BarBeR - Barcode Benchmark Repository: Implementation and Reproducibility Notes

Enrico Vezzali[1], Federico Bolelli[1], Stefano Santi[2], and Costantino Grana[1]

[1] University of Modena and Reggio Emilia, Modena, Italy
{name.surname}@unimore.it
[2] Datalogic, S.p.A, Bologna, Italy
stefano.santi@datalogic.com

Abstract. This paper provides a detailed description of how to install, set up, and use "BarBeR" (Barcode Benchmark Repository) to reproduce the results presented in the ICPR 2024 paper "BarBeR: A Barcode Benchmarking Repository". The paper details the tests available in the repository and how the configuration parameters affect and influence experimental results.

Keywords: BarBeR · Barcodes· Benchmark· QR Codes · Public Dataset

1 Introduction

Barcodes, a prevalent form of machine-readable data, have revolutionized data collection and identification due to their cost-effectiveness and efficiency [12]. They are widely used in supply chain management [5], warehousess [4], manufacturing [12], retail [6], and robot guidance [7]. Despite their inception over seven decades ago, barcodes continue to hold their ground in today's digital age, and their use is forecasted to increase in the future [3, 9]. Barcodes come in one-dimensional (1D) and two-dimensional (2D) forms. While 1D barcodes have limited data storage, 2D barcodes offer greater capacity. The process of reading a barcode involves localization and decoding. Recent advancements in edge deep learning have spurred interest in developing deep learning-based barcode localization solutions. Despite the huge interest in the field, several issues prevent definitive conclusions about methods' effectiveness and applicability. The first is that existing research relies on small datasets that do not reflect real-world scenarios accurately and make training deep learning models difficult. Then there is the problem of reproducibility. The lack of public code implementations makes replicating results challenging. Finally, different studies use different metrics, leading to contradictory comparisons even with identical algorithms and datasets. To address these challenges, we have developed "BarBeR" (Barcode Benchmark Repository) [10, 11] — an open-source benchmark for barcode localization with standardized test protocols and evaluation metrics. BarBeR includes multiple localization algorithms and a large annotated dataset of 8 748 barcode images. Our goal is to enhance reproducibility and facilitate reliable algorithm comparisons within the research community.

2 The Dataset

For this project, we collected a large dataset of barcodes to compare algorithms and train object detection neural networks. After conducting a thorough literature review, we gathered datasets totaling 8 748 images with 9 818 annotated barcodes (8 062 linear and 1 756 two-dimensional). The BarBeR dataset follows the VGG annotation format [1]. The Dataset consists of two components: images in jpg format and a folder of JSON files with the annotations. The annotations contain the polygon shape of each barcode plus three important characteristics of the code: its type, PPE (pixels per element, other times referred to as PPM or pixels per module), and encoded string. In total, we have 19 classes for the argument `Type`, 18 of which identify a particular barcode type (Code 128, Code 39, EAN-2, EAN-8, EAN-13, GS1-128, IATA 2 of 5, Intelligent Mail Barcode, Interleaved 2 of 5, Japan Postal Barcode, KIX-code, PostNet, RoyalMail Code, UPC, Aztec, Datamatrix, PDF-417, and QR Code). The last class is the class '1D' which indicates a 1D barcode that was not automatically assigned to a category because it was impossible to decode. The dataset contains images in a wide range of different resolutions (from 200×141 to 5 984×3 376) captured in various settings with different devices, featuring diverse subjects and environments. This diversity includes barcodes in different lighting conditions, some underexposed or overexposed, and others with variable lighting. Additionally, the dataset contains both planar and skewed or warped barcodes, as well as barcodes affected by blur, noise, or partial obstruction.

3 The Evaluation Framework

BarBeR is available on GitHub[3] and includes various detection methods and scripts to train neural networks for barcode detection. The associated dataset and the trained models can be downloaded from our website.[4] The repository has been developed with Linux as the main target OS. However, the code is not architecture-specific and it is possible to build and run all the tests on different Operative Systems. Both x86-64 and ARM architectures have been tested without any reported issues.

