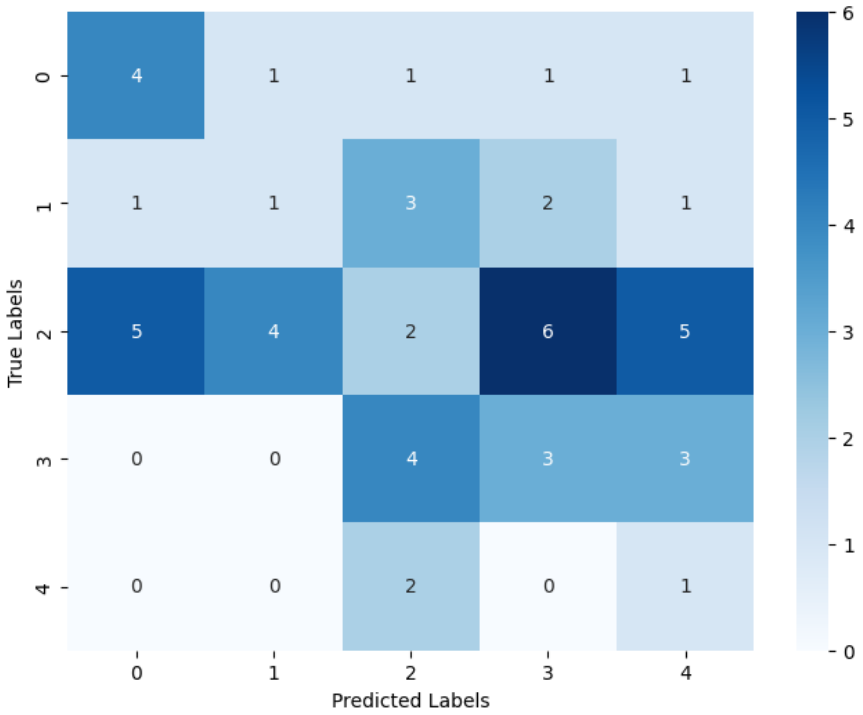

Fig. 8.10 DNN confusion matrix

8.2.6 SMOTE

As an alternative to balancing class weights during training, another approach used for classifier training is SMOTE, which stands for "Synthetic Minority Over-sampling Technique." SMOTE is a technique employed in data mining to address the issue of class imbalance, as seen in our case in Figure 9.3. It generates synthetic data in the minority class to balance the dataset and improve the performance of machine learning models. The result of this class balancing is depicted in Figure 9.4. In brief, here is how SMOTE operates:

1. Identification of Minority Instances: Initially, SMOTE identifies instances belonging to the minority class in the dataset.

2. Selection of Nearby Instances: For each selected minority instance, SMOTE randomly chooses some of its neighboring minority instances.

3. Creation of Synthetic Instances: For each selected minority instance and its nearby instances, SMOTE generates new synthetic instances. For each feature, SMOTE calculates a weighted average between the values of the minority instance and the nearby instances, and then generates a synthetic instance based on these averages.

4. Addition of Synthetic Instances to the dataset.

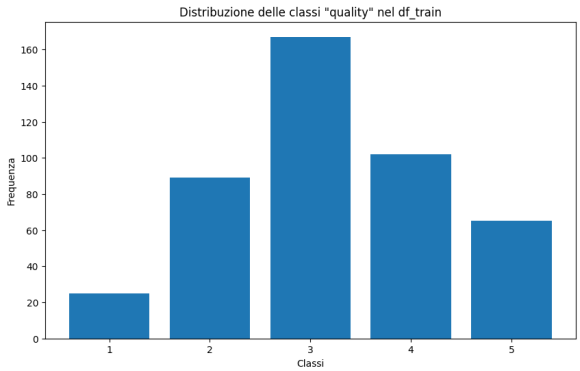

Fig. 8.11 Dataset before SMOTE

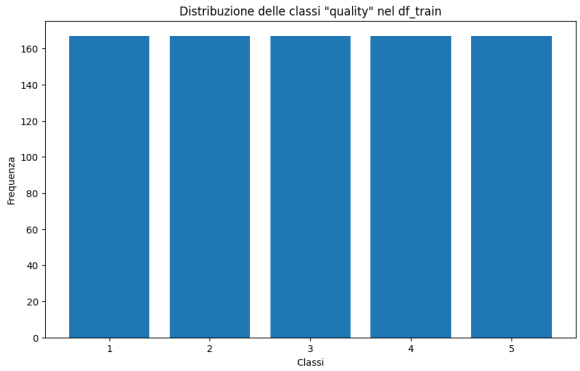

Fig. 8.12 Dataset after SMOTE

5. Model Training.

Two classifiers were trained using this technique: K-nearest neighbors (KNN) and Random Forest. Both exhibited an improvement in performance, but Random Forest claimed the top position for performance. The classifier was then trained using the same pipeline, subsequently undergoing hyperparameter tuning with a greed search.

Several quantities of synthetically generated data have been tested to find the appropriate trade-off between synthetic and real data. The issue with synthetic data is that if one exaggerates, the model will go into overfitting or will not reflect the actual performance of the trained model, as it will be unable to generalize.

With pipeline Faster RCNN :
Accuracy = 0.53
Recall = 0.43
F1 = 0.47
Deviation = 0.84
With pipeline line detection :
Accuracy = 0.65
Recall = 0.59
F1 = 0.61
Deviation = 0.73

In this case, the best performance is achieved with the pipeline that identifies arrows without utilizing the Faster R-CNN. This is because it can more finely identify lines in the case of complex and confusing images, thus discerning the differences more finely among classes belonging to lower categories. In fact, there is an increase of almost 10% compared to the first pipeline.

8.2.7 Ensemble Methods with three classifier and SMOTE

Ensemble is a machine learning technique wherein the results of multiple machine learning models (in this case, classifiers) are combined to enhance overall performance and prediction robustness. This is accomplished through a voting mechanism employed by each classifier to assign a predicted class; this voting method is known as 'soft voting.' In this scenario, rather than outputting a single class, classifiers yield class probabilities. The final outcome is derived from the weighted average of these probabilities.

In the implemented code, three different classifiers are utilized: Random Forest, Gradient Boosting, and Support Vector Machine (SVM). For each classifier, hyperparameter fine-tuning is initially conducted through grid search. The three best-trained classifiers become

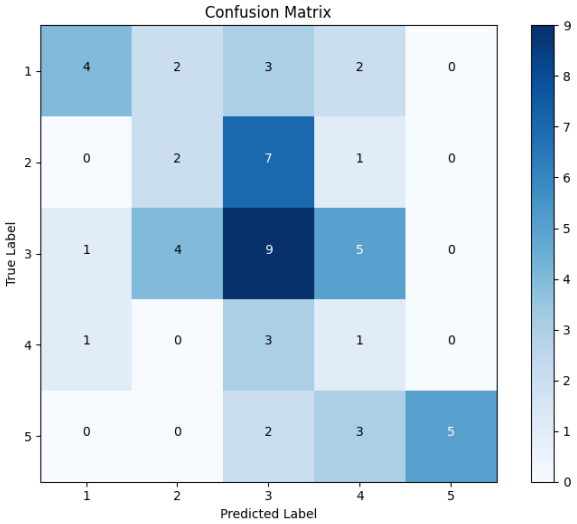

Fig. 8.13 Confusion Matrix esemble method

	precision	recall	f1-score	support
1	0.67	0.36	0.47	11
2	0.25	0.20	0.22	10
3	0.38	0.47	0.42	19
4	0.08	0.20	0.12	5
5	1.00	0.50	0.67	10
accuracy			0.38	55
macro avg	0.47	0.35	0.38	55
weighted avg	0.50	0.38	0.41	55

Fig. 8.14 Report esemble method

part of the voting pool, and they express their votes in terms of probabilities, which are subsequently employed to determine the final vote.

The Synthetic Minority Over-sampling Technique (SMOTE) is also employed to rebalance the classes within the dataset, given its initial class imbalance. The scikit-learn library in Python is used for both the classifiers and the ensemble.

With pipeline Faster RCNN :

Accuracy = 0.50

Recall = 0.38

F1 = 0.41

The confusion matrix is depicted in Figure 8.13, while the report is presented in Figure 8.14.

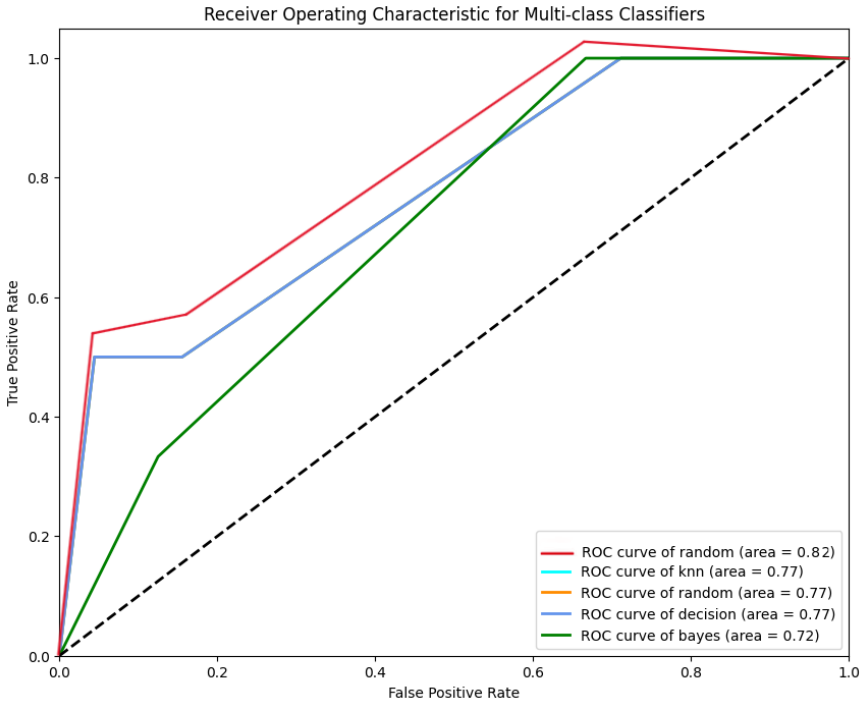

Fig. 8.15 AUC analysis for classifier

8.3 The best classifier

The ROC curve (Receiver Operating Characteristic curve) is an excellent tool for evaluating and comparing classifier performance. The ROC curve allows for visualizing the trade-off between sensitivity (True Positive Rate) and specificity (True Negative Rate) of a classifier at various decision thresholds. A classifier with an ROC curve that closely approaches the upper-left part of the graph is generally considered better, providing a quantitative measure of each classifier's performance. A higher AUC indicates better performance.

