Department of Computer Science
Technical University of Cluj-Napoca

Pattern Recognition Systems

Laboratory activity

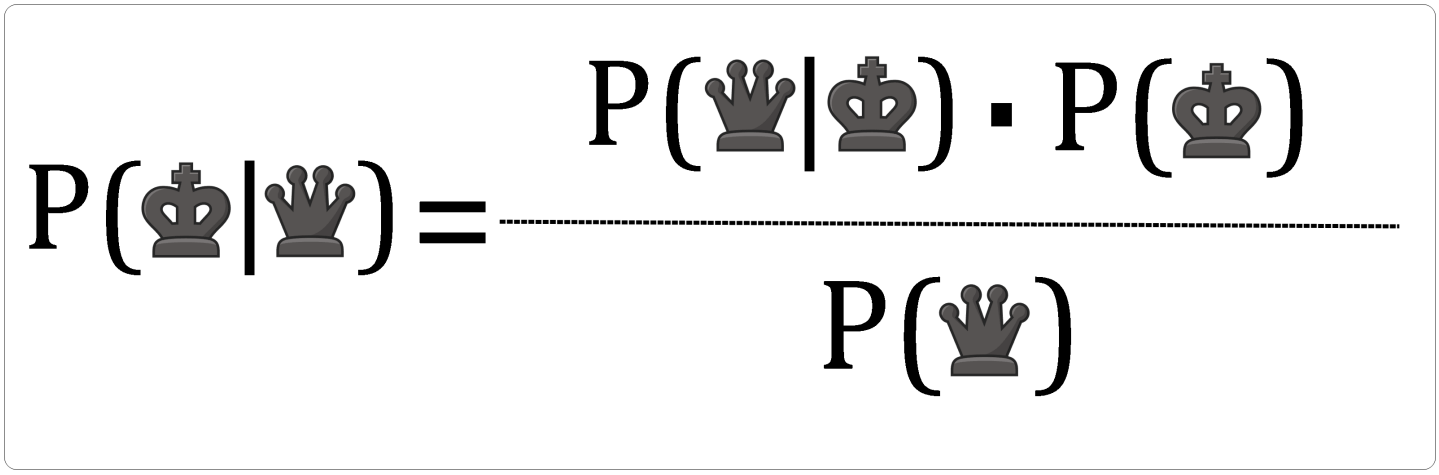
Digital Chess Piece Recognition using Naive Bayesian Classifier

Trufin Radu-Sebastian

Group: 30244

Supervisor: Asist.dr.ing Varga Robert

Date: 13-01-2021



Contents

This project aims to implement a digital chess piece classifier. This could be a great tool in the development of an AI that plays against itself, starting from an input board state that is given from an image, or a learning platform that teaches the user different tactics or important world chess championship games. The chess pieces will be distinguished using the Naive Bayes algorithm, which can produce accurate classification results with a minimum training time and capacity.

∗ ∗ ∗

Research

The digital chess piece classification problem is relatively similar to the digit recognition problem. The Modified National Institute of Standards and Technology (MNIST) database contains a lot of handwritten digits used for training in various image processing problems. Most of the available chess piece datasets consist of real chess pieces and not their digital form, therefore I had to create the dataset by myself, collecting and modifying the information from the most popular online chess platforms (Chess.com, Lichess and Chess24). There are 6 different pieces in the game of chess (pawn, bishop, knight, rook, queen, king) and there are two sides (white and black pieces) therefore the total number of classes is 12. Each train directory currently has between 23 and 30 different sample pieces (each picture having the size of 90 x 90 pixels), while the testing directories have about 4-5 instances each. The training size consists of 311 different files and it may see a growth in the future, if the popular chess platforms provide new skins.

Figure 1.1: Queen training set

Theoretical description

The Naive Bayes classifier is based on Bayes' theorem of probability (shown in the cover of this paper).

