Neural Networks -Deep Learning - 1st Project (*Intermediate*)

Iordanis Kokkinidis - 3412

This is the Intermediary project for the Neural Networks - Deep Learning course in the Computer Science Department of the Aristotle University of Thessaloniki
In this project we are going to use SK Learn's Nearest Neighbors and Nearest Centroid classifiers, in order to classify the handwritten digits from the MNIST Dataset.
Below, we will talk a little bit about the data, and the classifiers used, while comparing their performance.
Going forward, our goal for the first project is to create a Neural Network from scratch (using Python and numPy) to recognize the handwritten digits of the MNIST Dataset and compare its performance to the three classifiers below.

MNIST dataset

The MNIST dataset of hand written digits is a dataset that consists of 70,000 grayscale images of 28×28 pixels, that represent a digit (from 0 to 9). 60,000 of these images are part of the training set, while 10,000 of these images are part of the test set. Each pixel has a value between 0 - 255 that represents the grayscale value of the pixel (with 0 being black and 255 being white). Therefore, each image is represented by a 28×28 matrix.

For simplicity's sake, we import the dataset from keras

```
from keras.datasets import mnist
```

Loading the data

We load the data from the `mnist.dataset`, using `mnist.load_data()` and store each NumPy array in the appropriate NumPy array. For example, we have `X_train` containing the images of the training set and `y_train` containing the labels of the training set.

We then reshape the `X_train` and `X_test` NumPy arrays from $(60000, 28, 28)$ and $(10000, 28, 28)$ to $(60000, (28 \times 28))$ and $(10000, (28 \times 28))$ respectively, basically transforming each image's 28×28 pixel matrix to a a vector.
We also keep the original dataset in the `X_train_og` and `X_test_og` NumPy arrays, in order to use them later when we want to plot the (28×28) images.

```
def load_data():
  (X_train, y_train), (X_test, y_test) = mnist.load_data()
  X_train_og = X_train
  X_test_og = X_test
  X_train = X_train.reshape(60000, (28*28))
  X_test = X_test.reshape(10000, (28*28))
  return X_train, y_train, X_test, y_test, X_train_og, X_test_og
```

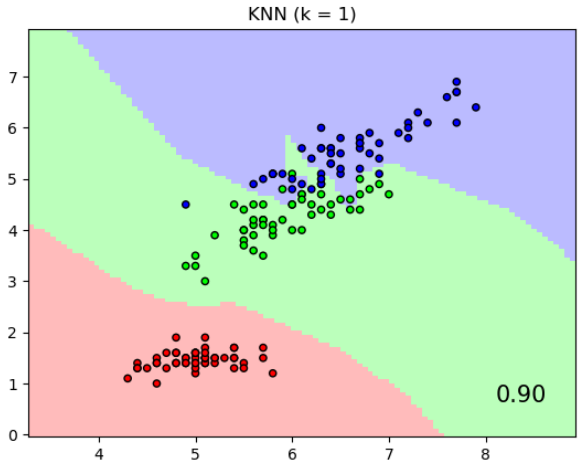
SKLearn's Classifiers

For this intermediate project, we are going to use two of SKLearn's machine learning classifier's in order to classify the digits from the MNIST dataset. More specifically, we are going to use the KNearestNeighbor Classifier and the NearestCentroid Classifier.

Nearest Neighbor Classification

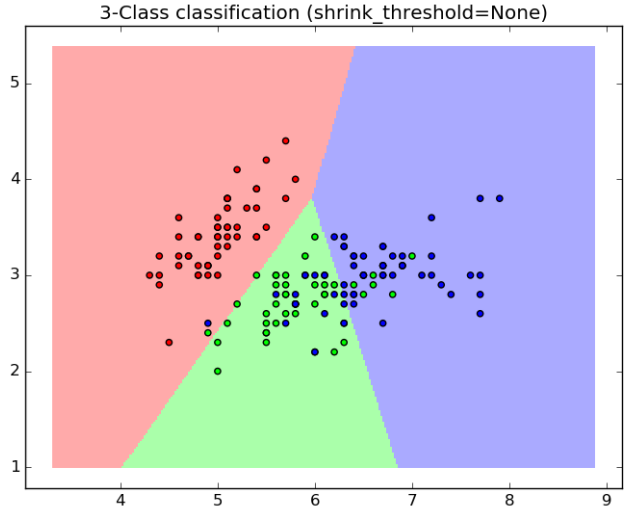
Neighbors-based classification is a type of *instance-based learning* or *non-generalizing learning*: it does not attempt to construct a general internal model, but simply stores instances of the training data. Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the nearest neighbors of the point.