In Figure 8.15, the performance graph of various classifiers is displayed, and Random Forest classifier using pipeline 2 with SMOTE has been selected has best one. Below, the ROC curves (fig. 8.16, fig. 8.17, fig. 8.18, fig. 8.19, fig. 8.20) for the quality classes of the Random Forest classifier are shown to provide a more general overview. In figure 8.21 , is possible to see the overall performance with the weighted metric for the chosen classifier , and the confusion matrix in figure8.22.

This classifier is been obtained using the feature engineering , deleting less important feature and adding some other else. All this plots and reports give a precise look of the best classifier obtained , furthermore in the chapter 9 are been analyzed more precisely all the limits of the classifier.

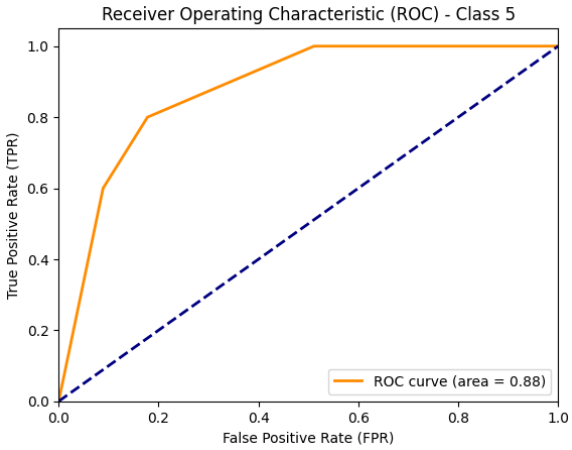

Fig. 8.16 Roc curve class 5

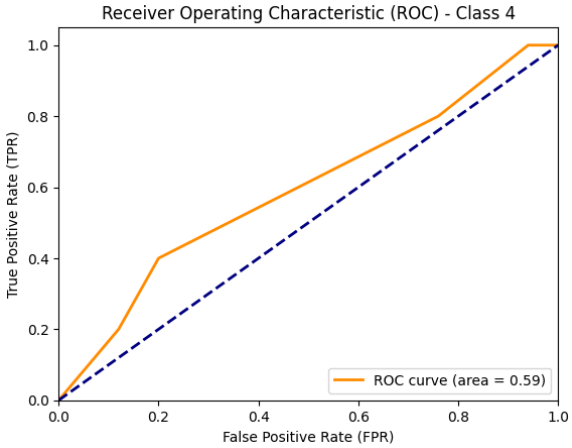

Fig. 8.17 Roc curve class 4

In general, the best results were obtained using the pipeline with direct line identification, which led to more precise outcomes and, consequently, improved accuracy and AUC. Arrows identify a significant portion of the problem and classification, so the impact of enhancing this identification is substantial, even with minor improvements in the final classifier. Furthermore, the SMOTE technique has assisted in increasing and overcoming the challenges posed by the imbalanced dataset.

8.4 Correlation Analysis

After the classification process, an analysis was conducted to evaluate the feature that has the greatest importance compared to the others, showing the ranking of the features based on the absolute value of their correlation with the ground truth layout quality in table 8.1. Features

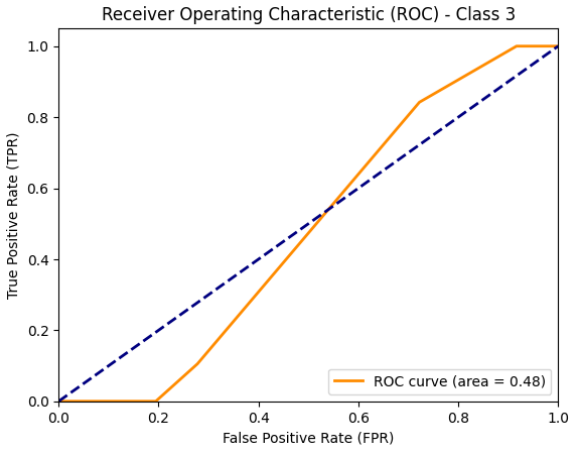

Fig. 8.18 Roc curve class 3

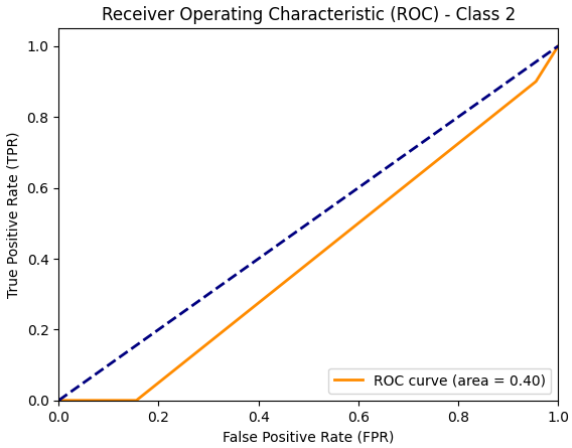

Fig. 8.19 Roc curve class 2

with a correlation score closer to 1 are the ones that have a more significant contribution (positively or negatively) to the layout score of the diagram.

The distributions of the six most important features are shown in figures 8.23, 8.24, 8.25, 8.26,8.27 and 8.28.

As evident from this analysis, the most influential metrics pertain to UML diagram classes, and secondarily, to the arrows in the diagram. High-quality diagrams are characterized by a uniform distribution of classes; they also have a smaller number of diagram elements (specifically, the number of lines and rectangles). In high-quality graphs, the arrows tend to be composed of fewer broken lines. Additionally, it is worth noting that the variance for higher quality categories tends to be smaller than that observed in lower quality categories.

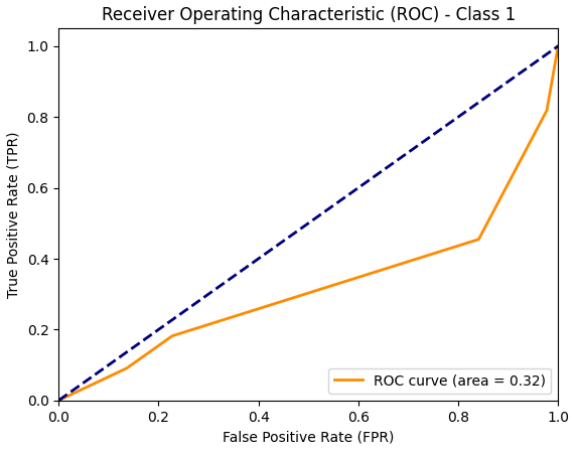

Fig. 8.20 Roc curve class 1

Report di Classificazione del Modello Finale:

	precision	recall	f1-score	support
1	0.80	0.50	0.62	8
2	0.38	0.62	0.48	8
3	0.75	0.68	0.71	22
4	0.71	0.50	0.59	10
5	0.17	0.33	0.22	3
accuracy			0.59	51
macro avg	0.56	0.53	0.52	51
weighted avg	0.66	0.59	0.61	51

Fig. 8.21 Best classifier performance

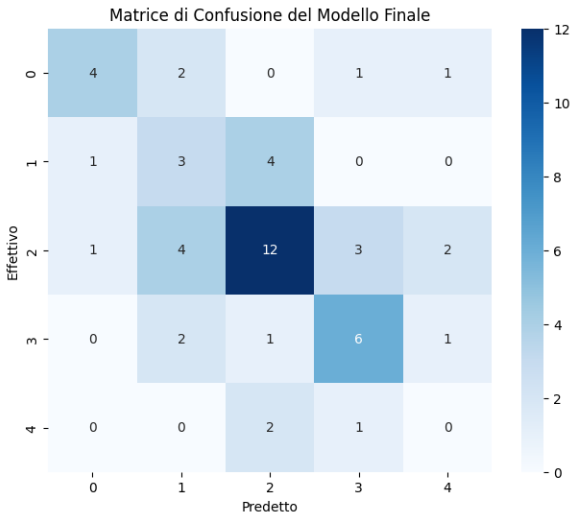

Fig. 8.22 Best classifier confusion matrix

Table 8.1 Feature importance ranking based on absolute correlation coefficient

Rectangle proximity	0.456313
Number of rectangles	0.430160
Line bends	0.403082
Number of lines	0.374926
Longest line	0.368305
Average line length	0.356026
Line length variation	0.347196
Rectangle size	0.328496
Image blur	0.327464
Line orthogonality	0.279491
Image contrast	0.177282
Line angles	0.167674
Shortest line	0.142395
Crossing angles	0.134158
Line crossings	0.125000
Rectangle size variation	0.051013
Image aspect ratio	0.047006
Class aspect ratio	0.046983
Rectangle distribution	0.035965
Rectangle coverage	0.032958

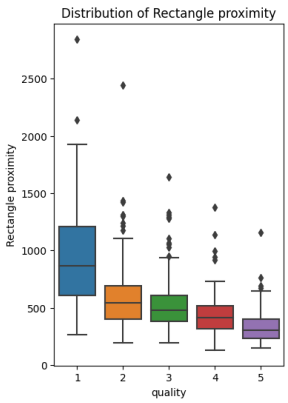

Fig. 8.23 Correlation of rectangle proximity

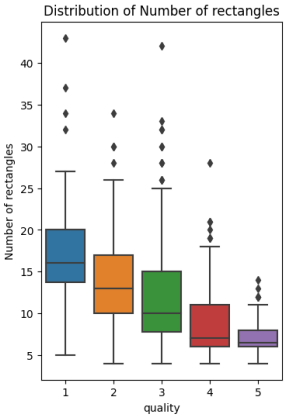

Fig. 8.24 Correlation of #class

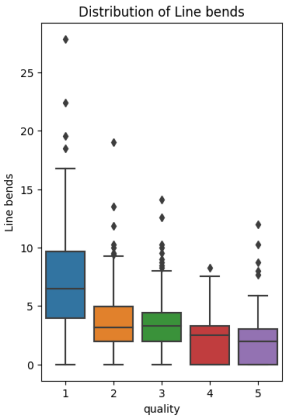

Fig. 8.25 Correlation of line bends

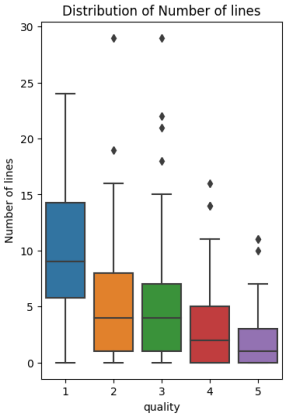

Fig. 8.26 Correlation of #lines

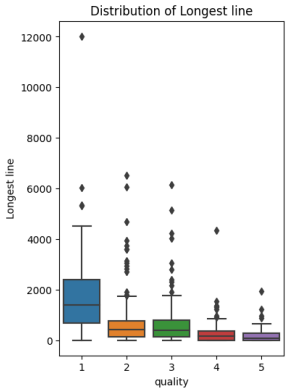

Fig. 8.27 Correlation of longest line

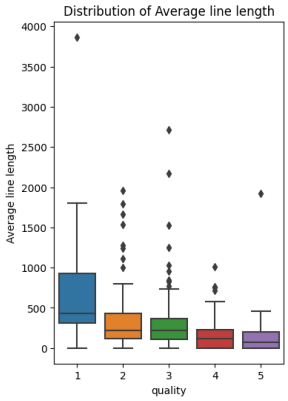

Fig. 8.28 Correlation of avg. line lenght

Chapter 9

Limitations of the application

Intro

This chapter is dedicated to the limitations of various stages within the pipeline of the final model. An attempt has been made to analyze every aspect of each phase to find the best trade-off or a solution, which has proven effective in many cases. Particular attention has been paid to the limitations, which result from various causes that interconnect, leading to an inevitable degradation of overall performance. The limitations of the entire system are, therefore, the sum of the limitations of the various subsequent phases.

