MNIST - Study Case on different Algorithms

Blidea Tudorel Alexandru

December 28, 2022

Abstract

Nowadays, there are a lot of PDF editors who are able to recognize handwritten text, those editors can identify and transform the text into a digital one. This project tries to see which algorithms, to detect handwritten digits, are the most suitable for a real-time text editor.

Contents

1 Introduction

2 Tested Algorithms

There are a lot of algorithms in the Artificial Intelligence field. I went with a few of the most popular ones.

- Naive Bayesian
  - The Simple Version
  - Tuned with Bagging (Smaller Naive Bayesians which votes for a digit).
- K Nearest Neighbor
  - The Simple Version
  - Tuned with Distance Transform
- Convolutional Neural Network

3 Dataset

The used dataset is the THE MNIST DATABASE of handwritten digits.

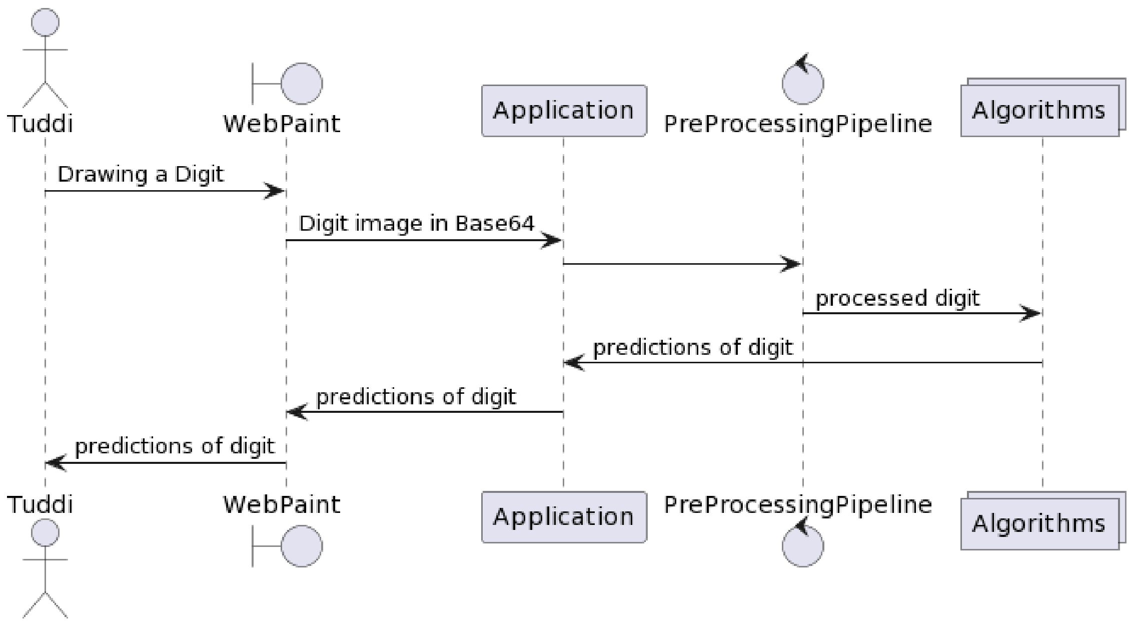
4 Application

To have a better interaction with the Algorithms and see their Power, I implemented a web application using

- Spring with Java (I implemented the algorithms from scratch in Java, and exposed the interaction with them using REST services)
- VueJS (for the Front-End application)