The `KNeighborsClassifier` implements learning based on the k nearest neighbors of each query point, where k is an integer value specified by the user. In our case, we implement the classifier for $k = 1$ and $k = 3$.



Nearest Centroid Classification

The `NearestCentroid` classifier is a simple algorithm that represents each class by the centroid of its members. It also has no parameters to choose, making it a good baseline classifier. It does, however, suffer on non-convex classes, as well as when classes have drastically different variances, as equal variance in all dimensions is assumed.



source: scikit-learn.org

Implementing the Classifiers

K Nearest Neighbor Classifier

We define a function in which we define our knn classifier, using SKLearn's `KNeighbor` classifier and return said classifier. This function takes k as a parameter, which is the number of neighbors we want to use with our classifier.

```
def define_knn(k):
  knn = neighbors.KNeighborsClassifier(k)
  return knn
```

Nearest Centroid Classifier

In the same way as with the *K Nearest Neighbor Classifier*, we define our NearestCentroid classifier and return it.

```
def define_ncc():
  ncc = neighbors.NearestCentroid()
  return ncc
```

By using these two functions, we can create as many classifiers as we want in our driver program. For instance, below we will create two KNeighbors classifiers using the `define_knn(k)` function for a classifier wth $k = 1$ and a classifier with $k = 3$ and one Nearest Centroid classifier using the `define_ncc()` function.

Fitting the data in the Classifiers

This function is used by both classifiers (KNN and NearestCentroid) to fit the classifier's model according to the given training data (from SKLearn's documentation). The function takes as parameters a classifier (one of the two defined above), and the training set data. X is an array of the image vectors and y is an array of the labels. For both the KNN and NearestCentroid classifiers, we use their `fit(X, y)` method and then we return the classifier that is ready to make predictions on the test set.

```
def fit_data(classifier, X, y):
  classifier.fit(X, y)
  return classifier
```

Defining the Classifiers and getting predictions

Below we define three functions, each corresponding to one of the classifier's we want to test (***KNN*** *with* $k = 1$, ***KNN*** *with* $k = 3$ and ***NearestCentroid*** classifiers). Each of these functions takes **X_train X_test** and **y_train** arrays as parameters and behaves the same way. First, they define the appropriate classifier (*The first two also have a variable* k*, that is the* ***number of neighbors*** *we want to use in the* ***KNearestNeighbor classifier***) and then call on the `fit_data()` function giving the classifier and training set arrays (X_train, y_train) as parameters.

Then, we use the `time.process_time()` function to set a start_time variable (and subsequently an end_time) in order to time how long it takes for the classifier to make predictions on the test set of *10,000* images. For that, we use the `[classifier].predict()` function (where classifier is one of our classifiers, **knn** or **ncc**) that takes the **X_test** array of image vectors as a parameter (our test set) and performs classification on that given test data (predicts the label that the test set data have). The predictions that the classifier makes are returned and stored in the predictions array, in order to be returned to the main driver progam, along with the delta_time (time elapsed while the classifier was making predictions `[classifier].predict(X_test)`).