9.1 First model limits : segmentation

In this phase, several limitations of the model trained for segmentation become apparent, and these limitations subsequently affect various stages, leading to a degradation of overall performance. Indeed, if the Faster R-CNN model is not optimal, it significantly influences all subsequent calculations. Concerning the segmentation part related to the classes, there are not many issues because it is a relatively straightforward problem, and miss detections are rare, as can also be observed in the segmenter's performance.

The situation becomes different when segmenting lines. In simple cases, there is little room for error. However, in cases where lines intersect or are very close to each other, since the segmenter is trained to create bounding boxes, there may be multiple lines within the same bounded box, most of which will be filtered out and not considered. These situations occur infrequently, but even a human eye would face difficulties in many instances when trying to delineate bounding boxes. An example can be seen in the schema in Figure 9.1. To address this issue, Pipeline 2 was subsequently created, which did not use the Faster

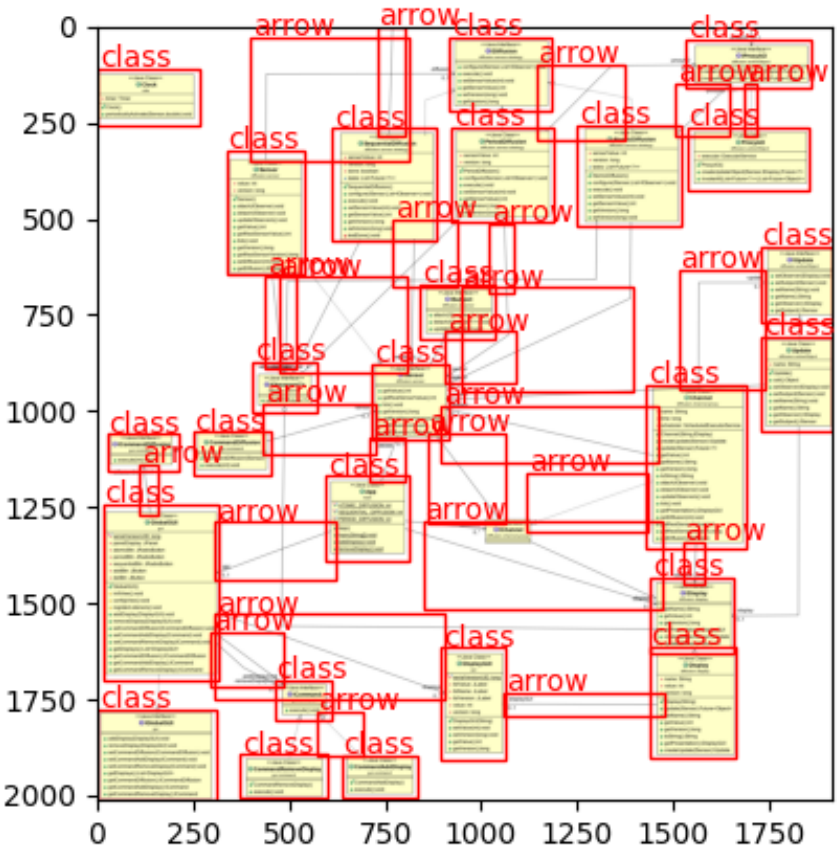

Fig. 9.1 Complex image where some arrow are miss detected

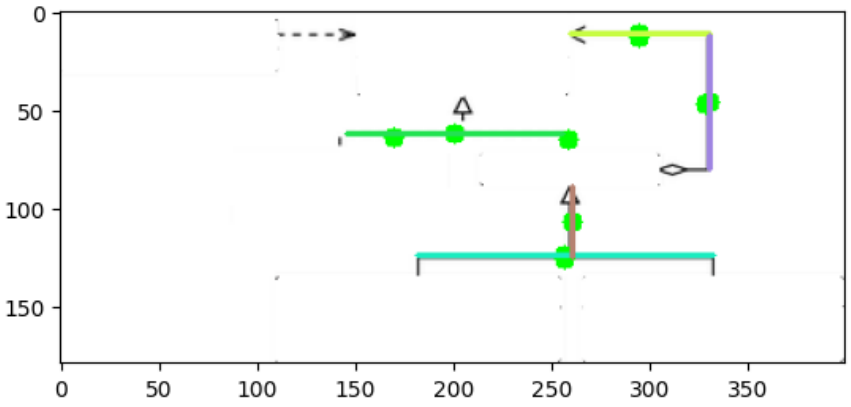

Fig. 9.2 Arrow miss detected from Pipeline2

R-CNN segmentation model but relied on line identification algorithms such as the Hough transform and the Canny algorithm. An increase in performance was noted compared to Pipeline 1. However, in this case as well, some lines are missed due to the nature of the available algorithms and the subsequent filtering of detected lines, necessitated by the low precision of the initial algorithms. In Figure 9.2, it is evident that the highlighted line is not detected by the algorithm. It is necessary to remember that these miss detections also occur due to the trade-off made; to address the miss detection of 100 lines, the miss detection of other lines is introduced. The models have indeed been fine-tuned with hyperparameters.

Efforts have been made to improve this phase to the best extent possible, but the limitations, especially in the case of Faster R-CNN, stem from the limited data available. With more data, it would have been possible to enhance accuracy.

Although the precision of the segmenter is close to 90%, these minor issues have a cascading effect on feature extraction and the final classifier. Even in the reference paper, there are issues with the identification of various classes and arrows, so this is not a new problem.

9.2 Feature extraction limits

The feature extraction phase is highly dependent on the accuracy of the segmenter; the greater the precision, the higher the quality of the extracted features. The features that have been developed are derived from the reference paper1, along with some additional features that were conceived during image processing.

The primary objective of each feature is to highlight both positive and negative aspects of the graph under analysis. There are not many limitations in this phase; it mainly serves to

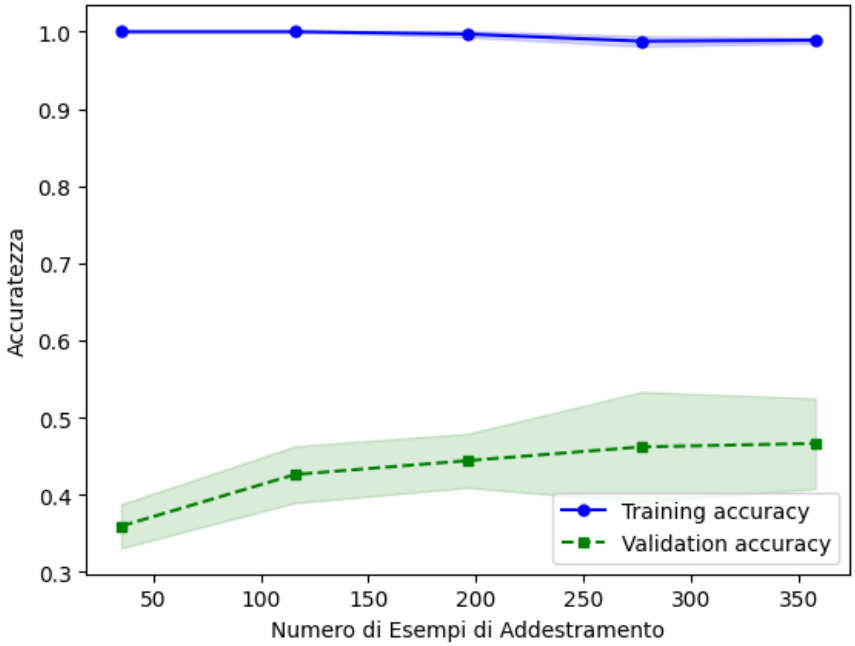

Fig. 9.3 Accuracy plot without SMOTE

amplify the limitations of the segmenter. This is because mathematical formulas that are not subject to random errors are employed in this phase.

9.3 Classifier limits

9.3.1 Amount of data

In this phase, there are significant limitations that significantly impact the overall performance of the entire software. The first issue that arises is that the dataset is too small for the complexity of the problem. In fact, in this case, we have the entire dataset of approximately 650 images, split into 80% for training, 10% for validation, and 10% for testing. Therefore, the training set consists of approximately 500 rows, each composed of 25 features.