The conditional probability that an event X (a cause) belongs to a class piece(K) (an outcome) can be computed from knowledge of the probability of each cause and the conditional probability of the outcome of each cause:

$$\mathsf{P}(\text{piece}(K) \mid X) = \frac{\mathsf{P}(X \mid \text{piece}(K)) \cdot \mathsf{P}(\text{piece}(K))}{\mathsf{P}(X)}$$

This equation finds the probability that the given data X belongs to a class piece(K), thus we can decide the best candidate by choosing the class with the highest probability among all possible classes. Naive Bayes algorithm assumes that the features are independent (for simpler calculation cost):

$$\mathsf{P}(X \mid \text{piece}(K)) = \prod_{i=0}^{d} \mathsf{P}(X_i \mid \text{piece}(K))$$

For the training set, a feature matrix X will be defined in order to store the intensity values of each pixel from the 90 x 90 images. The matrix will have the dimension of $N \cdot d$, where N is the number of training instances and $d = 90 \cdot 90$ is the number of features which is equal to the image size. Another structure is needed, this time a column vector named Y where the class labels will be stored.

The priori probability for each class K can be easily calculated by dividing the train instances of the K class to the total number of training instances:

$$\mathsf{P}(\text{piece}(K)) = \frac{N_K}{N}$$

In essence, we need to find the likelihood of having a feature (a pixel from the image) colored with black, given that the piece class is K (that means the pixel is part of the chess piece because the white pieces images will be converted to their negative form). The likelihood can be computed by using the following expression:

$$\mathsf{P}(\text{black}(X_j) \mid \text{piece}(K)) = \frac{1 + \text{count}(\text{black}(X_{ij})) \ \ \& \ \ Y_i = K}{N_K + 12}$$

Where the numerator represents how many features from the feature vector (i means we iterate through the training instances, which are the rows of the matrix) share the same property and belong to the same class. In order to avoid likelihoods having the value of 0, the Laplace smoothing method is used by adding an additional 1 to the numerator and also adding the total number of classes to the denominator. The probability for white version of the piece is simply computed by subtracting the previous result from 1.

$$\mathsf{P}(\text{white}(X_j) \mid \text{piece}(K)) = 1 - \mathsf{P}(\text{black}(X_j) \mid \text{piece}(K))$$

After combining the previous results, we can rewrite the posterior probability in the following form:

$$\mathsf{P}(\text{piece}(K) \mid X) = \mathsf{P}(\text{piece}(K)) \cdot \prod_{i=0}^{d} \mathsf{P}(X_i \mid \text{piece}(K))$$

Where the denominator $\mathsf{P}(X)$ is ommited because its value is the same for every class. The Naive Bayes classifier is often referred as the maximum a posteriori (MAP) decision rule. The best candidate, which is the prediction result, can be mathematically expressed as:

$$K = \arg\max_j \ \mathsf{P}(\text{piece}(K_j)) \cdot \prod_{i=0}^{d} \mathsf{P}(X_i \mid \text{piece}(K_j))$$

The MAP is mathematically used to obtain a point estimate of an unobserved quantity on the basis of empirical data. In order to avoid numerical precision problems in the computation process, we can use the logarithm on both sides and transform the product of probabilities into a sum of probabilities. The ordering of the posterior values does not change when the log function is applied, thus the result will be the same. The final expression for the predicted class will be written in the following manner:

$$\log(K) = \arg\max_j \ \log(\mathsf{P}(\text{piece}(K_j))) + \sum_{i=0}^{d} \log(\mathsf{P}(X_i \mid \text{piece}(K_j)))$$

Implementation details

I have implemented the classifier using the Python language, with the help of the OpenCV2 library. The main function will initialise the following important variables:

1. nr_classes → The total number of classes (6 different pieces for each color)
2. interest → In order to increase computing speed, we can define an interest zone which will reduce the size of the feature vector. Instead of iterating through the entire image of size (90 x 90) we can simply define a square region that shares the same center as the original image (this can be helpful because most of the pieces share the same white background). For example, if the interest value is 30, then the square size will be 30 x 30.
3. train → If this flag is set to false, the training process is skipped and the test function will be immediately called, otherwise the training process restarts.
4. debug → If this flag is set to true, then we can see the probabilities for each class for every tested image.

```
def bayes():

    nr_classes = 12
    d = 90
    interest = 90
    train = False
    debug = False
    bayes_train(nr_classes, d, train, interest, debug)
```