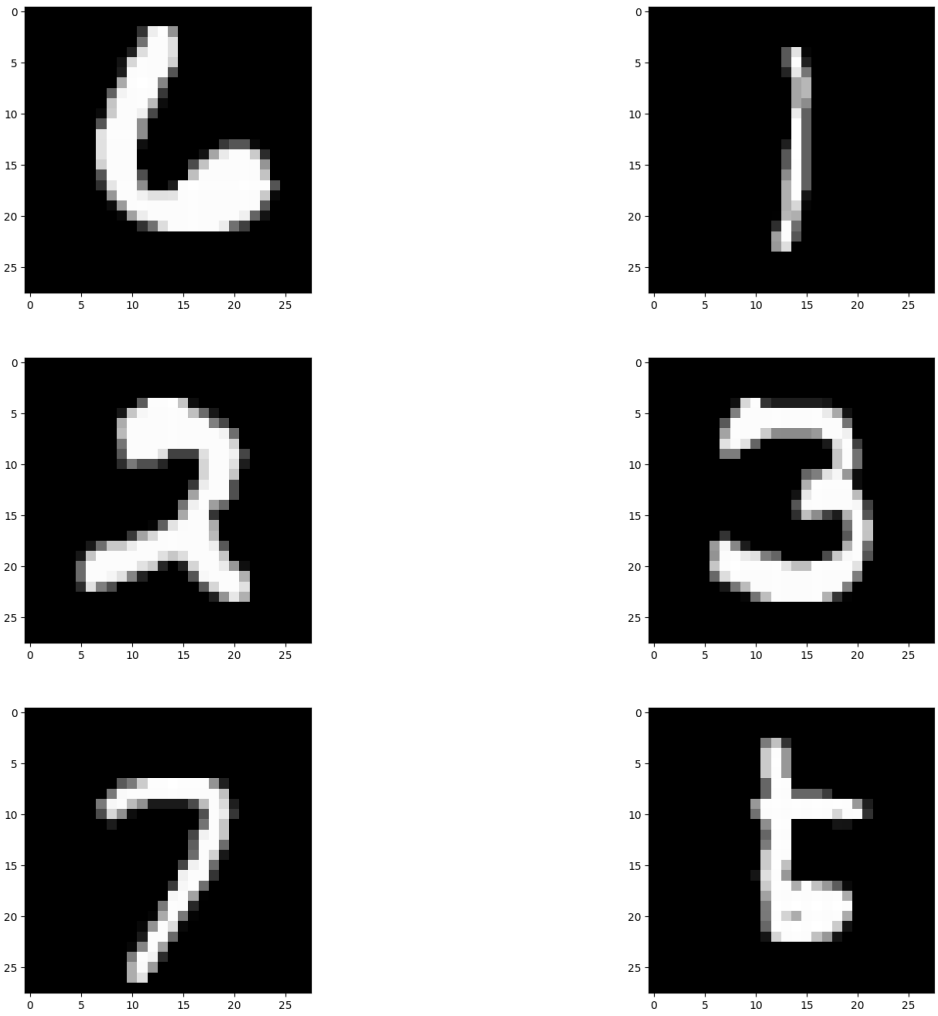
```
def knn1_predictions(X_train, X_test, y_train):
  k = 1
  knn = define_knn(k)
  fit_data(knn, X_train, y_train)
  start_time = time.process_time()
  predictions = knn.predict(X_test)
  end_time = time.process_time()
  delta_time = (end_time - start_time) * 1000
  delta_time = round(delta_time)
  print("Made predictions with KNN Classifier with k = 1")
  print("Time: " + str(delta_time) + "ms")
  return predictions, delta_time

def knn3_predictions(X_train, X_test, y_train):
  k = 3
  knn = define_knn(k)
  fit_data(knn, X_train, y_train)
  start_time = time.process_time()
  predictions = knn.predict(X_test)
  end_time = time.process_time()
  delta_time = (end_time - start_time) * 1000
  delta_time = round(delta_time)
```

```
  print("Made predictions with KNN Classifier with k = 3")
  print("Time: " + str(delta_time) + "ms")
  return predictions, delta_time

def ncc_predictions(X_train, X_test, y_train):
  ncc = define_ncc()
  fit_data(ncc, X_train, y_train)
  start_time = time.process_time()
  predictions = ncc.predict(X_test)
  end_time = time.process_time()
  delta_time = (end_time - start_time) * 1000
  delta_time = round(delta_time)
  print("Made predictions with the NCC Classifier.")
  print("Time: " + str(delta_time) + "ms")
  return predictions, delta_time
```

Showing Images of our data

*The function below is used to show an image of our data, given an **index** (which sample we want to show) and the **X** and **Y** NumPy arrays of the data set*. We must use the `X_train_og` and `X_test_og` arrays, since we kept them in dimensions of $(60000, (28 \times 28))$ and $(10000, (28 \times 28))$ respectively. We use the ***matplotlib library*** in order to plot the images of our data samples.

```
def image_show(index, X, Y):
  image = X[index]
  fig = plt.figure
  plt.imshow(image, cmap='gray')
  print("Showing sample data " + str(index) + " with label: " + str(Y[index]))
  plt.show()
```



Printing the predictions

We have defined a method below, that prints the predictions of a classifier. The function takes as parameters the predictions array, which is the predictions of a classifier we want to print, the X_test_og array, in order to plot the image of the samples if we want, the y_test which is the array that holds the labels of our dataset, and two boolean variables called print_wrong_predictions and print_correct_predictions, so that we can specify what kind of predictions (or both) we want to see. It also takes a number_of_predictions parameter, that specifies how many predictions we want to see. This number must be between 1 and 10,000, although setting it to a large number might not be the optimal choice.