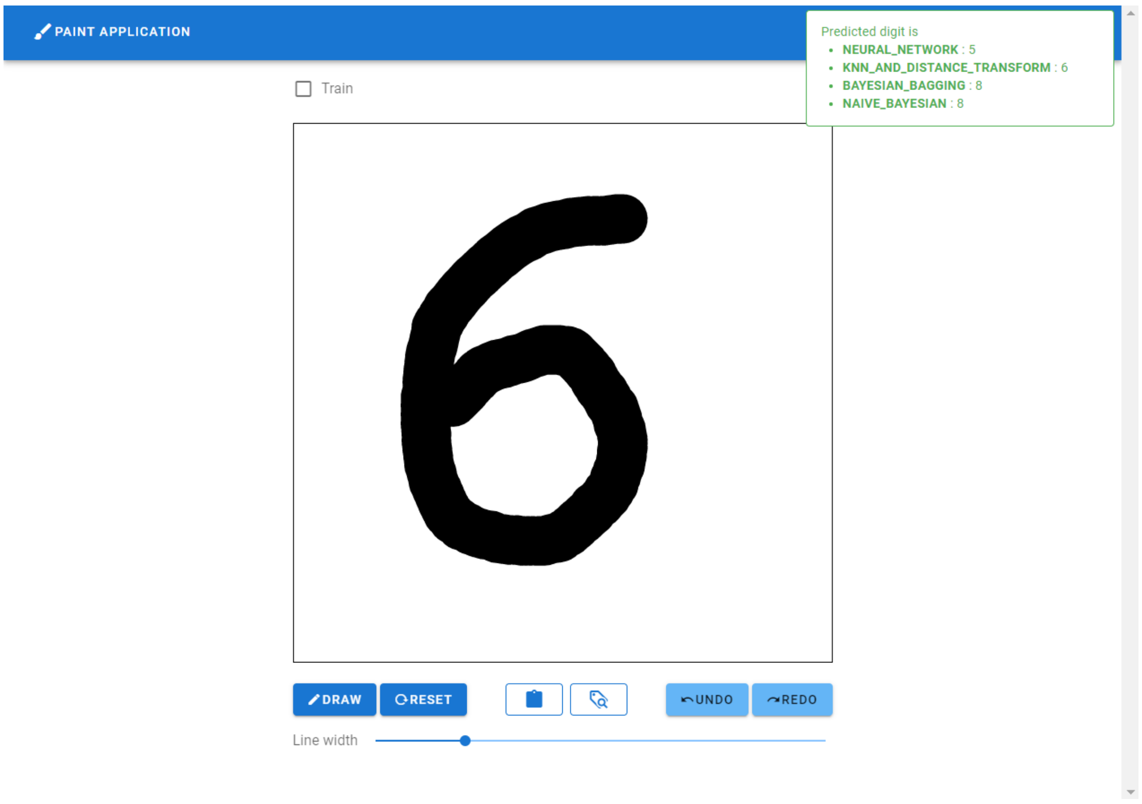
4.1 The flow of the Application

As can be seen below, a lot of processing details are hidden by the web paint. The user simply draws a digit in **WebPaint**, which sends it to the **Application**, here, applied the **PreProcessing** operations and after that, is sent to the **Algorithms** which will be applied to predict/guess/identify the digit.



4.2 WebPaint Preview

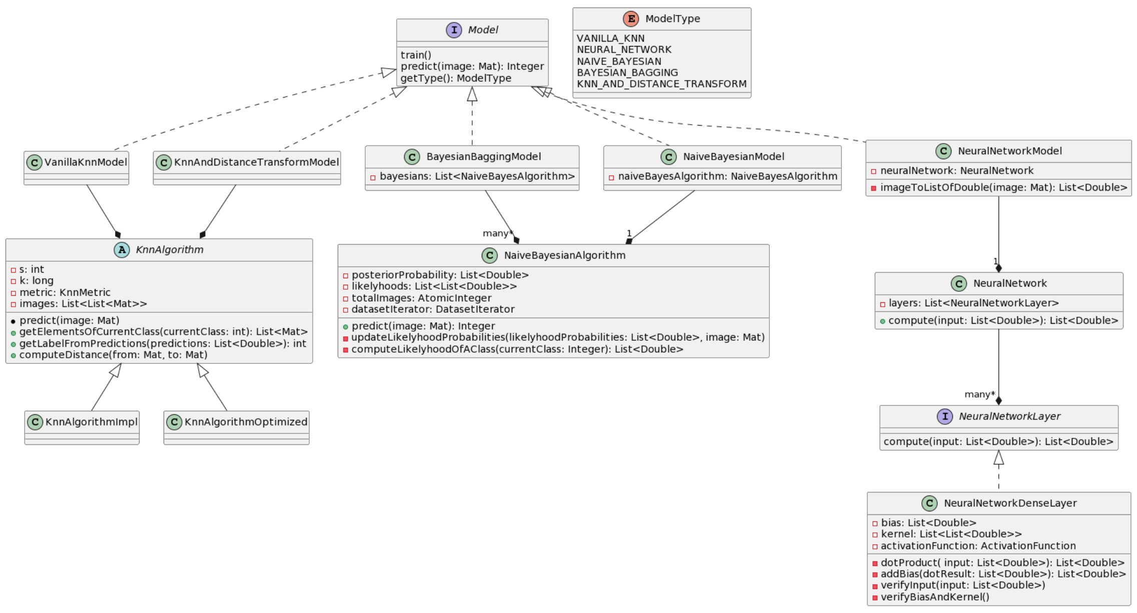
The WebPaint has an area to Draw (a simple implementation of Paint), and a button to send the drawing to the Application. After the the processing is done, a response will come back with the name of each algorithm and the predicted digit.