The best classifier is the Random Forest, as illustrated in the classifier section, which has been fine-tuned, but the performance is still too low. As observed in Figure 9.3, the model learns well during training without overfitting or underfitting, but the reason for the low accuracy is simply the lack of data. To overcome this issue, SMOTE was applied, which involves creating synthetic samples for the minority classes, i.e., the classes with fewer samples, to balance the dataset, increasing the sample count from about 500 to nearly 1000 samples. As seen in Figure 9.4, the performance has significantly improved. However, if one

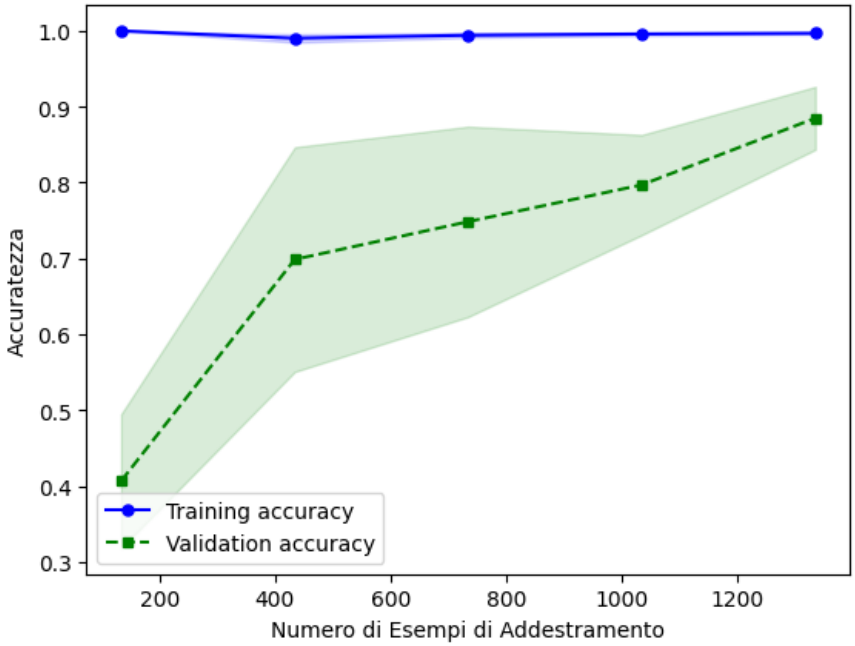

Fig. 9.4 Accuracy plot with SMOTE

goes overboard with the SMOTE technique, creating too many synthetic instances compared to the real dataset, the training graph will look better, as shown in Figure 9.5, but the model will overfit. As a result, the test set will exhibit very low performance because the model won't be able to generalize well. This is because SMOTE may have introduced synthetic data that contains noise or does not accurately represent the distribution of real data. Additionally, SMOTE generates synthetic data, but the original dataset is too small, so there is not enough information to train a model that generalizes effectively.

Therefore, it was decided to use SMOTE in a way that optimally balances the performance of the final classifier, as shown in previous chapters and used in the final model pipelines.

9.3.2 Important feature

All features have been analyzed to eliminate and exclude from the classifier's training those features that have a lower impact compared to others. This feature selection serves various purposes:

1. Model simplification, making the training faster and improving performance on the test dataset.

2. Reducing the risk of overfitting, especially if your initial model had a tendency to be overly complex due to high dimensionality.

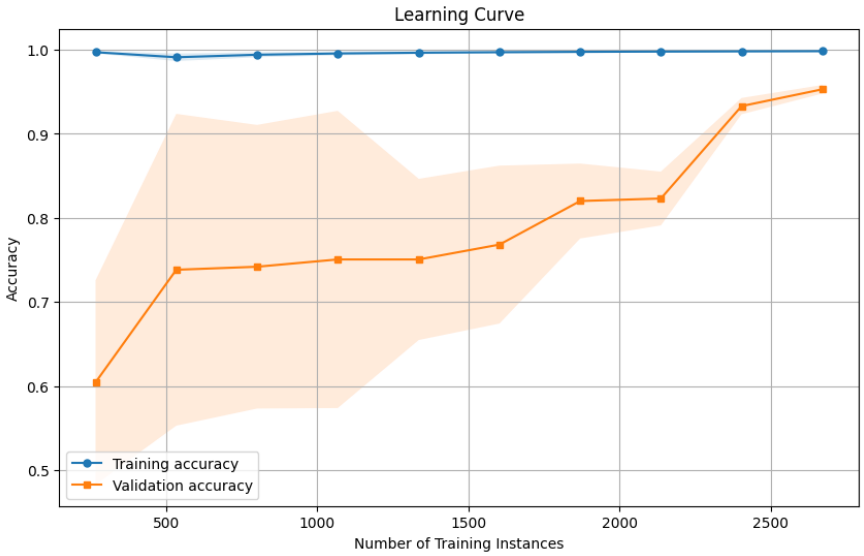

Fig. 9.5 Accuracy plot with extreme SMOTE

3. There is a risk of losing valuable information if not done properly.

In Figure 9.6, all features are displayed in order of importance, from which the last 3 features, which were less informative, have been removed, resulting in an average gain of +2% in overall accuracy for the model.

9.3.3 Miss classified image

In the analysis of the features and the confusion matrix, a discrepancy and an unusual error were observed. In fact, the model tends to make mistakes very easily in classifying from class 3 to class 2, as depicted in Figure 9.7. Therefore, the initial idea was to identify the features that were responsible for this misclassification and eliminate them if they did not impact the identification of other metrics.

To understand the correlations between the features and the misclassified classes, the distributions of your features for rating classes 2 and 3 were examined and overlaid to determine if the distributions are significantly different between these two classes. Additionally, the Kolmogorov-Smirnov test was conducted to compare the distributions. This is a non-parametric test that assesses the difference between two cumulative distributions. A low p-value indicates that the two distributions are significantly different.

Among the 25 feature plots, if the shape of the distribution of a feature is markedly distinct between rating classes 2 and 3, it may suggest that this feature interacts differently with these two classes and aids in distinguishing between them. Conversely, if the distributions of a

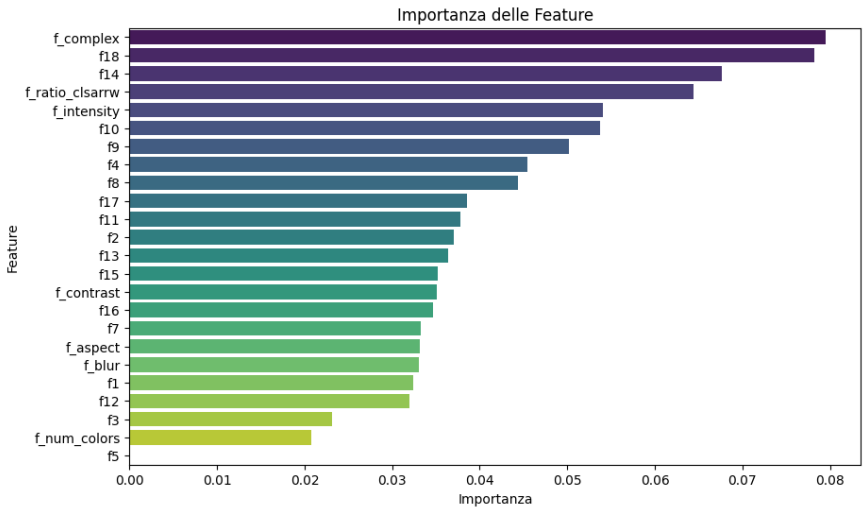

Fig. 9.6 Feature importance

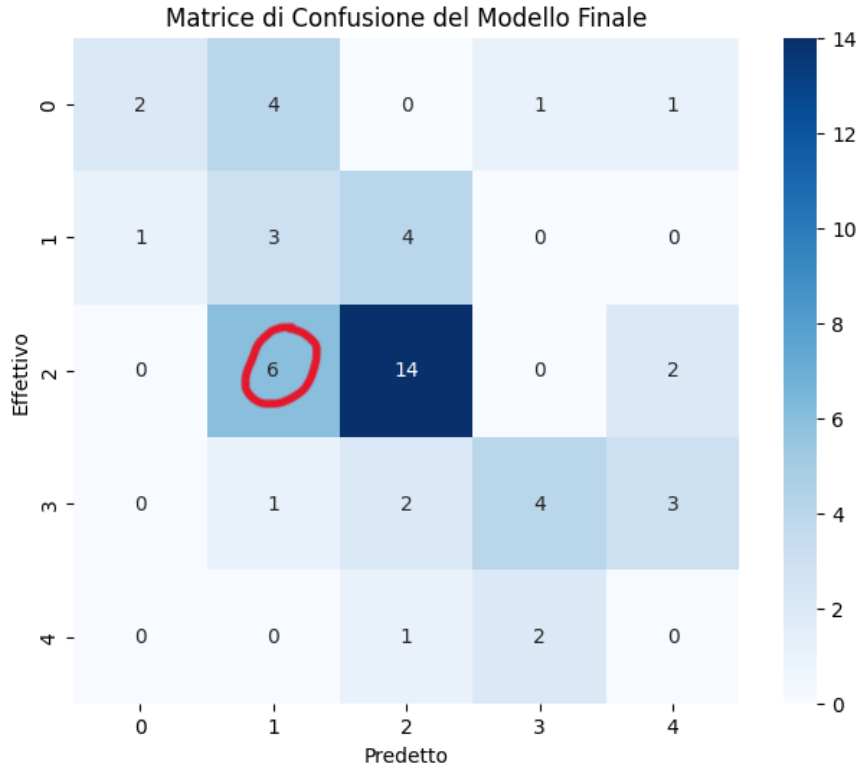

Fig. 9.7 Miss classified schemes

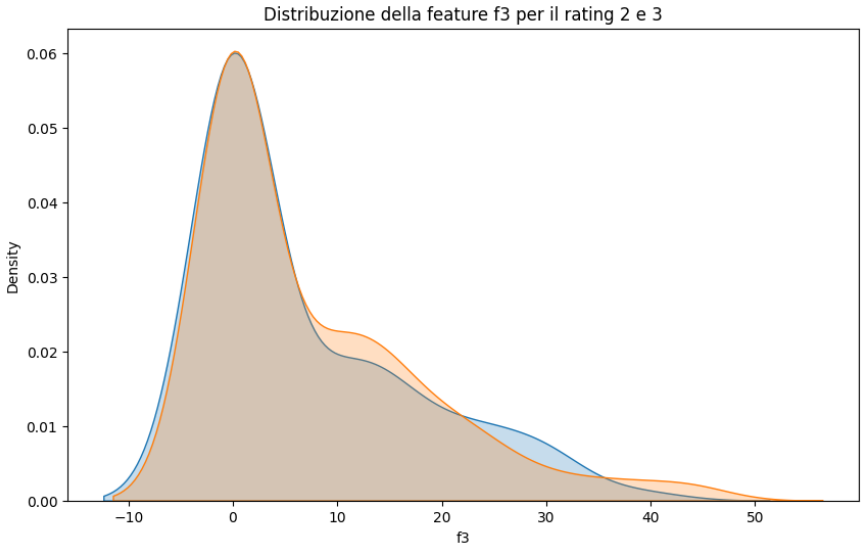

Fig. 9.8 Similar distribution p=1

feature for rating classes 2 and 3 exhibit substantial overlap, it implies that this feature might not be very useful for distinguishing between these two classes.

As shown in Figure 9.8 and Figure 9.9, the classes overlap considerably, while in other cases, such as in Figure 9.10 and Figure 9.11, they are markedly distinct and assist the model in differentiation.

The initial strategy implemented, following a thorough analysis, involved the elimination of all features with similar distributions that currently do not have significant importance in the model, as indicated by the previously examined graph. The model was then retrained, and its performance was evaluated.

Eliminating features without considering their importance could lead to a loss of information from those features, which might have been essential for identifying another class, potentially resulting in a worsened model.

