Pattern Recognition - Lab1

Authors: Thodoris Lymperopoulos, Giorgos Nanos

October 2021

Steps of the Process

Step 1

In the first step, in order to load the dataset, we write a function called *load_data()*. The datasets (training and test set) happen to be in txt files, and by using the numpy.loadtxt() method, provided by the numpy library, he can handle them as numpy arrays. We used numpy library since it provides an easy handling of tabular data. Another alternative is to use the pandas library, which simplifies the code even more, but we found the use of numpy to be more consistent and the code to be more organized.

Our function takes no arguments, since it would be inconvenient to pass the paths of the files, which are considered already known. After loading the train and test data with the use of numpy.loadtxt(), which takes as an argument the path of the file, we split the data into four numpy arrays: X_train, X_test, y_train, y_test. The values of X_train and X_test are the grayscale values of the corresponding pixels, for the training and test set accordingly. Thus, a row of the X_train, indexed as i, represents the values of the i-th image of the training set, containing 256 values. The function then prints some relevant information and returns the four arrays.

We call our function as below:

```
In [11]: X_train, y_train, X_test, y_test = load_data()

The size of the training set is 7291, with 256 features
The size of test set is 2007
```

We can have a glance at our data:

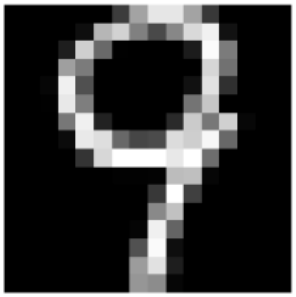
```
In [18]: print(f"X_train: \n {X_train} \n y_train: \n {y_train}")

X_train:
 [[-1.    -1.    -1.    ... -1.    -1.    -1.   ]
 [-1.    -1.    -1.    ... -0.671 -0.828 -1.   ]
 [-1.    -1.    -1.    ... -1.    -1.    -1.   ]
 ...
 [-1.    -1.    -1.    ... -1.    -1.    -1.   ]
 [-1.    -1.    -1.    ... -1.    -1.    -1.   ]
 [-1.    -1.    -1.    ... -1.    -1.    -1.   ]]
 y_train:
 [6. 5. 4. ... 3. 0. 1.]
```

Step 2

In order to plot any digit from the X_train array, we define a function, called *show_sample()*, that takes as an argument a numpy array and an integer, corresponding to the index of that array. The row of that index is first reshaped, with the use of np.reshape() function that takes as an argument the array to be reshaped and its new dimensions. Since images are 16x16, the second argument of the method is (16, 16). We then show the image with the help of plt.imshow(), with no axis (no useful information) and in greyscale, as asked.

We then call our function, with the index asked; 131.

```
In [22]: show_sample(X_train, 131)
```



Step 3

Our function *plot_digit_samples()* takes two arguments; two np arrays. For the first array, its rows correspond to grayscale images of digits. An element of the second array is the label-digit for the corresponding row of the first array. As an output, it plots 10 images in two rows with 5 elements each, for a better view.

Out function plots a digit for every number-category. First, we find the indices of the rows that correspond to each number. For example, for number 0, we find all rows that the respective images depict a 0. We do this once, for all digits and rows. Then we take a random element from each list with indices, and plot the respective image. This is achieved with the use of figure.add_subplot() which allows us to add as many plots in a figure as we want. An iterator places a new plot in the right position.

We then call our function with X_train and y_train arguments. We have to exclude test set, since it is considered unknown.

```
In [30]: plot_digits_samples(X_train,y_train)
```

A second call with the same arguments will (most probably) plot different images of digits, since the choice of the indices is random.

```
In [31]: plot_digits_samples(X_train,y_train)
```

Steps 4 and 5

Our function *digit_mean_at_pixel()* calculates the mean value of a pixel for a given digit. First, we find all indices of rows that depict that digit. Then, we create a vector that collects all values of that particular pixel. In order to find the pixel in the vectorized images in X, we have to multiply the coordinate with 16 (since each row has 16 elements) and add the second coordinate. np.take() is a "fancier" slicing that helps us take the value of the given pixel, for the indices we found to represent the given digit. After that, np.mean() gives the result.