The `algorithms` folder contains a Python class for every localization algorithm available (Table 1). The `config` folder contains the YAML configuration files for each Python script that needs a configuration file. These configuration files are examples and can be modified depending on the configuration needed. The `python` folder contains all Python files, including all test scripts. Other important folders in the repository are `results`, which is the default path to the generated results of the tests, and `scripts`, which contains bash scripts to run multiple tests in the pipeline. Finally, there are a few other folders in the repository, which contain the code to build the needed libraries to run the tests.

[3] https://github.com/Henvezz95/BarBeR
[4] https://ditto.ing.unimore.it/barber

Table 1. This is a list of all the available localization algorithms featured in BarBeR.

Algorithm	File	1D Detection	2D Detection	Multi-Label
Gallo *et al.* [2]	gallo_detector.py	✓	✗	✗
Soros *et al.* [8]	soros_detector.py	✓	✓	✗
Yun *et al.* [13]	yun_detector.py	✓	✗	✗
Zamberletti *et al.* [14]	zamberletti_detector.py	✓	✗	✓
Zharkov *et al.* [15]	zharkov_detector.py	✓	✓	✓
Ultralytics Models	ultralytics_detector.py	✓	✓	✓
Detectron2 Models	detectron2_detector.py	✓	✓	✓
Pytorch Models	pytorch_detector.py	✓	✓	✓

4 Setting Up the Repository

To correctly install and run the current version of BarBeR, the following packages, libraries, and utilities are required:

- CMake 3.13 or higher (https://cmake.org);
- OpenCV and OpenCV Contrib 4.0 or higher (http://opencv.org);
- Boost 1.53 or higher;
- A C++ compiler supporting C++11 or higher;
- A list of Python libraries contained in `requirements.txt`.

The installation procedure is well detailed in the aforementioned GitHub repository and the main steps can be resumed as follows:

- Clone the repository;
- Generate the BarBeR project using CMake;
- Open the project folder and build;
- Download the dataset;
- [Optional] Download the pre-trained models.

The dataset comes in .zip format. Once Unzipped, you will find 2 folders inside: `Annotations` and `Dataset`. If you place these two folders directly inside the BarBeR folder, there is no need to change the paths of the configuration files. Similarly, the folder `Saved Models` can be unzipped and placed in the BarBeR folder with no need to change configuration paths.

4.1 Converting the Annotations

To run a test, we need COCO annotations divided into `train.json`, `val.json`, and `test.json`. To configure how to split the annotations, a YAML configuration file is used. An example of such a file is `config/generate_coco_annotations_config.yaml`. With the configuration file, we can select which files to use and which annotations, the train-test split size, and if we are using K-fold cross-validation. The script used to generate the annotation is `python/generate_coco_annotations.py`, which takes as input a configuration file and optionally the index k, which indicates the index of the current cross-validation test (use 0 if K-fold cross-validation is not used). It can be called with the following command:

```
python3 python/generate_coco_annotations.py -c \
./config/generate_coco_annotations_config.yaml  -k 0
```

If we also need to train an Ultralytics model, we need YOLO annotations, which will be generated with the following command:

```
python3 python/convert_coco_to_yolo.py -c \
./annotations/COCO/ -o ./dataset/
```

5 Reproduce the Tests

The repository is equipped with a variety of test scripts, each supporting diverse configurations. All test scripts are written in Python and take as input argument a YAML configuration file and output a YAML file containing multiple evaluation metrics for every tested algorithm. The available metrics are Precision, Recall, and F1 score at different IoU thresholds. For algorithms that also output a confidence score, the Benchmark also computes the Average Precision (AP@.5, AP@[.5:.95]) for each class, the mean Average Precision (mAP@.5, mAP@[.5:.95]) and the Average Recall (AR100, AR10, AR1). Finally, the benchmark allows the filtering of these metrics depending on the size of the ground truth and its pixel density.