Predicted digit is
- NEURAL_NETWORK : 5
- KNN_AND_DISTANCE_TRANSFORM : 6
- BAYESIAN_BAGGING : 8
- NAIVE_BAYESIAN : 8

5 Algorithms detailed

5.1 Block Diagram of Algorithms


I Model
train()
predict(image: Mat): Integer
getType(): ModelType
E ModelType
VANILLA_KNN
NEURAL_NETWORK
NAIVE_BAYESIAN
BAYESIAN_BAGGING
KNN_AND_DISTANCE_TRANSFORM
C VanillaKnnModel
C KnnAndDistanceTransformModel
C BayesianBaggingModel
bayesians: List<NaiveBayesAlgorithm>
C NaiveBayesianModel
naiveBayesAlgorithm: NaiveBayesAlgorithm
C NeuralNetworkModel
neuralNetwork: NeuralNetwork
imageToListOfDouble(image: Mat): List<Double>
A KnnAlgorithm
s: int
k: long
metric: KnnMetric
images: List<List<Mat>>
predict(image: Mat)
getElementsOfCurrentClass(currentClass: int): List<Mat>
getLabelFromPredictions(predictions: List<Double>): int
computeDistance(from: Mat, to: Mat)
many
1
C NaiveBayesianAlgorithm
posteriorProbability: List<Double>
likelyhoods: List<List<Double>>
totalImages: AtomicInteger
datasetIterator: DatasetIterator
predict(image: Mat): Integer
updateLikelyhoodProbabilities(likelyhoodProbabilities: List<Double>, image: Mat)
computeLikelyhoodOfAClass(currentClass: Integer): List<Double>
1
C NeuralNetwork
layers: List<NeuralNetworkLayer>
compute(input: List<Double>): List<Double>
many
I NeuralNetworkLayer
compute(input: List<Double>): List<Double>
C KnnAlgorithmImpl
C KnnAlgorithmOptimized
C NeuralNetworkDenseLayer
bias: List<Double>
kernel: List<List<Double>>
activationFunction: ActivationFunction
dotProduct(input: List<Double>): List<Double>
addBias(dotResult: List<Double>): List<Double>
verifyInput(input: List<Double>)
verifyBiasAndKernel()

5.2 K Nearest Neighbor

5.2.1 Pseudocode

```
input toPreditct: Image

functions
    function distance (from, to): Score
end functions

data
    scoreList: List of scores
end data

for digitImage in MNIST_Dataset then
    add in scoreList distance(toPredict, digitImage) and label of digitImage
end for

sortedScoreList = sort the scoreList by score

firstKScores = peek first K elements from sortedScoreList

output: most frequent label from firstKScores
```

5.2.2 Code

```java
    public int predict(final Mat toPredict) {
        List<KnnPair> pairs = IntStream.range(0, numberOfClasses)
                .boxed()
                // For every class (0, 1, ..., 9)
                .flatMap(currentClass -> getElementsOfCurrentClass(currentClass)
                        .parallelStream()
                        // Create the list of "distances"
                        .map(image -> KnnPair.builder()
                                .currentClass(currentClass)
                                .distance(computeDistance(toPredict, image))
                                .build()))
                // Then sort the resulted list
                .sorted(Comparator.comparing(KnnPair::getDistance))
                // And take the first K
                .limit(getK())
                .collect(Collectors.toList());
        // With the first k, create an array of predictions
        List<Double> predictions = DoubleStream.generate(() -> 0.0)
                .boxed()
                .limit(numberOfClasses)
                .collect(Collectors.toList());

        for (KnnPair pair : pairs) {
            predictions.set(pair.getCurrentClass(),
                predictions.get(pair.getCurrentClass()) + 1);
        }
        // and return the class of most frequent label
        return getLabelFromPredictions(predictions);
    }
```

5.3 Vanilla Knn

This is the first algorithm implemented, even in laboratory work. I find this one of the most interesting, because it has a great performance in production, but is still slow (see in the 1).

5.3.1 Improving Performance

The best way to improve the performance, of most algorithms, is to tune the hyperparameters. The Vanilla KNN algorithm had only the K, the S and the Distance Function. (The S should not be changed too much), but I worked with the K and the Distance. And I found that the best performances were for k = 10 and the LP_1 (see the metrics in the Appendix).