As mentioned previously, the train size is not the same for every piece, but that is not a problem if we implement a function that returns the number of files from a directory. Now we can calculate the total train size N and the train size N_K for every class $K \in \{1 \cdots 12\}$.

```
def count_folder_files(path):
    count = 0
    for root, dirs, files in os.walk(path):
        for f in files:
            count = count + 1
    return count
```

In the first part of the train function the number of train instances and the interest zone for the images are initialised, then the X and Y vectors with their corresponding dimensions. The priori structure is also declared and with the help of the function that counts the training instances from a specific folder and a dictionary which converts an integer to string that defines a chess piece, we can easily populate the vector:

```
def bayes_train(nr_classes, train, interest, debug):

    n = count_folder_files(sys.path[1] + f'\\train\\')

    d = interest * interest

    x = np.zeros((n, d), dtype=int)
    y = np.zeros((n, 1), dtype=int)

    priori = np.zeros((nr_classes, 1), dtype=float)

    for i in range(0, nr_classes):
        priori[i, 0] = count_folder_files(sys.path[1]
        + f'\\train\\{class_to_piece[i]}') / n
```

The next part of the function iterates through all 12 train directories, binarizes the images (also converts them to their negative form if the chess piece is white). The intensity values are inserted into the feature vector (X structure) and the corresponding class (which is basically the subdirectory name) is appended into the Y vector. If the interest variable is set at its default value of 90, notice that the for loops will iterate in the (0, 90) range, otherwise those loops will represent a smaller window with the same center as the original image, with the size of (interest x interest).

```
if train:

    row = 0
    for subdir, dirs, files in os.walk(sys.path[1] + f'\\train\\'):
        for f in files:

            filepath = subdir + os.sep + f

            image_train = cv2.imread(filepath)
            gray_image = cv2.cvtColor(image_train, cv2.COLOR_BGR2GRAY)
            ret, bin_image = cv2.threshold(gray_image, 127, 255, cv2.THRESH_BINARY)

            if subdir[-2] == 'w':
                bin_image = 255 - bin_image

            values = []

            for i in range((90 - interest) // 2, (90 + interest) // 2):
                for j in range((90 - interest) // 2, (90 + interest) // 2):
                    values.append(bin_image[i, j])

            for i in range(0, d):
                x[row, i] = values[i]

            piece_name = subdir[-2] + subdir[-1]
            y[row, 0] = piece_to_class[piece_name]
            row = row + 1
```

We are currently in the branch that tells us to train the dataset, therefore the content of two files will have to be updated, one contains the priori of each piece and the other one contains the likelihood values. The likelihoods are calculated using the Laplace smoothing method, previously explained on the paper.

```
priori_file = open("train_priori.txt", "w")
        for row in priori:
            np.savetxt(priori_file, row)
        priori_file.close()

        likelihood = np.zeros((nr_classes, d), dtype=float)

        for c in range(0, d-1):
            for i in range(0, n-1):
                if x[i, c] == 255:
                    likelihood[y[i, 0], c] = likelihood[y[i, 0], c] + 1

        for i in range(0, nr_classes):
            for j in range(0, d-1):
                likelihood[i, j] = likelihood[i, j] + 1
                likelihood[i, j] = likelihood[i, j] / (count_folder_files(sys.path[1]
                + f'\\train\\{class_to_piece[i]}') + nr_classes)

        likelihood_file = open("train_likelihood.txt", "w")
        for row in likelihood:
            np.savetxt(likelihood_file, row)
        likelihood_file.close()
```

If we chose not to train the dataset and have a better computing speed, then we simply load the stored values into the likelihood and priori structures. The training process is followed by the test function, which requires the computed likelihood and priori vectors. The rest of the parameters wouldn't be needed if we worked with a class that uses them as features.