5.1 Single-Class Localization

The *Single-Class Localization Test* runs all the selected algorithms considering only images with the selected type of barcodes (1D or 2D). The Python script that runs the Single-Class detection test is `test_single_class.py`: inputs are a configuration file and a path pointing to where the output report will be saved. In this repository, there are three working examples of configuration files, one for single-ROI 1D barcode detection (`./config/test1D_singleROI.yaml`), one for multi-ROI 1D barcode detection (`./config/test1D_multiROI.yaml`), and one for single-ROI 2D barcode detection (`./config/test2D_singleROI.yaml`). The Listing 1.1 shows an example of how a configuration file should be formatted. To launch the test run the following command:

```
python3 python/test_single_class.py -c \
./config/test_config.yaml -o ./results/test_results.yaml
```

The results will be saved in the desired output path in YAML format. These are some of the options that can be tuned in the configuration file:

- `coco_annotations_path:` path to the folder containing the annotations in COCO format (`test.json`).
- `longest_edge_resize:` this is used to select the resolution of the resized images. In particular, it indicates the number of pixels of the longest side of the resized image. If this setting is set to a number lower than 0, no rescaling will be applied, and the localization algorithm will run on the image at the original resolution;
- `class:` indicates the class of barcodes used for the tests, which can be either 1D or 2D;

```yaml
# Single Class Detection Configuration
coco_annotations_path: ./annotations/COCO/
longest_edge_resize: 640
class: 1D
single_ROI: true
bins: [0,1,2,3,4,5,6,7,100]
algorithms:
- args:
    lib_path: ./path_to_lib/lib1.so
    arg1: 11
  class: First_detector
  library: first_detector
  name: First_Algorithm
- args:
    lib_path: ./path_to_lib/lib2.so
    arg1: 5
    arg2: 3
  class: Second_detector
  library: second_detector
  name: Second_Algorithm
```

Listing 1.1. Configuration file example for single class localization.

- `single_ROI`: since some algorithms only support single ROI localization, it is possible to select only the images containing a single barcode. To do so, set this option to `true`;
- `bins`: if this option is present, the YAML file of the results will also present different metrics depending on the PPE range. Each element of the bin is an edge of a PPE range. For example, if $bins = [0, 1, 2]$ we will have additional metrics for barcodes with PPE $\in [0, 1]$ and PPE $\in [1, 2]$;
- `algorithms`: indicates the algorithms to test. For each algorithm `library` indicates the library where the function is contained and `class` is the name of the detector class. The `name` key is the name that will be assigned to this algorithm in the results report. In addition, each algorithm is assigned a dictionary of arguments (`args`).

5.2 Multi-Class Localization

This script runs all the selected algorithms on all the images of the test set. As for Single-Class detection, we can choose the resizing resolution and which algorithms are included in the test. It does not support the `bins` keyword, though. The Python script that runs the Single-Class detection test is `test_multi_class.py`. The inputs are a configuration file and a path pointing to where the report with the results will be saved. In this repository, there is one working configuration file for Multi-Class detection (`./config/test_multiclass.yaml`). This script will generate the results for Precision, Recall, F1-score, Average Precision (AP@.5, AP@[.5:.95]) for each class, mean Average Precision (mAP@.5,

```yaml
# Timing Benchmark Configuration
coco_annotations_path: ./annotations/COCO/
longest_edge_resize: 640
num_repeats: 3
num_threads: 1
step: 1
define: &device 'cpu'

- args:
    model_path: ./Saved Models/zharkov_640_0.pt
  class: Zharkov_detector
  library: zharkov_detector
  name: Zharkov
  device: *device
```

Listing 1.2. Configuration file example for the timing benchmark.

mAP@[.5:.95]) and Average Recall (AR100, AR10, AR1). These metrics will be saved to a YAML file in the designated output path. To launch the test run the following command:

```
python3 python/test_multiclass.py -c \
./config/test_config.yaml -o ./results/test_results.yaml
```

5.3 Timing Benchmark

The script `time_benchmark.py` measures the time required to run the localization algorithms. The times can be taken from the average times on all datasets or a subsection of it. It is possible to measure the algorithms' performance on a single core or multiple cores as well as on GPU. As for the previous test scripts, the required inputs are a path to a configuration file and a path to the destination folder for the generated report. In the repository, there is a configuration file named `config/timing_config.yaml` that can be used to replicate the results showed in the original paper [11]. To launch the test run the following command:

```
python3 python/time_benchmark.py -c \
.config/timing_config.yaml -o ./results/timing_results.yaml
```