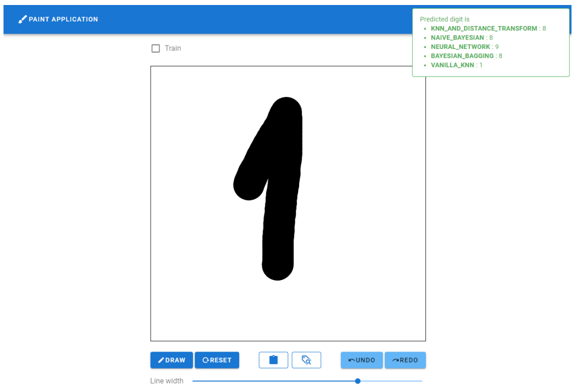
5.3.2 Confusion Matrix

$$\begin{bmatrix} 939 & 3 & 1 & 0 & 0 & 14 & 20 & 2 & 1 & 0 \\ 0 & 1132 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 22 & 235 & 674 & 22 & 8 & 4 & 8 & 39 & 18 & 2 \\ 3 & 78 & 5 & 867 & 1 & 14 & 1 & 19 & 11 & 11 \\ 0 & 60 & 0 & 0 & 808 & 0 & 17 & 3 & 0 & 94 \\ 8 & 54 & 0 & 33 & 5 & 737 & 20 & 7 & 1 & 27 \\ 11 & 25 & 0 & 0 & 19 & 6 & 896 & 1 & 0 & 0 \\ 0 & 96 & 1 & 0 & 1 & 0 & 0 & 893 & 0 & 37 \\ 27 & 96 & 1 & 36 & 13 & 25 & 11 & 13 & 700 & 52 \\ 10 & 27 & 1 & 8 & 18 & 1 & 2 & 30 & 0 & 912 \end{bmatrix}$$

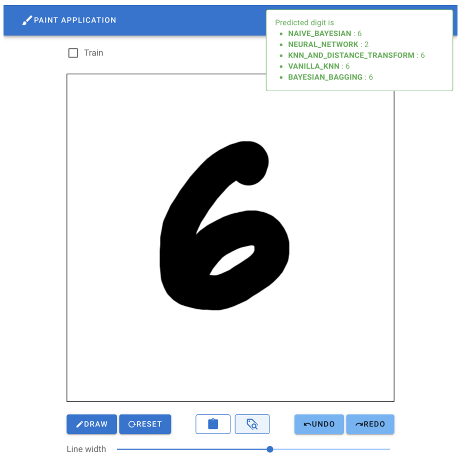
accuracy $= 85.58\%$

5.3.3 Production Results

As it can be seen below, the Vanilla KNN has better predictions in the application, even than the Neural Network, which I think might deal with over fitting.



Predicted digit is
- KNN_AND_DISTANCE_TRANSFORM : 8
- NAIVE_BAYESIAN : 8
- NEURAL_NETWORK : 9
- BAYESIAN_BAGGING : 8
- VANILLA_KNN : 1



Predicted digit is
- NAIVE_BAYESIAN : 6
- NEURAL_NETWORK : 2
- KNN_AND_DISTANCE_TRANSFORM : 6
- VANILLA_KNN : 6
- BAYESIAN_BAGGING : 6

5.4 Convolutional Neural Network

Neural Networks are a **hot topic** right now, because of high accuracy scores and simplicity. Given their structure, are able to run on **GPUs** and/or **TPUs** which makes their training fast enough. Also, their simple structure (Matrix computations), makes them suitable to run on low-power devices such as embedded systems. **Unfortunately, the Neural Networks are black boxes.**

For benchmark results visit the table 1.

For that implementation, I used Dense Layers and the activation function R_eLU (which can be seen in the appendix).

The computation formula is:

$denseLayer(input) = activationFunction(dot(inputs, kernel) + bias)$

Which means:

$$denseLayer(<i_1, i_2, \ldots, i_n>) = ReLU\left(<i_1, i_2, \ldots, i_n> \cdot \begin{pmatrix} w_{11} & w_{12} & \ldots w_{1n} \\ w_{21} & w_{22} & \ldots w_{2n} \\ & \ddots & \\ w_{n1} & w_{n2} & \ldots w_{nn} \end{pmatrix} + <b_1, b_2, \ldots, b_n> \right) \tag{1}$$

5.4.1 Code

```
public List<Double> compute(List<Double> input) {
    List<Double> result = input;
    for (NeuralNetworkLayer layer : layers) {
        result = layer.compute(result);
    }
    return result;
}
```

5.4.2 Improving Performance

The performance could be easily improved by: **normalizing the data**, adding/removing hidden layers, increasing/decreasing the number of perceptrons in layers, and playing with the activation function.

For this project, I went with a simple configuration, because the Neural Network will be added to an embedded system. And also, I wanted to keep the response time as low as possible, as it can be seen in the Jupyter Notebook, every percentage of accuracy was achieved with bigger changes.