The eliminated classes that are possible to delete are without have significant loss of information are F3 and F_aspect.

The model's performance has slightly worsened, likely due to the fact that there is already limited data, and then we further diminish the amount of extracted information, leading to a degradation in performance. In the confusion matrix as well, no significant improvements are observed. This is because the most important features also have distributions that are quite similar, and by removing the less informative ones, no particular advantages are gained, as depicted in Figure 9.12.

Another strategy could involve creating new features that capture specific information that your current model might not discern. Interactions between features and transformations

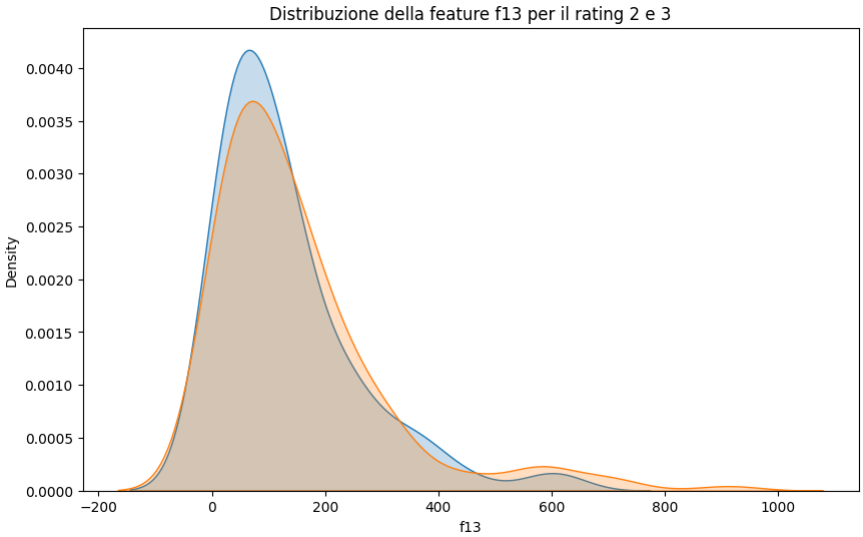

Fig. 9.9 Similar distribution p=0.55

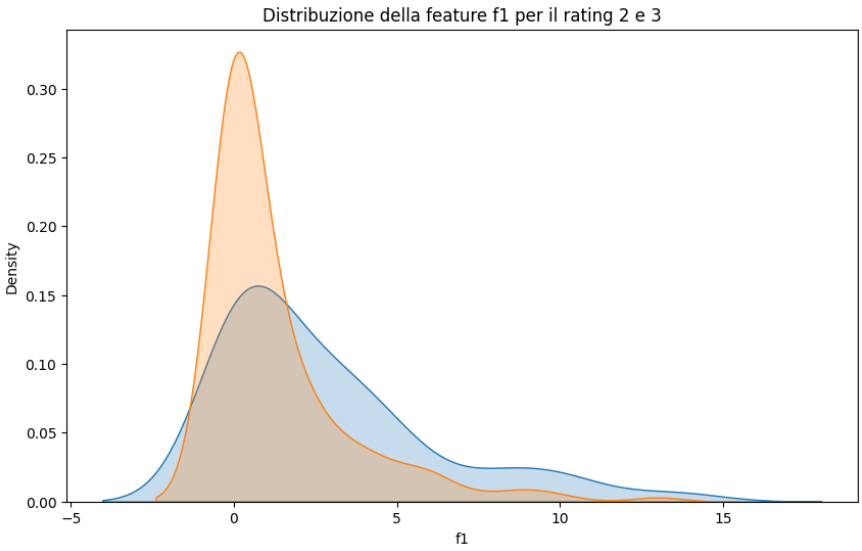

Fig. 9.10 Different distribution p=0.0

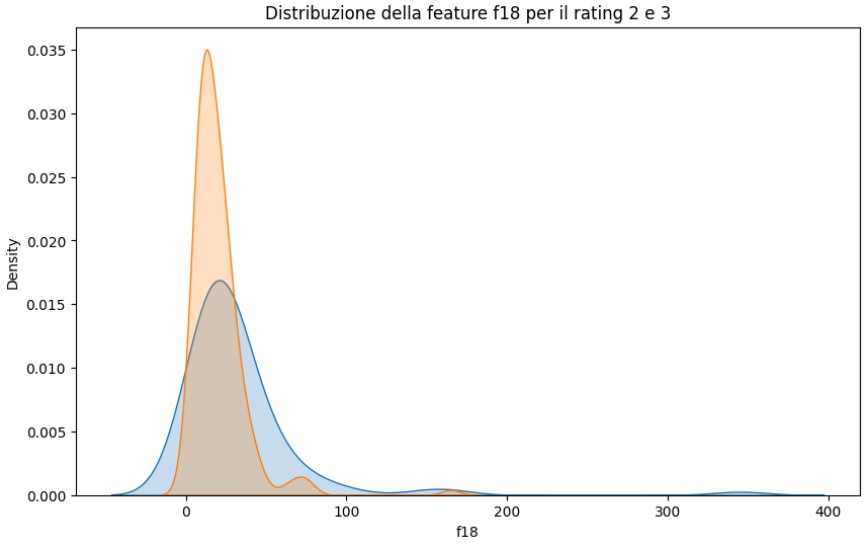

Fig. 9.11 Different distribution p=0.02

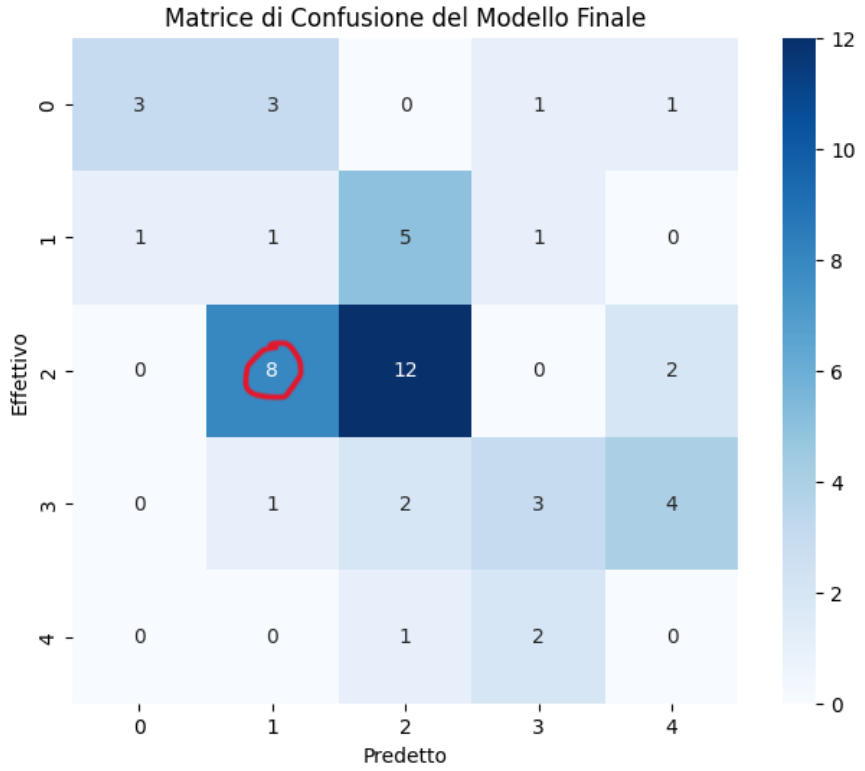

Fig. 9.12 Confusion matrix after feature selecting

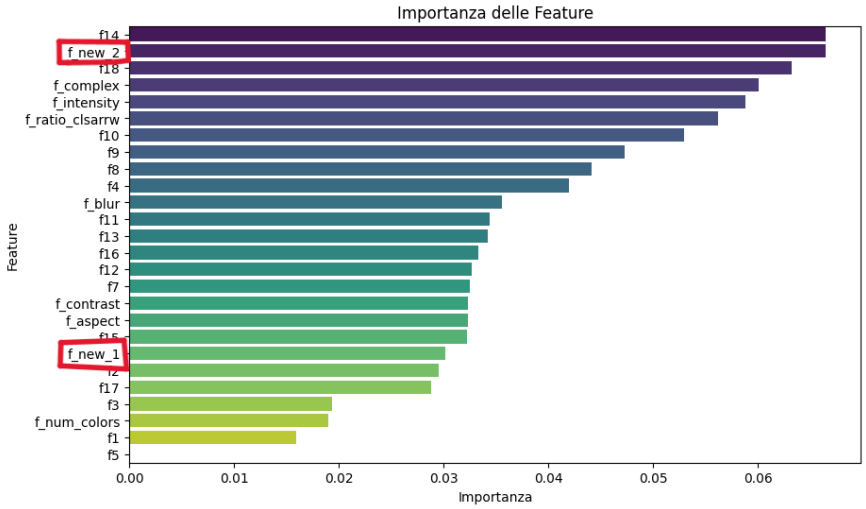

Fig. 9.13 Importance of Feature1 and Feature2

of the original features could be constructed. There are several methods for manipulating and creating new features, such as adding/multiplying two features, generating polynomial features, or aggregating multiple features.

The features created to specifically emphasize the differences between classes 2 and 3 have been guided by a deep understanding of the problem. Since merely observing the graphs is insufficient, not all the created features will be useful, and they will be filtered by importance. Additionally, there is a risk of increasing the model's dimensionality excessively.

The following features have been created:

1. Combining aspects that emphasize the difference between patterns with low quality and those with high quality. For the first feature: the feature 'Rectangle distribution' is multiplied by 'Number of crosses. $f\_new\_1 : f13 * f1$

2. For the second feature: the feature 'Number of lines' is multiplied by 'Number of rectangles. $f\_new\_2 : f18 * f17$

These new features lead to a little reduction in the number of misclassified images as is possible to seein Figure 9.16, despite their relatively high importance within the classifier, as depicted in Figure 9.13. Furthermore, as per the initial objective, the features, as evident from the distribution graphs of features with images from class 2 and 3 in Figure 9.14 and 9.15, exhibit distinct distributions. These differences theoretically facilitate the classifier's task of distinction, but in practice, it remains challenging due to the limited amount of data.

These data and graphs demonstrate the effective work done in creating new features that hold considerable significance within the classifier while highlighting distinctions between classes with a rating of 2 and those with a rating of 3.