The Listing 1.2 shows an example of a configuration file for the timing benchmark. To reduce the impact of the background processes, detections are repeated multiple times per image, and the lowest time is taken. How many times to repeat the test for every image is specified with the keyword `num_repeats` in the configuration file. The final time, reported in the output report in YAML format, is the average between the results for every image. The number of threads is specified by the keyword `num_threads`. The target device can be selected by changing the definition of the variable `device`. The supported targets are `cpu` pr `gpu`. Only neural network models support GPU benchmarking at the moment.

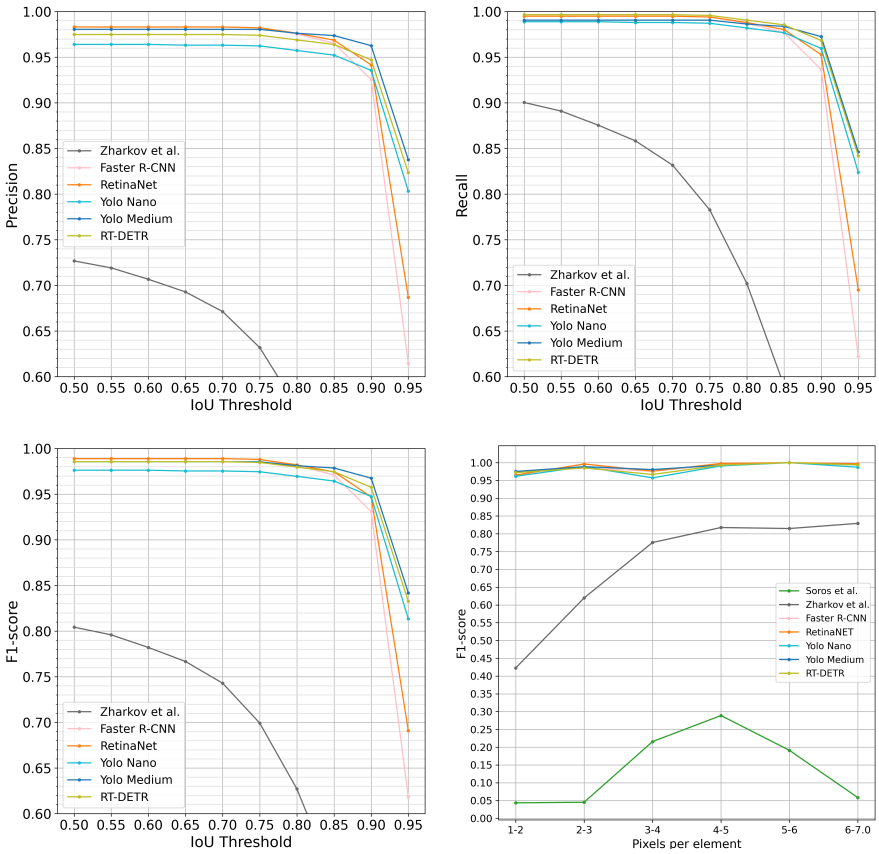

Fig. 1. Example of the 4 graphs generated by the script `single_class_graphs.py`. The first three graphs represent Precision, Recall, and F1-score curves of 2D barcode detection algorithms at different values of IoU threshold. The last graph shows the F1-scores on 2D barcodes at different ranges of pixels per element.

If the target device is set to `gpu`, the `num_threads` keyword will be ignored. Finally, the `step` keyword is used to run the test on a subsection of the dataset. If `step` is 1, all images will be used. With a `step` of n only 1 image every n will be used in the test.

The time results presented in the BarBeR paper [11] were obtained by running the benchmark on two contrasting platforms: a high-end PC and a Raspberry Pi 3B+. The high-end PC was equipped with a 24-core AMD Ryzen Threadripper Pro 5965WX CPU, 128 GB of DDR4 RAM, and an RTX 4090 GPU. Changes in the testing hardware will lead, of course, to changes in the results obtained.