In this function, we use a pseudo-random number generator to generate a number between 0 and 10,000, that is the index of the data sample we are going to print, in order to show a number of different predictions each time and not all of them, since that will take some time and the output will not be very readable.

When an index is generated, we check if the `predictions[index]` is equal to the `y[index]` in order to determine if the prediction was correct or not. This is not only done so that we can accompany the image of our data sample with the appropriate label (i.e., the prediction was CORRECT!), but also to allow the user to only print correct or wrong predictions if he wishes.

```
def print_predictions(predictions, X, y, print_wrong_predictions,
    print_correct_predictions, number_of_predictions_to_print):
  for i in range(0, number_of_predictions_to_print):
    index = random.randrange(0,10000)
    if((predictions[index] != y[index])):
      if(print_wrong_predictions):
        image_show(index,X, y)
        print("Classifier's prediction: " , predictions[index])
        print("The prediction was WRONG!")
        print("The correct label is: ", y_test[index])
        print("\n")
    else:
      if(print_correct_predictions):
        image_show(index,X, y)
        print("Classifier's prediction: " , predictions[index])
        print("The prediction was CORRECT!")
        print("\n")
```

Getting the wrong predictions

The function below is used to get the wrong predictions and the number of wrong predictions that a classifier has made. It takes as parameters the predictions of the classifier and the y (y_test) array, which holds the labels for our test data. We iterate through the predictions array `for i in range(1, 10000)` and check if the `predictions[i]` is not equal to the `y[i]`. If yes, then the `wrong_predictions_total` is incremented and that prediction is put in the wrong_predictions array. We finally return the `wrong_predictions` array and the `wrong_predictions_total`.

```
def get_wrong_predictions(predictions, y):
  wrong_predictions = []
  wrong_predictions_total = 0
  for i in range(1, 10000):
    if((predictions[i] != y[i])):
      wrong_predictions_total += 1
      wrong_predictions.append(i)
  return wrong_predictions, wrong_predictions_total
```

Getting a classifier's accuracy

This function is used to calculate the accuracy of a classifier's predictions. It simply takes as a parameter the `wrong_predictions` number of a classifier and return the percentage of correct predictions.
Since the test set contains 10,000 data samples, we divide the number of `wrong_predictions` by that and to get error. Then, we subtract that error from 1, to get the accuracy, that we multiply by a hundred to present it as a percentage (%).

```
def get_accuracy(wrong_predictions):
  return ((1-(wrong_predictions/10000)) * 100)
```

Driver Program

Below is the main part of the program that loads the data on the three classifiers and for each one gets the predictions, accuracy, time, wrong predictions and can even call a function to print the correct and/or wrong predictions

We want to test each classifier on the MNIST dataset.

We set the print_wrong_predictions and print_correct_predictions for each classifier block of code, depending on what prefferences we have on the output. For example if we want to see the wrong predictions of the NearestCentroid classifier, but not of the KNearestNeighbor classifiers.