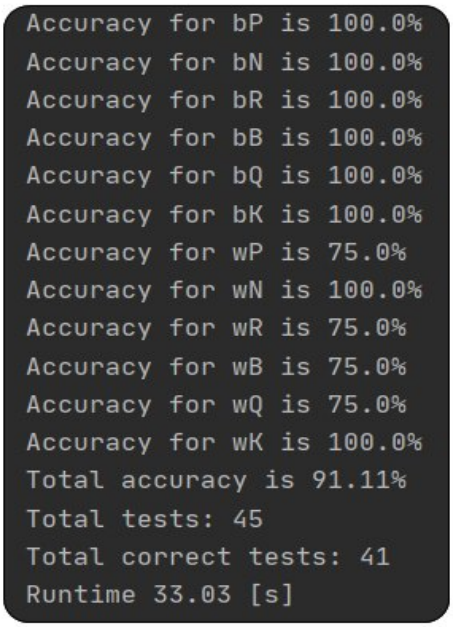
```
else:
    likelihood = np.loadtxt("train_likelihood.txt").reshape(nr_classes, d)
    priori = np.loadtxt("train_priori.txt").reshape(nr_classes, 1)

bayes_test(nr_classes, likelihood, priori, d, interest, debug)
```

The core part of the testing function is the part that calculates the probabilities for each class using the loaded likelihood and priori vectors. The candidate with the highest probability is chosen as the predicted class. Priori and likelihood values should not be 0 because log(0) is undefined.

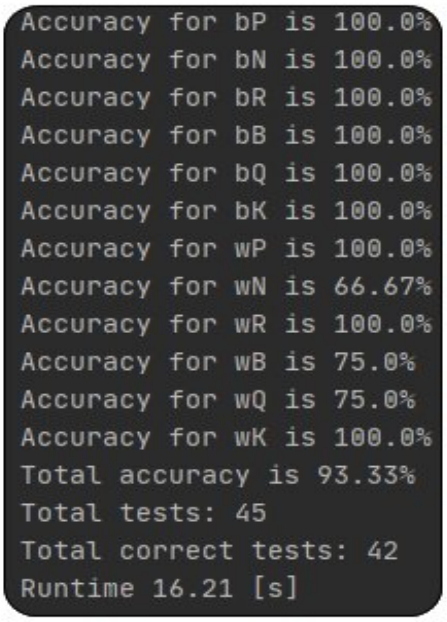
```
for c in range(0, nr_classes):
    probability = 0
    for i in range(0, d-1):
        if feature_vector[i] == 255:
            probability = probability + math.log(likelihood[c, i])
        else:
            probability = probability + math.log(1 - likelihood[c, i])
        if priori[c, 0] > 0:
            probability = probability + math.log(priori[c, 0])
        probabilities.append(probability)

max_class = 0
max_prob = -sys.maxsize

for i in range(0, nr_classes):
    if probabilities[i] > max_prob:
        max_class = i
        max_prob = probabilities[i]
```

Results

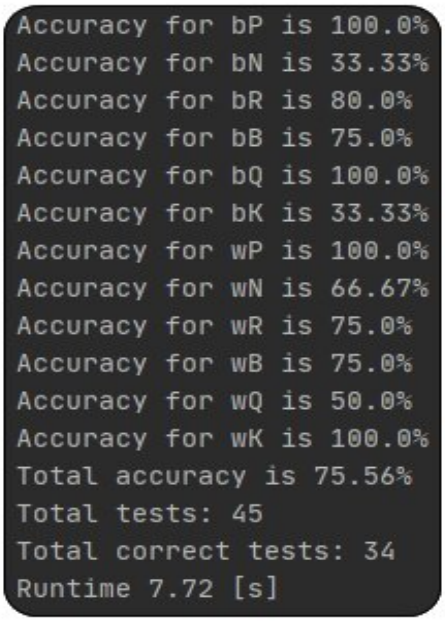
The following images represent results of the classifier after the training and testing operations are successfully executed. From the total of 45 tests, 41 are predicting the right class (for the default interest value of 90). For a 60 x 60 window the runtime of the classifier is about twice as fast as the first version and displays a similar accuracy, while the 45 x 45 window also halves the runtime, but doesn't classify the pieces as well as the previous scenarios. If the train variable is set to False (in order to skip the training operation) then the runtime improves even more.