5.4.3 Confusion Matrix

$$\begin{bmatrix} 964 & 1 & 1 & 3 & 0 & 2 & 4 & 1 & 3 & 1 \\ 0 & 1106 & 2 & 1 & 2 & 1 & 2 & 2 & 19 & 0 \\ 4 & 5 & 983 & 7 & 4 & 2 & 7 & 4 & 16 & 0 \\ 2 & 0 & 10 & 940 & 1 & 19 & 0 & 6 & 23 & 9 \\ 1 & 2 & 5 & 3 & 920 & 2 & 9 & 11 & 3 & 26 \\ 2 & 2 & 1 & 20 & 1 & 838 & 10 & 5 & 12 & 1 \\ 8 & 3 & 2 & 0 & 5 & 15 & 922 & 0 & 3 & 0 \\ 0 & 3 & 14 & 20 & 3 & 2 & 0 & 969 & 5 & 12 \\ 6 & 0 & 5 & 9 & 5 & 7 & 5 & 4 & 928 & 5 \\ 9 & 3 & 0 & 6 & 15 & 5 & 0 & 10 & 12 & 949 \end{bmatrix}$$

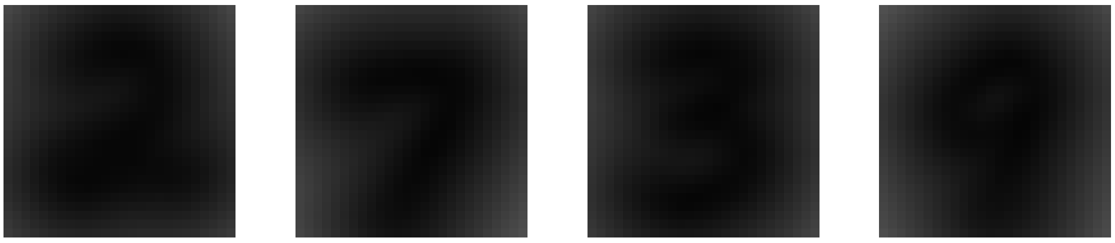
accuracy = 95.19%

5.4.4 Production Results

In the WebPaint, as it can be seen for the previous algorithm, the Neural Network doesn't have the results it is expected to have.

5.5 Knn and Distance Transform

This is the algorithm that I proposed the first time. Initially, the idea was great, I think, but the dataset images are too small, due that, the distance transforms does not provide enough information.



Packed Distance Transforms of MNIST digits

The above images are used by the algorithm internally. It is obvious that the digits are not quite visible (darken and diffused, which makes them harder to recognize).

5.5.1 Pseudocode

The same as for Vanilla KNN (but instead of images, it has Distance Transforms).

5.5.2 Code

The same as for Vanilla KNN (but instead of images, it has Distance Transforms).

5.5.3 Improving Performance

What improved the performance overall, was to have fewer digits (that made the computation faster), but for that, to not lose performance, I ”packed” the distance transforms (made an average of them), this helped with the visibility, but not that much. The above images 5.5 are ”packed”.

5.5.4 Confusion Matrix

$$\begin{bmatrix}
966 & 0 & 1 & 3 & 0 & 0 & 2 & 0 & 8 & 0 \\
1 & 477 & 7 & 23 & 0 & 0 & 5 & 0 & 622 & 0 \\
133 & 0 & 718 & 53 & 5 & 0 & 42 & 0 & 81 & 0 \\
106 & 0 & 21 & 789 & 0 & 0 & 1 & 1 & 89 & 3 \\
84 & 0 & 4 & 9 & 470 & 0 & 48 & 0 & 197 & 170 \\
390 & 0 & 3 & 140 & 1 & 32 & 6 & 0 & 316 & 4 \\
197 & 0 & 0 & 2 & 3 & 0 & 692 & 0 & 64 & 0 \\
108 & 0 & 22 & 17 & 4 & 0 & 2 & 558 & 209 & 108 \\
75 & 0 & 3 & 84 & 1 & 0 & 7 & 2 & 798 & 4 \\
92 & 0 & 2 & 20 & 4 & 0 & 5 & 1 & 261 & 624
\end{bmatrix}$$

accuracy $= 61.24\%$

5.6 Naive Bayesian

This is a well-known algorithm for the MNIST dataset. Is called **Naive** because it does not consider the order of digits or their positioning. It has good accuracy and a nice response time (as can be seen in the benchmark table 1).

Check in the appendix the Naive Bayesian Likelyhoods B.2

5.6.1 Pseudocode

```
... computing the likelihoods ...
... computing the posteriorProbabilities ...

input image

scoreList = emptyList()

for every class then
    score = posteriorProbability of class
    for pixel of image then
        score += isPixelActive(pixel) ? likelihood of pixel : 0
    end for
    add score and class to scoreList
end for

output pick class of best score from scoreList
```