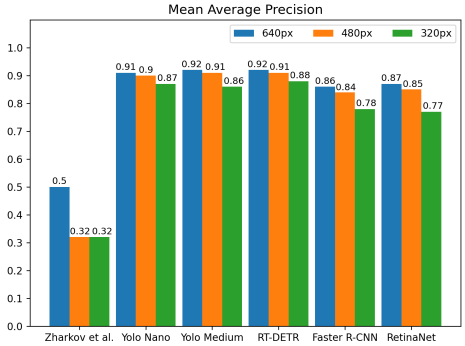

Fig. 2. Example of graph generated by the script `multi_class_graphs.py`. It incorporates results from at different scales. The graph shows the values of mAP@[.5:.95] of different models at three different scales: longest side resized to 640 pixels, longest side resized to 480 pixels, and longest side resized to 320 pixels.

5.4 Visualize the Results

To generate a graph from the results generated by a Single-Class detection test run the following command:

```
python3 python/visualizer/single_class_graphs.py
```

It is necessary to select the right barcode type, changing the variable `ttype`, which could be '1D' or '2D'. Graphs will be generated inside `results/graphs`. This script generates four graphs. The first three represent Precision, Recall, and F1-scores at different values of the IoU threshold. The last graph is generated only if the `bins` keyword was used. It represents the F1 scores (with an IoU threshold of 0.5) of the tested detection methods at different ranges of pixels per element. In Fig. 1, we see an example of the four generated graphs.

To generate a graph from the results generated by a Multi-Class detection test, run the following command:

```
python3 python/visualizer/multi_class_graphs.py
```

It will generate a graph similar to Fig. 2. It can incorporate results from different scales. In particular, the graph shows the values of mAP@[.5:.95] of different models at the different scales the tests were conducted. To change the path of the input reports, the variable `base_path`, present in both scripts, must be changed.

6 Training New Models

The repository allows the training of new detection models from Ultralytics or Detectron2. In addition, it contains the script to train the neural network proposed by Zharkov *et al.* [15]. This last script can easily be adapted to train any Pytorch detection model. To train a model with Ultralytics, run `python/ultralytics_trainer.py`. A configuration file is needed (e.g., `config/ultralytics`

`_training_config.yaml`), as well as an output path for the trained model (default folder is "Saved Models"). Detectron2 model can be trained with the script `python/detectron2_trainer.py`. A configuration file is needed (e.g., `config/detectron2_training_config.yaml`), as well as an output path for the trained model. Finally, to train a Zharkov model, you can run the script `Zharkov2019/zharkov_trainer.py`. Again, a configuration file is needed (e.g. `config/zharkov_training_config.yaml`), as well as an output path for the trained model.

7 K-Fold Cross-Validation

To run K-fold cross-validation, it would be necessary to run the scripts multiple times manually. Since running the scripts multiple times and changing the configuration each time would be too cumbersome, it is possible to automate the process using a bash script. The python file `python/create_configuration_yaml.py` is used to generate a new configuration each time. Scripts can also be used to train multiple networks, each with a different test set. The folder `scripts` contains examples of how to use scripts to run multiple Python files one after the other:

- `k_fold_training_ult.sh`: trains 5 Ultralytics networks, leaving out a different chunk of the dataset from the training set each time. Each trained network will end its file name with a different index, going from 0 to 4;
- `k_fold_training_det.sh`: trains 5 Detectron2 networks, leaving out a different chunk of the dataset from the training set each time. Each trained network will end its file name with a different index, going from 0 to 4;
- `k_fold_training_zharkov.sh`: trains 5 Zharkov networks, leaving out a different chunk of the dataset from the training set each time. Each trained network will end its file name with a different index, going from 0 to 4;
- `k_fold_test_1D_singleROI.sh`: runs 5 single-class tests with 1D barcodes. The index in the file names of the models will be incremented automatically;
- `k_fold_test_multiclass.sh`: runs 5 multi-class tests. The index in the file names of the models will be incremented automatically.