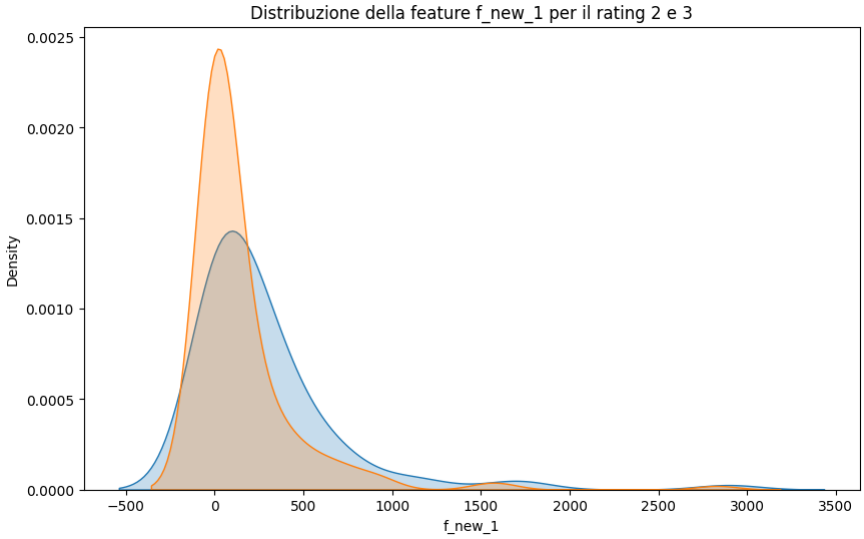

Fig. 9.14 Feature1 distribution analysis

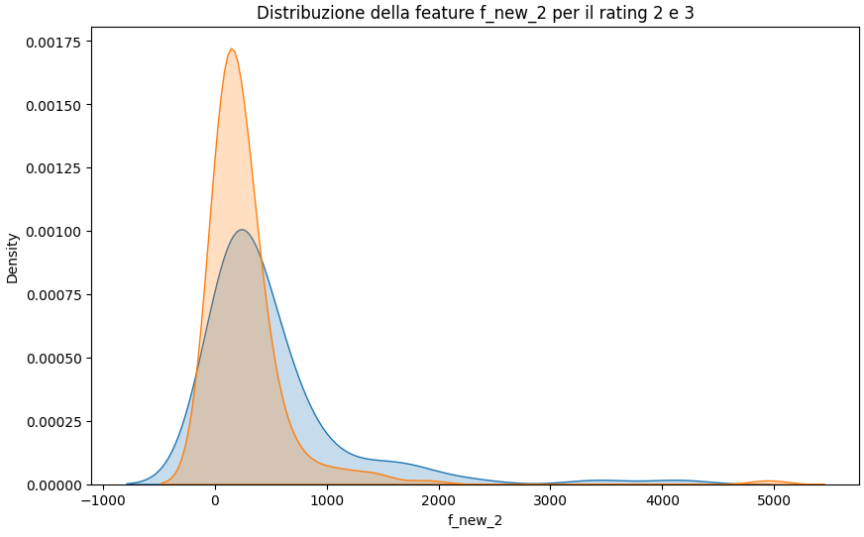

Fig. 9.15 Feature2 distribution analysis

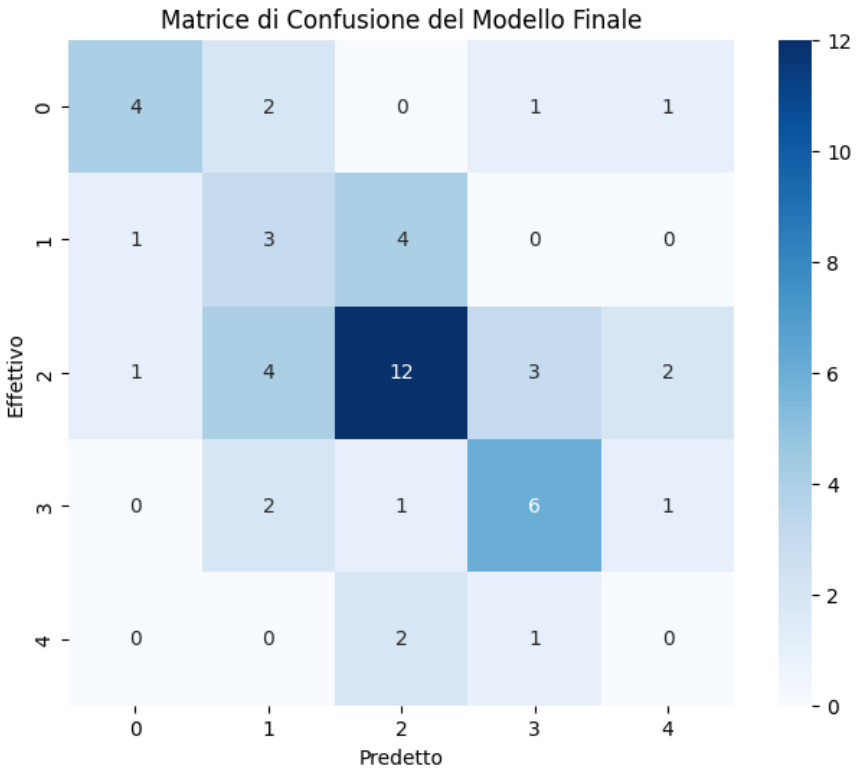

Fig. 9.16 Confusion Matrix with new feature added

By combining the two techniques used for feature engineering in the Random Forest classifier with SMOTE, by adding the features f_new_1 and f_new_2, and removing the features num_color, f_aspect, f_blur, f_contrast, and f1, not only has there been a reduction in missclassifications between the rating 3 and 2 classes but also the following performance improvements have been achieved:

Accuracy = 0.65

Recall = 0.59

F1 = 0.61

Deviation = 0.76

The reason for the limited improvement is believed to be associated with the scanty amount of data, which hinders the thorough learning of differences. Indeed, a larger sample size would be required.

Chapter 10

Final Application

Intro

In the final application, the two models previously created are combined. Therefore, the application will take as input an image of a specific UML diagram and, after processing, provide an output label related to the quality of the diagram, which ranges from 1 to 5.

10.1 Composition of the two model

The models used are the best among all those analyzed. Therefore, for the first phase, we have the Faster R-CNN, which underwent fine-tuning. For the second model, a KNN classifier was employed, and hyperparameter tuning was performed.

Two similar pipelines have been implemented in certain parts but with significantly different performances. Here are the two pipelines:

First Pipeline : The pipeline that the final application will execute is as follows:

1. Receives the input.
2. The first model (Faster R-CNN) segments the UML diagrams into three characteristic objects: arrows, classes, and crosses.
3. Each image portion containing one of these classes will be analyzed by specific algorithms and optimized, primarily based on mathematical concepts such as Canny edge detection, Hough Transformation, and Ramer–Douglas–Peucker.
4. After processing, precise metrics will be extracted from each identified object, totaling 23 metrics that highlight various aspects of UML diagram quality.

5. Classification using the second Random Forest model with the extracted metrics.

6. Final result with a quality label ranging from 1 to 5.

The performance results obtained from the final application are consistent with those observed in the Random Forest classifier since the feature extractor has extracted the same features and they were re-evaluated on the same Random Forest model.

As previously observed, there was an accuracy of 53% across 5 classes, with an average misclassification rate of 0.83. Therefore, misclassifications were not mainly between extremely poor and perfect diagrams but rather within the intermediate quality classes.

Second Pipeline :

The second pipeline focuses on an approach that relies less on the Faster R-CNN for the identification and extraction of metrics from arrows. Here are the steps of the pipeline:

1. Input reception.

2. Segmentation by the Faster R-CNN of classes and "crosses."

3. All segmented images are processed with algorithms based on mathematical concepts such as Canny edge detection, Hough Transformation, and Ramer–Douglas–Peucker to extract all the previously described metrics.

4. To process the arrows, the images are filtered to remove all noise, retaining only the arrows in the image. This is achieved using the bounding boxes of the identified classes. The area of the classes is filled with the background color, ensuring that only the arrows remain in the image.

5. Given the image with only the arrows, Canny and Hough algorithms are executed, lines are selected, and the desired metrics are subsequently computed.

6. The computed metrics are then classified by the Random Forest classifier trained using SMOTE. The best available classifier is chosen.

7. After classification, an input quality rating ranging from 1 to 5 is obtained.

In this case, the performance is superior, as we achieve an accuracy of 57% with a deviation of 0.71. This is because it is more precise in line identification, especially in images with lower ratings. Furthermore, in the case of non-specialized hardware, it also proves to be less computationally intensive, requiring less hardware compared to the RCNN.

```
############ Image 0  ----> RATING : 5. ###########
The Rectangle proximity is already optimal for image 0.
The Number of rectangles is already optimal for image 0.
The Line bends is already optimal for image 0.
The Number of lines is already optimal for image 0.
The Longest line is already optimal for image 0.
The Average line length is already optimal for image 0.
############ Image 1  ----> RATING : 1. ###########
The Rectangle proximity should be decreased for image 1.
The Number of rectangles should be decreased for image 1.
The Line bends should be decreased for image 1.
The Number of lines should be decreased for image 1.
The Longest line should be decreased for image 1.
The Average line length should be decreased for image 1.
############ Image 2  ----> RATING : 2. ###########
The Rectangle proximity should be decreased for image 2.
The Number of rectangles should be decreased for image 2.
The Line bends should be decreased for image 2.
The Number of lines should be decreased for image 2.
The Longest line should be decreased for image 2.
The Average line length should be decreased for image 2.
```

Fig. 10.1 Output.txt file after computation

These performance limitations are primarily attributed to the limited dataset. With larger datasets, especially for the segmenter, and consequently for the classifier, the accuracy would improve.

In addition to the quality label output, a system has been introduced to provide feedback to the designer, offering suggestions on how to enhance the model to achieve a higher rating. The challenge with this type of feedback is that the quality label may be derived from patterns or metrics that are difficult for the designer to control, but they still provide guidelines.

Once the label is output, plots of the features that most significantly influenced the quality of their UML diagram are displayed, with a red point indicating how to improve that specific diagram.

The example in Figure 10.5 pertains to the 'number of lines' metric of a diagram rated with a score of 1. If the red point is positioned in the appropriate space relative to the assigned score, in this case, the model will provide the following advice: "The Number of lines needs to be improved. The Number of lines should be decreased," after conducting an analysis on the median of the feature across different image classes.

This can be highly beneficial for the designer, as it provides them with genuine feedback.

The option to load multiple images into the evaluation folder has also been implemented. After starting the program, once it has completed, a file named "output.txt" will appear in the folder containing evaluations for all the images, along with feedback for the designer for each image. In Figure 10.1, an example of the "output.txt" file is provided.

Following there are three example of different image rating in fig 10.2,10.3,10.4.