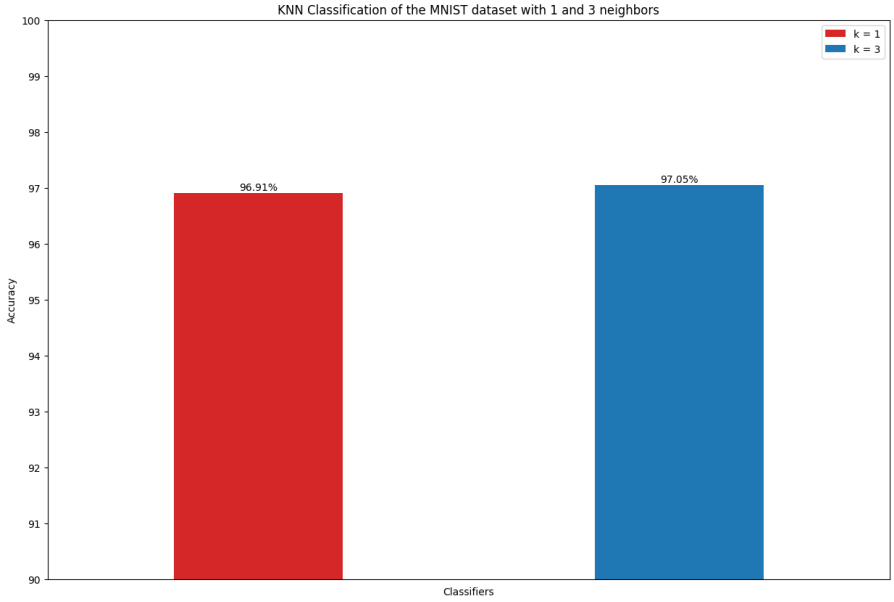
Then for each classifier, we get the predictions and the time it took to make them by using the appropriate function (those that were defined above). We also get the wrong predictions for each classifier, by using the `get_wrong_predictions()` funciton. Finally, we get the accuracy for each classifier, using the `get_accuracy()` method.
We have all the data we need from the classification process and we call on the `print_predictions()` method to print the predictions of each classifier, as explained above.

```
int_wrong_predictions = True
print_correct_predictions = True
number_of_predictions_to_print = 50

predictions, time1 = knn1_predictions(X_train, X_test, y_train)
wrong_predictions, wrong_predictions_total = get_wrong_predictions(predictions, y_test)
print(str(wrong_predictions_total) + " wrong predictions out of 10,000 predictions")
accuracy = get_accuracy(wrong_predictions_total)
print("Accuracy: " + str(accuracy) + "%\n")
print("Printing prediction's of KNearest Neighbor Classifier with k = 1")
print_predictions(predictions, X_test_og, y_test, print_wrong_predictions, print_correct_predictions, number_of_predictions_to_print)
print("\n")

print_wrong_predictions = True
print_correct_predictions = True
number_of_predictions_to_print = 50

predictions2, time2 = knn3_predictions(X_train, X_test, y_train)
wrong_predictions2, wrong_predictions_total2 = get_wrong_predictions(predictions2, y_test)
print(str(wrong_predictions_total2) + " wrong predictions out of 10,000 predictions")
accuracy2 = get_accuracy(wrong_predictions_total2)
print("Accuracy: " + str(accuracy2) + "%\n")
print("Printing prediction's of KNearest Neighbor Classifier with k = 3")
print_predictions(predictions2, X_test_og, y_test, print_wrong_predictions, print_correct_predictions, number_of_predictions_to_print)
print("\n")

print_wrong_predictions = True
print_correct_predictions = True
number_of_predictions_to_print = 50

predictions3, time3 = ncc_predictions(X_train, X_test, y_train)
wrong_predictions3, wrong_predictions_total3 = get_wrong_predictions(predictions3, y_test)
print(str(wrong_predictions_total3) + " wrong predictions out of 10,000 predictions")
accuracy3 = get_accuracy(wrong_predictions_total3)
print("Accuracy: " + str(accuracy3) + "%\n")
print("Printing prediction's of Nearest Centroid Classifier")
print_predictions(predictions3, X_test_og, y_test, print_wrong_predictions, print_correct_predictions, number_of_predictions_to_print)
print("\n")
```

Results

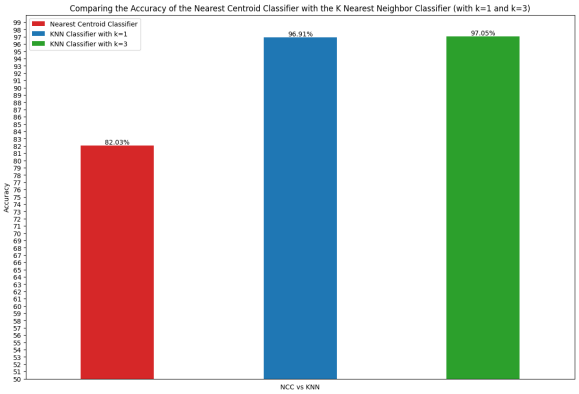
Comparing K Nearest Neighbor classifier with k=1 and k=3