5.6.2 Code

```
    public train(DatasetIterator datasetIterator) {
        this.datasetIterator = datasetIterator;
        likelyhoods = IntStream.range(0, numberOfClasses)
                .parallel()
                .boxed()
                .map(this::computeLikelyhoodOfAClass)
                .collect(Collectors.toList());
        posteriorProbability = posteriorProbability.stream()
                .map(p -> Math.log10(p / totalImages.get()))
                .collect(Collectors.toList());
```

```java
    }

    public Integer predict(Mat image) {
        return IntStream.range(0, numberOfClasses)
                .parallel()
                .boxed()
                .map(currentClass -> {
                    int height = (int) image.size().height;
                    int width = (int) image.size().width;
                    double probability = posteriorProbability.get(currentClass);
                    for (int i = 0; i < height; i++) {
                        for (int j = 0; j < width; j++) {
                            int index = i * height + j;
                            if (PixelHelper.isPixelActive(image.get(i, j))) {
                                probability += likelyhoods.get(currentClass).get(index);
                            }
                        }
                    }
                    return
                         MnistPair.builder().currentClass(currentClass).probability(-probability).build();
                })
                .sorted(Comparator.comparing(MnistPair::getProbability))
                .collect(Collectors.toList())
                .get(0)
                .getCurrentClass();
    }

    private void updateLikelyhoodProbabilities(List<Double> likelyhoodProbabilities, Mat
         image) {
        int height = (int) image.size().height;
        int width = (int) image.size().width;
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                int index = i * height + j;
                double previousValue = likelyhoodProbabilities.get(index);
                likelyhoodProbabilities.set(index, previousValue +
                     PixelHelper.getPixelValue(image.get(i, j)));
            }
        }
    }

    private List<Double> computeLikelyhoodOfAClass(Integer currentClass) {
        List<Double> likelyhoodProbability = DoubleStream.generate(() -> 1.0)
                .boxed()
                .limit(imageSize)
                .collect(Collectors.toList());
        int counter = 0;
        for (Mat image: DatasetIterator.of().getTrainingDataOfClass(currentClass)) {
            counter++;
            updateLikelyhoodProbabilities(likelyhoodProbability, image);
            posteriorProbability.set(currentClass, posteriorProbability.get(currentClass)
                 + 1);
        }
        totalImages.addAndGet(counter);
        final Integer numberOfElements = counter;
        likelyhoodProbability.replaceAll(probability -> Math.log10(probability /
             numberOfElements));
        return likelyhoodProbability;
    }
```

5.6.3 Improving Performance

To improve the performance of the algorithm, I came up with the next, where I created smaller classifiers and I put them to vote, that's why I called it Bayesian Bagging because is a Bagging Algorithm.

5.6.4 Confusion Matrix

$$\begin{bmatrix}
956 & 0 & 0 & 1 & 0 & 1 & 2 & 1 & 19 & 0 \\
0 & 468 & 15 & 4 & 0 & 0 & 6 & 0 & 642 & 0 \\
50 & 0 & 858 & 11 & 1 & 0 & 26 & 1 & 84 & 1 \\
29 & 0 & 55 & 810 & 0 & 2 & 5 & 1 & 101 & 7 \\
32 & 0 & 12 & 1 & 558 & 0 & 31 & 0 & 220 & 128 \\
130 & 0 & 9 & 125 & 4 & 213 & 13 & 1 & 381 & 16 \\
63 & 0 & 23 & 0 & 2 & 2 & 823 & 0 & 45 & 0 \\
24 & 0 & 19 & 7 & 7 & 0 & 2 & 693 & 179 & 97 \\
27 & 0 & 12 & 28 & 1 & 0 & 5 & 3 & 895 & 3 \\
35 & 0 & 13 & 9 & 19 & 0 & 0 & 5 & 184 & 744
\end{bmatrix}$$

accuracy = 70.18

5.7 Bayesian Bagging

5.7.1 Pseudocode

```
input image

predictionsList = emptyList()
for every smallNaiveBayes then
    prediction = prediction of smallNaiveBayes on image
    add prediction to predictionsList
end for

output most frequent class from predictionsList
```

5.7.2 Code

```java
    @Override
    public Integer predict(final Mat image) {
        List<Integer> predictions = Collections.synchronizedList(IntStream.range(0,
             numberOfClasses).boxed().map(value -> 0).collect(Collectors.toList()));
        bayesians.parallelStream().forEach(bayesianAlgorithm -> {
            int prediction = bayesianAlgorithm.predict(image);
            predictions.set(prediction, predictions.get(prediction) + 1);
        });

        int maxLabel = 0;
        int max = predictions.get(0);
        for (int i = 1; i < numberOfClasses; i++) {
            if (max < predictions.get(i)) {
                maxLabel = i;
                max = predictions.get(i);
            }
        }
        return maxLabel;
    }
```

5.7.3 Improving Performance

For this algorithm, I tried the cross-validation and tested how many smaller classifiers bring the best accuracy and I found that s = 200 brings the best accuracy. But this algorithm has the same accuracy as the previous one. Even the response time is smaller (as it can be seen in the table 1), so I don't really consider it an Improvement, but it was a nice try.