All detection tests presented in the BarBeR paper [11] used 5-fold cross-validation. To replicate the 1D localization results, run this command:

```
source scripts/k_fold_test_1D_singleROI.sh
```

To replicate the 2D localization results, instead run the following command:

```
source scripts/k_fold_test_2D_singleROI.sh
```

Finally, to replicate the results shown in the Multi-class Detection section, run this command:

```
source scripts/k_fold_test_multiclass.sh
```

The results obtained should be equal to the ones shown in the BarBeR paper [11]. However, different platforms could employ different floating point standards, theoretically leading to very small changes in the results.

8 Conclusions

This report provides a comprehensive guide on how to utilize the BarBeR benchmark and replicate the results presented in the BarBeR ICPR paper [11]. Specifically, it details the repository and dataset locations, project setup instructions, and test execution procedures. The detection results should be replicable on every platform with high precision. On the other hand, changes in the hardware used to run the tests will strongly impact the results of the timing test.

References

1. Dutta, A., Zisserman, A.: The VIA annotation software for images, audio and video. In: Proceedings of the 27th ACM international conference on multimedia. pp. 2276–2279 (2019)
2. Gallo, O., Manduchi, R.: Reading 1D Barcodes with Mobile Phones Using Deformable Templates. IEEE Transactions on Pattern Analysis and Machine Intelligence **33**(9), 1834–1843 (2010)
3. Kapsambelis, C.: Bar Codes Aren't Going Away! (2005)
4. Kubáňová, J., Kubasáková, I., Čulík, K., Štítik, L.: Implementation of barcode technology to logistics processes of a company. Sustainability **14**(2), 790 (2022)
5. McCathie, L.: The advantages and disadvantages of barcodes and radio frequency identification in supply chain management. Phd thesis, School of Information Technology and Computer Science (2004)
6. Melek, C.G., et al.: Datasets and methods of product recognition on grocery shelf images using computer vision and machine learning approaches: An exhaustive literature review. Engineering Applications of Artificial Intelligence **133** (2024)
7. Soliman, A., Al-Ali, A., Mohamed, A., Gedawy, H., Izham, D., Bahri, M., Erbad, A., Guizani, M.: AI-based UAV navigation framework with digital twin technology for mobile target visitation. Engineering Applications of Artificial Intelligence **123**, 106318 (2023)
8. Sörös, G., Flörkemeier, C.: Blur-resistant joint 1D and 2D barcode localization for smartphones. In: Proceedings of the 12th International Conference on Mobile and Ubiquitous Multimedia. pp. 1–8 (2013)
9. Vaishnavi Shyamsundar Mate, S.M.: Barcode Reader Market Size, Share, Competitive Landscape and Trend Analysis Report by Type, by Application : Global Opportunity Analysis and Industry Forecast, 2023-2032 (2023)
10. Vezzali, E., Bolelli, F., Santi, S., Grana, C.: State-of-the-art Review and Benchmarking of Barcode Localization Methods. Engineering Applications of Artificial Intelligence pp. 1–29 (2025)
11. Vezzali, E., Bolelli, F., Santi, S., Grana, C., et al.: Barber: A barcode benchmarking repository. In: 2024 27th International Conference on Pattern Recognition (ICPR) (2024)
12. Weng, D., Yang, L.: Design and Implementation of Barcode Management Information System. In: Information Engineering and Applications: International Conference on Information Engineering and Applications. pp. 1200–1207 (2012)
13. Yun, I., Kim, J.: Vision-based 1D Barcode Localization Method for Scale and Rotation Invariant. In: TENCON - IEEE Region 10 Conference. pp. 2204–2208 (2017)

14. Zamberletti, et al.: Robust Angle Invariant 1D Barcode Detection. In: 2013 2nd IAPR Asian Conference on Pattern Recognition. pp. 160–164 (2013)
15. Zharkov, A., Zagaynov, I.: Universal Barcode Detector via Semantic Segmentation. In: 2019 International Conference on Document Analysis and Recognition (ICDAR). pp. 837–843 (2019)