(a) 90x90 interest



(b) 60x60 interest



(c) 45x45 interest

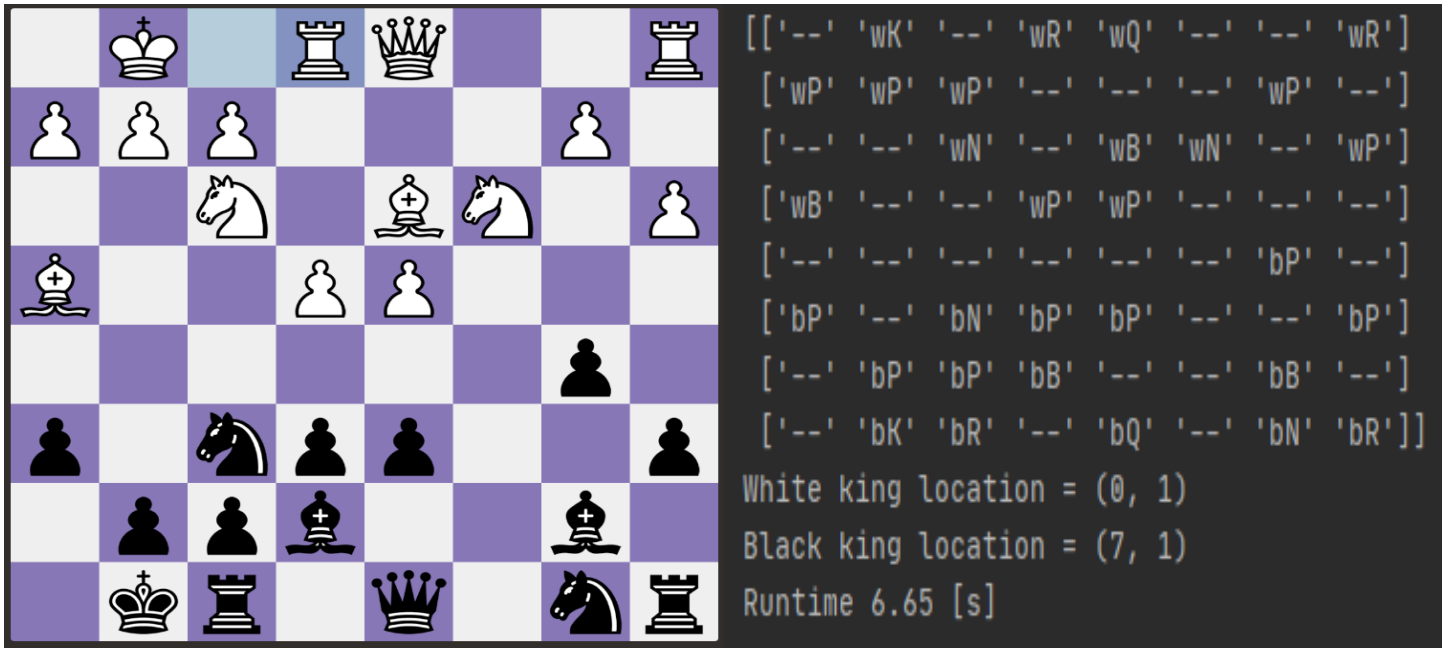
		Classifier Result											
		bP	bN	bR	bB	bQ	bK	wP	wN	wR	wB	wQ	wK
Test Piece	bP	5	0	0	0	0	0	0	0	0	0	0	0
	bN	0	3	0	0	0	0	0	0	0	0	0	0
	bR	0	0	5	0	0	0	0	0	0	0	0	0
	bB	0	0	0	4	0	0	0	0	0	0	0	0
	bQ	0	0	0	0	3	0	0	0	0	0	0	0
	bK	0	0	0	0	0	3	0	0	0	0	0	0
	wP	0	0	0	0	0	0	3	0	0	0	0	1
	wN	0	0	0	0	0	0	0	3	0	0	0	0
	wR	0	0	0	0	0	0	0	0	3	1	0	0
	wB	0	0	0	0	0	0	0	0	0	3	0	1
	wQ	0	0	0	0	0	0	0	0	0	0	3	1
	wK	0	0	0	0	0	0	0	0	0	0	0	3

The confusion matrix is a summary of prediction results on the classification problem. The diagonal elements represent the number of points for which the predicted label is equal to the true label, while off-diagonal elements are those that are mislabeled by the classifier. The higher the diagonal values of the confusion matrix the better, indicating many correct predictions.