5.7.4 Confusion Matrix

$$\begin{bmatrix}
956 & 0 & 0 & 1 & 0 & 1 & 2 & 1 & 19 & 0 \\
0 & 468 & 15 & 4 & 0 & 0 & 6 & 0 & 642 & 0 \\
50 & 0 & 858 & 11 & 1 & 0 & 26 & 1 & 84 & 1 \\
29 & 0 & 55 & 810 & 0 & 2 & 5 & 1 & 101 & 7 \\
32 & 0 & 12 & 1 & 558 & 0 & 31 & 0 & 220 & 128 \\
130 & 0 & 9 & 125 & 4 & 213 & 13 & 1 & 381 & 16 \\
63 & 0 & 23 & 0 & 2 & 2 & 823 & 0 & 45 & 0 \\
24 & 0 & 19 & 7 & 7 & 0 & 2 & 693 & 179 & 97 \\
27 & 0 & 12 & 28 & 1 & 0 & 5 & 3 & 895 & 3 \\
35 & 0 & 13 & 9 & 19 & 0 & 0 & 5 & 184 & 744
\end{bmatrix}$$

accuracy = 70.18

5.8 Benchmark Table

	KNN and DT	Vanilla KNN	Bayesian NaiveB	Vanilla NB	CNN
test ds size	300.0	300.0	300.0	300.0	300.0
avg resp time (msec/image)	19.67	2137.63	19.63	0.49	2.14
std	4.07	766.0	11.89	0.14	0.65
min resp time (msec/image)	12.48	333.34	1.84	0.18	1.36
25%	16.27	1558.74	12.52	0.39	1.53
50%	19.05	2006.54	17.2	0.47	1.96
75%	22.76	2623.54	23.29	0.57	2.79
max resp time (msec/image)	30.26	5032.45	82.45	1.07	3.63
avg rep time (min)	0.82	89.068	0.82	0.021	0.089

Table 1: Benchmark results. The running time on an image, and the report generation average time.

A Hyperparameters

A.1 Metrics

- $LP = ||x-y||_p = \left(\sum_{i\leq n}(x_i - y_i)^p\right)^{\frac{1}{p}}$
- $LP_1 = ||x-y||_1 = \sum_{i\leq n}|x_i - y_i|$
- $LP_2 = ||x-y||_2 = \sqrt{\sum_{i\leq n}(x_i - y_i)^2}$
- $distanceTransformScore(image) = \sum_{pixelIndex=0}^{nPixels} score(getPixel(pixel, pixelIndex), getPixel(dt, pixelIndex))$, where $nPixels = numberOfPixels(image)$ and

$$score(imagePixel, dtPixel) = \begin{cases} dtPixel & isPixelActive(imagePixel) \\ 0 & otherwise \end{cases}$$

A.2 Activation Functions

- $R_eLU(x) = max(0, x)$

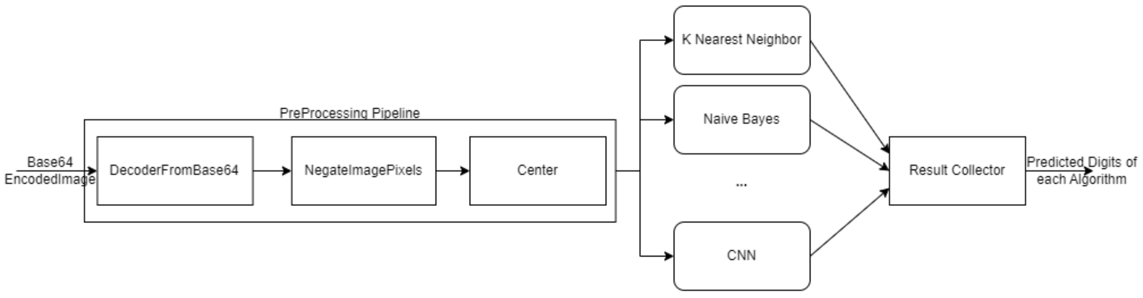
B More about Algorithms

B.1 Pipeline

The **WebPaint** produces a black digit written on white background. Also, the user can draw the digit however it wants and wherever it wants. For that, I defined a **Preprocessing Pipeline** which has different steps.

1. **DecoderFromBase64**: Due to data transmission over the ethernet, the image is encoded in base64 and it requires decoding.
2. **NegateImagePixels**: The **WebPaint** produces a black digit written on white background, this pipeline inverts the colours, so the image will look like a MNIST one and will fit better on implemented algorithms.
3. **Center**: The user can draw the digit however it wants and wherever it wants, to improve the prediction, I choose to centre the shape.