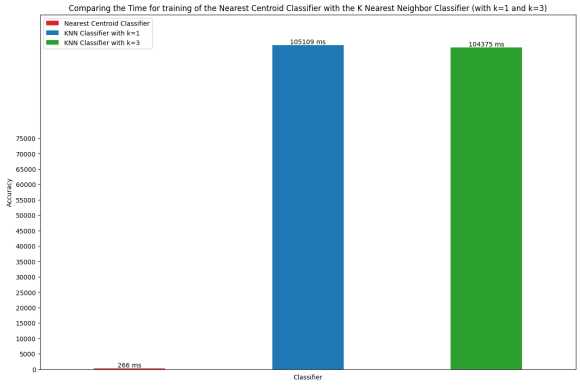
For our dataset, both classifiers performed very well, with an accuracy on the test set of 96.91% and 97.05% respectively. They were also very close on the time it took them to make their predictions, as we will see below.



Comparing the Nearest Centroid Classifier with the K Nearest Neighbor classifiers

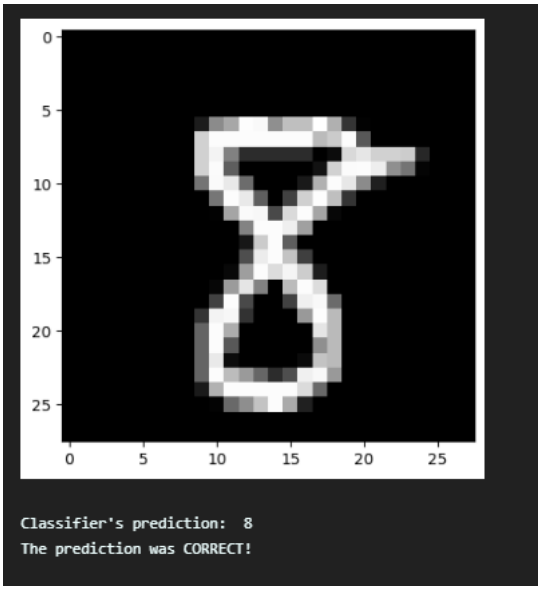
If we add the Nearest Centroid Classifier `ncc` to our comparisons, we will see that it doesn't fare well in terms of accuracy, when compared with the other two. Even though the time it took for the ncc classifier to produce it's predictions was almost instant, its accuracy was a "mere" 82.03%.

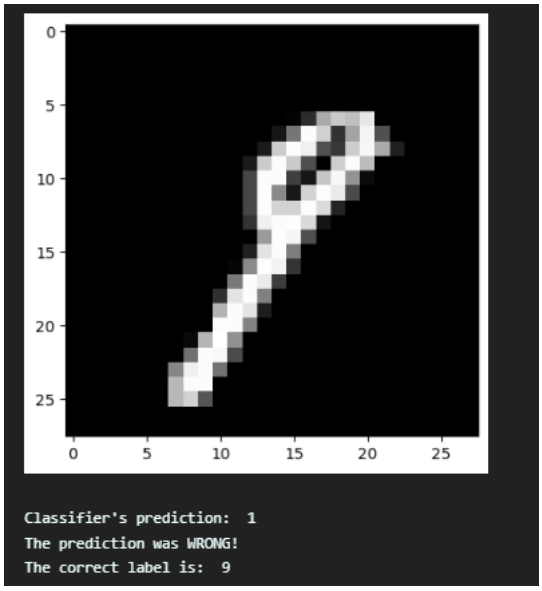




Some of the classifiers' predictions

```
Made predictions with KNN Classifier with k = 1
Time: 100328ms
309 wrong predictions out of 10,000 predictions
Accuracy: 96.91%



Made predictions with KNN Classifier with k = 3
Time: 102625ms
295 wrong predictions out of 10,000 predictions
Accuracy: 97.05%



Made predictions with the NCC Classifier.
Time: 203ms
1797 wrong predictions out of 10,000 predictions
Accuracy: 82.03%
```





It will be interesting to see how those two classifiers compare to the neural network we are going ot build next. Of course, a neural network will take more time to train, but we are hoping that the predictions will be more accurate, especially when compared to the Nearest Centroid classifier.