There is also a debugging part implemented in the classifier, which displays all the probabilities for each class and also iterates image by image (the user is able to see the results step by step).

Figure 4.1: Black Bishop - Probabilities

For future development ideas, we can give a custom digital chess board as input to a function that splits the image into 64 different images, negates the images that represent empty squares and classifies each piece from the respective board. The result represents an 8x8 matrix with the board state, which could be given further as input to an AI that plays against itself from the starting position, or to a human user for learning chess tactics.

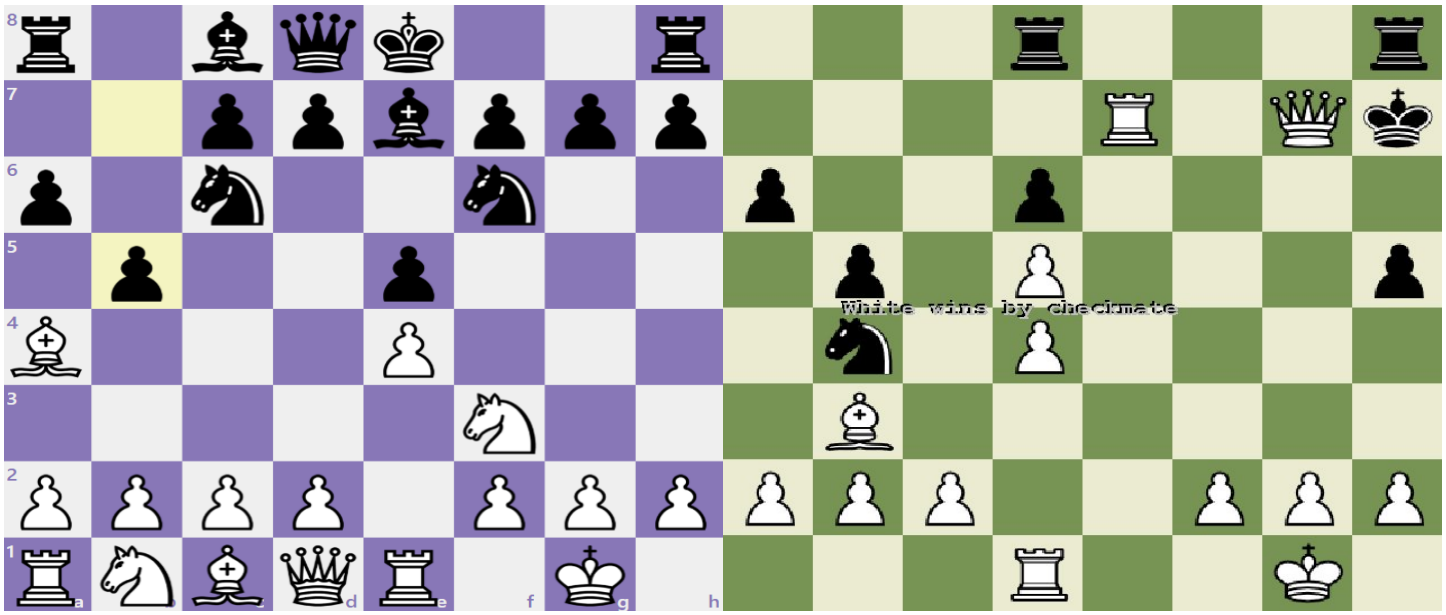


(a) Input board position

(b) Predicted board

Conclusions

Even though the training set size is reduced, the algorithm provided acceptable accuracy. Since the Naive Bayes classifier does not require any excessive training procedure commonly required in most of the artificial neural network architectures, the resulting classifier can yield an appropriate classification decision with very limited computational efforts. The training time advantage of this algorithm comes from the fact that Naive Bayes is not an iterative training algorithm.



(a) Input board position - Berlin Defense

(b) AI vs AI outcome - White Win

Bibliography

[1] http://www.isaet.org/images/extraimages/P1216004.pdf
Image Classification Using Naïve Bayes Classifier - Dong-Chul Park

[2] Naive Bayesian Classifier: Digit Recognition Application - TUCN Laboratory Activity

* * *