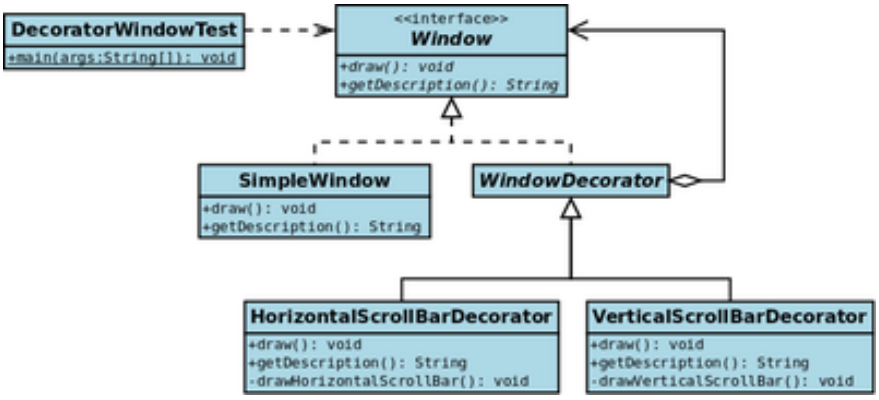

Fig. 10.2 UML schema rated **5**

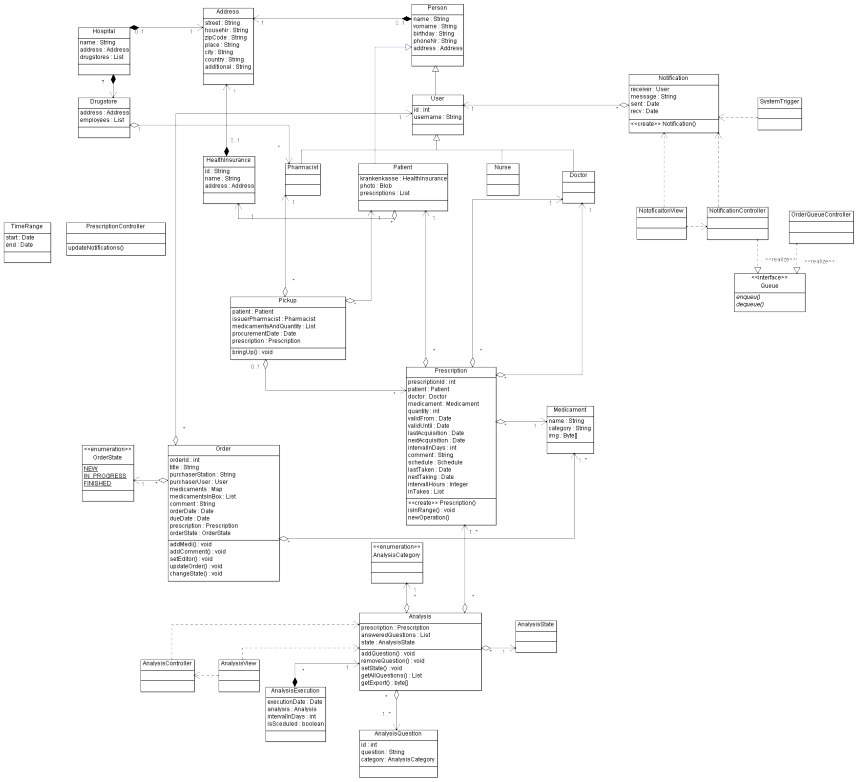

Fig. 10.3 UML schema rated **2**

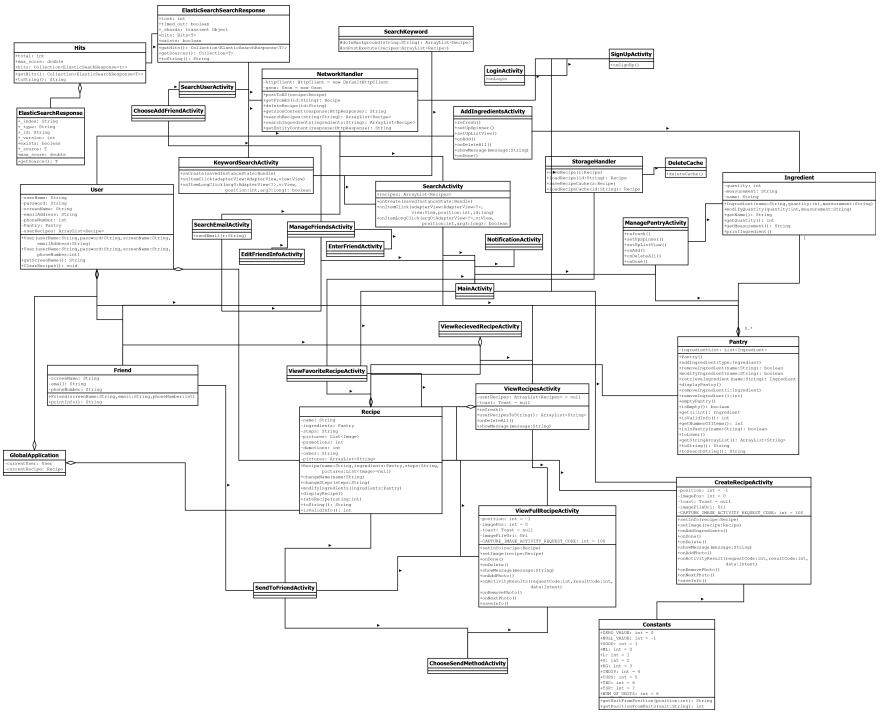

Fig. 10.4 UML schema rated **1**

10.2 Overall performance

The best performance is achieved using the second pipeline, which involves arrow identification without the use of Faster RCNN and utilizing the Random Forest classifier trained with SMOTE.

The performance metrics are as follows:

- Accuracy = 0.65
- Recall = 0.59
- F1 = 0.61
- Deviation = 0.75

These performance results were compared to Paper1 [3], which served as the reference. In the paper, three metrics were employed to assess the final classifier: Pearson's correlation coefficient (denoted as PCC), Mean Absolute Error (MAE), and Relative Absolute Error (RAE).

The authors of the paper used the default parameters for the Weka Random Forest classifier. To facilitate a fair comparison, Weka was downloaded, and the classifier and

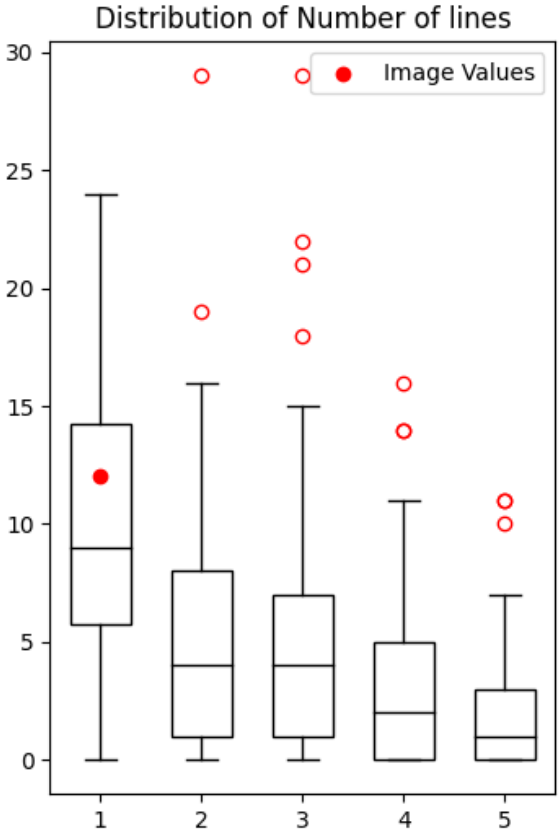

Fig. 10.5 Number of line detected in the Image

experiment were replicated using their intermediate data on the same images I also analyzed. The aim was to determine which feature extraction method yielded superior results. After training the classifier with their intermediate data, the predicted class results were extracted in CSV format, resulting in the following performance for their final model:

- Accuracy = 0.59
- Recall = 0.48
- F1 = 0.50
- Deviation = 0.59

Thus, it is evident that both models yield similar performance, albeit with different architectures. The Faster RCNN pipeline may be slightly slower on equivalent hardware but offers greater deployment ease.

It is important to note that Faster RCNN can significantly enhance its performance with larger datasets. In this case, the training dataset consisted of only 600 images, which is quite limited. Meanwhile, the algorithms used in the paper have already reached their performance peak, as they are not dependent on dataset size.

10.3 Flask application

To create the functional application and perform deployment, the Flask framework was employed. To construct the entire application, the following pages were authored: index.html, result.html, layout.html, upload.js, and custom.css. Flask is a lightweight and flexible web framework for building web applications in Python. It is commonly used for developing web applications and RESTful APIs (Application Programming Interfaces). Flask is known for its simplicity and minimalism, making it an excellent choice for small to medium-sized projects where you need to quickly set up a web server and define routes and endpoints. Flask is often referred to as a "micro" framework because it provides only the essential components for building web applications. Flask provides convenient tools for handling HTTP requests and responses, including handling form data, cookies, and session management. Before running the server, the administrator must load the models into memory, specifically the Faster R-CNN and KNN models. The application then takes an image as input, as depicted in Figure 10.6, subsequently uploads it to the Drive and previews it, as illustrated in Figure 10.7. Following processing by the complete pipeline, the class of membership is displayed, as shown in Figure 10.8.