After the preprocessing, the image is sent to each algorithm which response with a prediction. The resulting predictions are collected and returned back to the user.

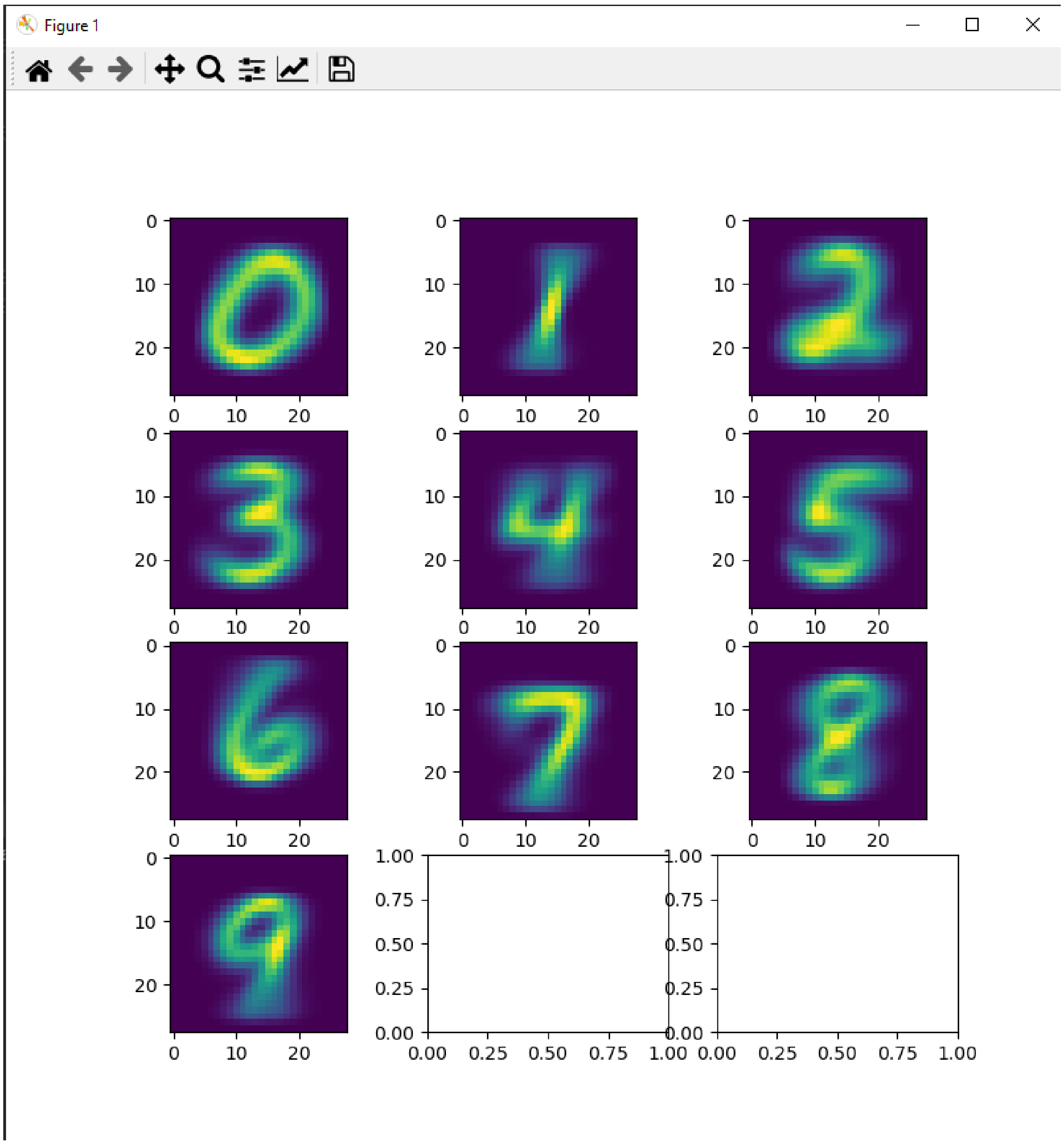


B.2 Naive Bayesian Likelyhoods

As can be seen in the plot below, the likelihood probability for each class looks like the digit of the class, with an intense colour where the pixel is more frequent for that image. for example, most of 7 do not have a strike, and the 2 have the most frequent pixels on the loop. This classifier is a white box because, you know what and how recognizes, you can fool, for example:

- **Drawing 7 a little bit upper**, this will make the classifier think is a 2 (due to the similar shape).
- **Draw where are the most frequent pixels**, this is in case you really want to be recognized as the class...

  What improved the performance of this classifier, was to centre the shape.



Naive Bayesian Likelyhoods plot

References