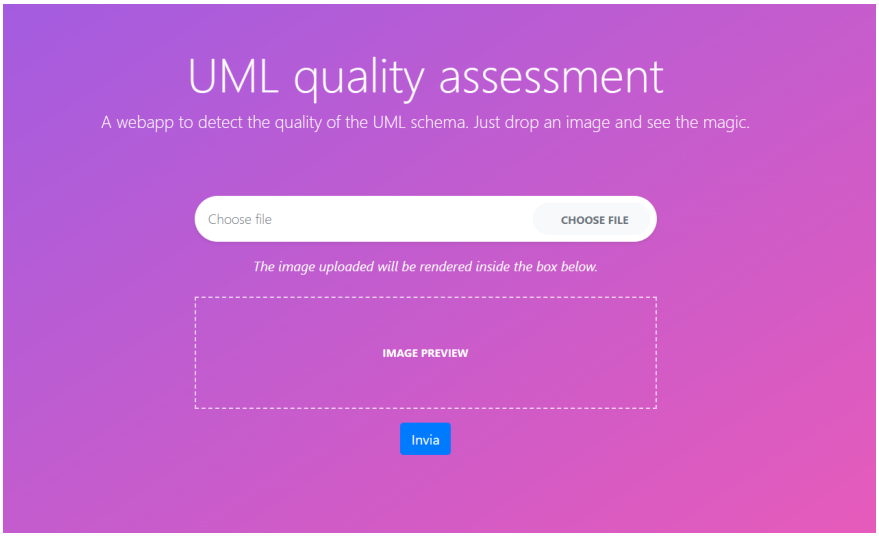

Fig. 10.6 Homepage of website

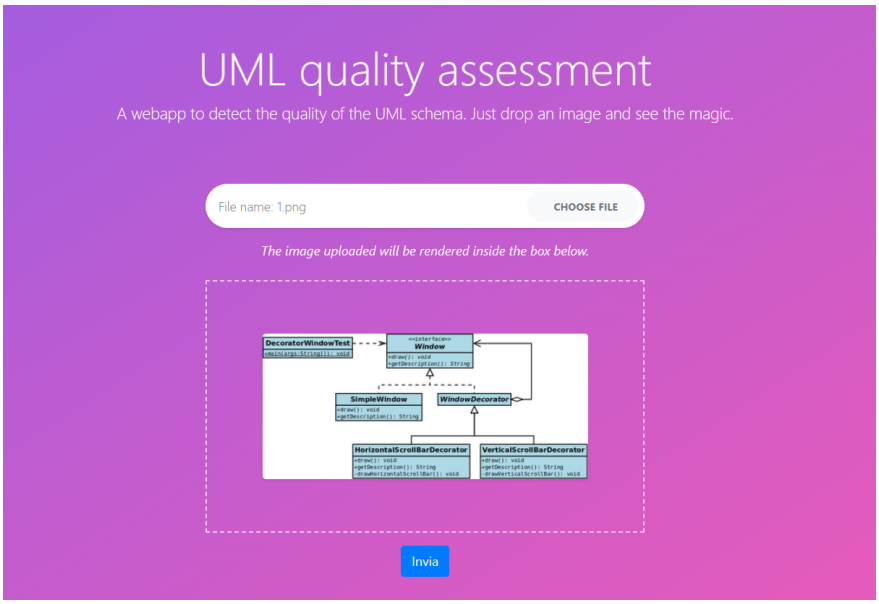

Fig. 10.7 Uploaded UML in the website

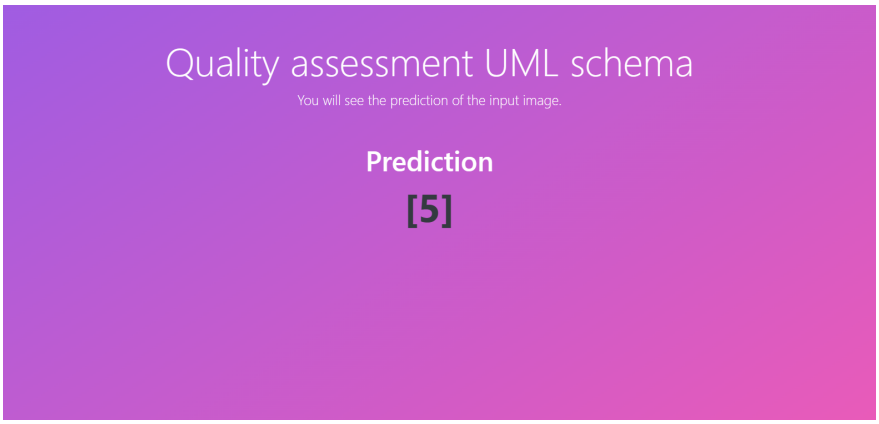

Fig. 10.8 Result page of website

10.4 Conclusion and future work

10.4.1 Conclusion

A fully automated approach has been designed for evaluating the layout of UML class diagrams and providing feedback to the designer. This approach was developed using machine learning and deep learning techniques, employing a comprehensive dataset of 600+ manually labeled diagrams as ground truth for training and testing. The performance of the automatic evaluator was assessed using a quality scale ranging from 1 (indicating very poor) to 5 (indicating very good). The evaluator generated a score on this scale with an accuracy of 60

The features that had the most significant influence on the assignment of quality were thoroughly analyzed, and feature engineering was performed in an effort to enhance overall performance.

The evaluator can serve various purposes: in an industrial context, it can be integrated into Quality Assurance tools to automatically assess the quality of artifacts created within a project, thereby improving the management of modeling artifacts. In an educational setting, our evaluator can be utilized as a component in the automated grading of UML class diagrams, particularly suitable for online learning environments that are gaining popularity. Additionally, our evaluator may find applications in algorithms that aim to automatically generate layouts for class diagrams. For instance, such algorithms are used in reverse engineering scenarios where diagrams represent source code artifacts. In this scenario, our evaluator could serve as an oracle, providing feedback to machine-learning layouting algorithms.

As part of this study, a dataset of UML schema images, including the segmentation of the diagrams in all their parts, was also created manually and validated.

10.4.2 Future work

One of the potential future works that could be implemented is the semantic evaluation of schema correctness, not limiting it solely to readability and clarity but incorporating metrics that also assess adherence to the designer's choices and, in case of errors, the correction of these errors.

There are several challenges in addressing this issue, such as the dataset. In addition to images, it would require textual descriptions or a textual representation of the problem being described and represented. Datasets are challenging to obtain, especially when it comes to schemas used in corporate contexts.

Another issue would be the choice of technology. Using a fine-tuned transformer for this purpose would be ideal, as it could capture the semantic meaning of the problem and provide an overall perspective. Moreover, transformers like GPT-4 are currently capable of generating UML schemas for simpler problems. For instance, by generating JSON code for PlantUML, followed by the use of converters to render the UML code. In Figure 10.9, an example of a management system for a pizzeria written by ChatGPT and then rendered with PlantUML is shown.

The comparison between generated schemas and those received from the designer could potentially be done by evaluating terms with GPT's spatial distance, but it would require in-depth research to identify the most precise schema given a command.

Another consideration is the hardware required to maintain a service with transformers. Dedicated servers or very powerful hardware would be necessary, especially during the training phase. If one intends to use a transformer via API, options like OpenAI's Chat-GPT could be used, or there are open-source transformers like Meta's LAMA2.

Another possible future work, focusing on the qualitative aspect, would be to expand the software to process other types of diagrams, such as sequence diagrams or other diagrams used in the corporate context. The main challenge with this endeavor is primarily the availability of datasets, which are often private or proprietary to companies and, containing sensitive information, are not made public. The pipeline would not require significant modifications since the class and arrow segmenter would work for sequence diagrams as well. However, it would likely be necessary to adapt the extracted and computed features to match the specific characteristics of sequence diagrams. Another aspect that could be expanded upon in this work pertains to the identification of outliers within the classes. Indeed, a system for data and text extraction from the classes has already been developed, but it could be

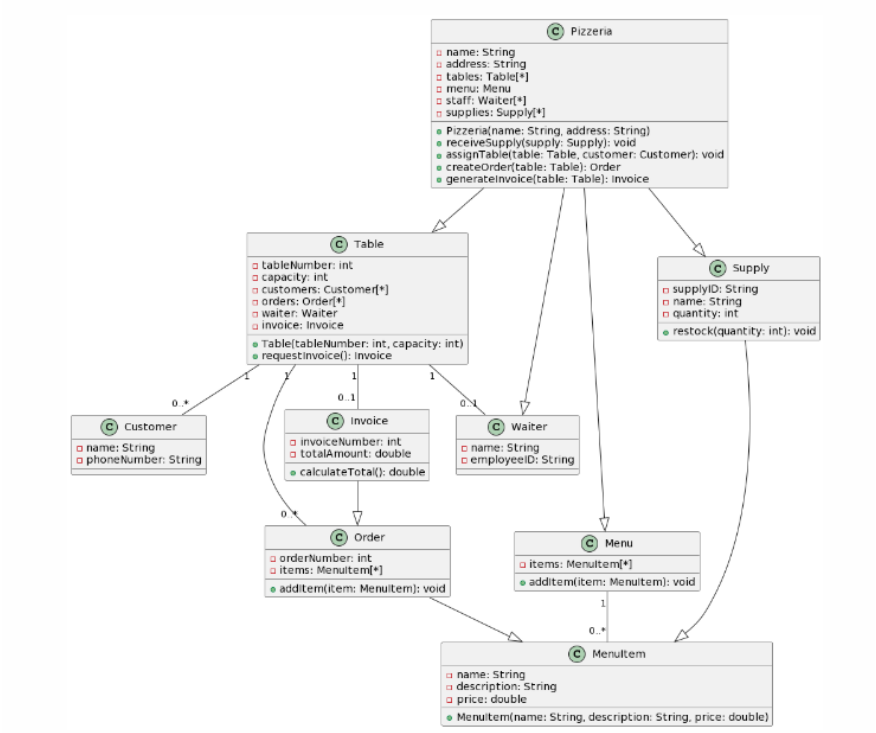

Fig. 10.9 UML schema generated by GPT4

extended to identify classes that do not align with the context. Unfortunately, to accomplish this, a dataset of schemas belonging to the same domain of expertise would be required, such as the aerospace or naval domain, to analyze the vocabulary used and frequent patterns. With the dataset available, it was not possible to identify a domain-specific vocabulary, as the UML diagrams were sourced from random repositories on GitHub, belonging to entirely generic areas of expertise. For this type of work, reliance could be placed on computer vision for text extraction, as previously demonstrated, and transformers for the identification of outliers.

Acknowledgements

Words cannot express my gratitude to my professor and chair of my committee for her invaluable patience and feedback. I am also grateful to my friends and classmates , late-night feedback sessions, and moral support during all the Master Degree. Thanks are also due to my friends in Manfredonia, who have consistently supported me in every decision and aided me in difficult situations over the past 10 years. I owe them a great deal. Lastly, my family, particularly my parents, deserves mention. Their unwavering belief in me has sustained my spirits and motivation throughout this enduring journey known as 'University'. I would also like to express gratitude to myself for never giving up.

References

[1] Bilal Karasneh, M. R. C. (2013). Extracting uml models from images. *CSIT*, 1.

[2] Gautam, S. (2022). Parsing structural and textual information from uml class diagrams to assist in verification of requirement specifications. *The Repository at St. Cloud State*, 1.

[3] Gustav Bergström, Fadhl Hujainah, T. H.-Q. R. J. M. R. C. (2022). Evaluating the layout quality of uml class diagrams using machinelearning. *The Journal of Systems Software*, 1.

[4] Truong Ho-Quang, M. R. C. (2014). Automatic classification of uml class diagrams from images. *Asia-Pacific Software Engineering Conference*, 1.