We call our function as below;

```
In [13]: digit_mean_at_pixel(X_train, y_train, 0, (10,10))

-0.5041884422110553
```

For the calculation of the variance, we define another function *digit_variance_at_pixel()*. The preprocessing is the same, the only thing that changes is the calculation at the end of the function. We use np.var() instead of np.mean().

We get the following results;

```
In [35]: digit_variance_at_pixel(X_train, y_train, 0, (10,10))

0.5245221428814929
```

Step 6

In order to calculate the mean value and variance for every digit of the images depicting zero, we define two functions *digit_mean(), digit_variance()*. For a particular digit, they find all rows of the matrix depicting that digit and apply the np.mean() and np.var() correspondingly, to get and return the mean value and variance for a digit.

In order to calculate these values for 0, we make the following calls;

```
In [104]: digit_mean(X_train, y_train, 0)[0, :20]

Out[104]: array([-0.99862814, -0.99539782, -0.98492295, -0.94125126, -0.83334255,
                 -0.57142295, -0.13158459,  0.15260804,  0.04628392, -0.35370101,
                 -0.74124874, -0.92091625, -0.98502513, -0.99718258, -0.99993467,
                 -1.        , -0.99823451, -0.99346566, -0.94997069, -0.81149414])
```

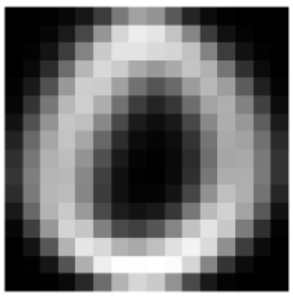
```
In [23]: digit_variance(X_train, y_train, 0)[0, :20]

Out[23]: array([2.24522353e-03, 6.60664324e-03, 1.95906222e-02, 6.29011999e-02,
                1.80366523e-01, 3.72113773e-01, 5.31683397e-01, 5.38201451e-01,
                5.62586277e-01, 4.57620803e-01, 2.35928060e-01, 8.26775407e-02,
                1.21866074e-02, 2.34500854e-03, 2.24866375e-06, 0.00000000e+00,
                2.64368538e-03, 6.10332085e-03, 5.44480636e-02, 2.28699335e-01])
```

The dimensions of the two vectors are quite large (256,), so they couldn't fit in a single snip. We only project the first 20 values.

Step 7

In order to draw a zero calculated by the mean values of the zero digits, we call predefined functions, as below;

```
In [40]: show_sample(digit_mean(X_train, y_train, 0), 0)
```

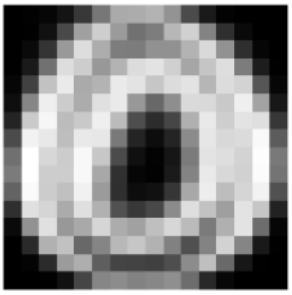


We calculate the digit's mean values with the use of *digit_mean()* an call the *show_sample()* function for these values.

Step 8

To draw a zero calculated by the variance of the zero digits, the procedure is very similar. We calculate the digit's variance with the use of *digit_variance()* an call the *show_sample()* function for these values.

```
In [24]: show_sample(digit_variance(X_train, y_train, 0), 0)
Out[24]: <matplotlib.image.AxesImage at 0x209f02844f0>
```



We can see that the two plots differ a lot. This is because some zeros in our dataset are slightly larger than others, while some other have a smaller radius. Most of them are similar to the mean zero and thus we can see in that figure that while its borders are blurry and more black, the interior is quite clear. On the other hand, in the image produced by the variance zero, see observe the opposite result. The borders are clearer - because of the slighly larger and smaller zeros, the variance there is quite large - but the interior is blurry - because most of the zeros are placed there, the variance there is small -. We can say that the two results are supplementary and opposite the one to another.

Step 9

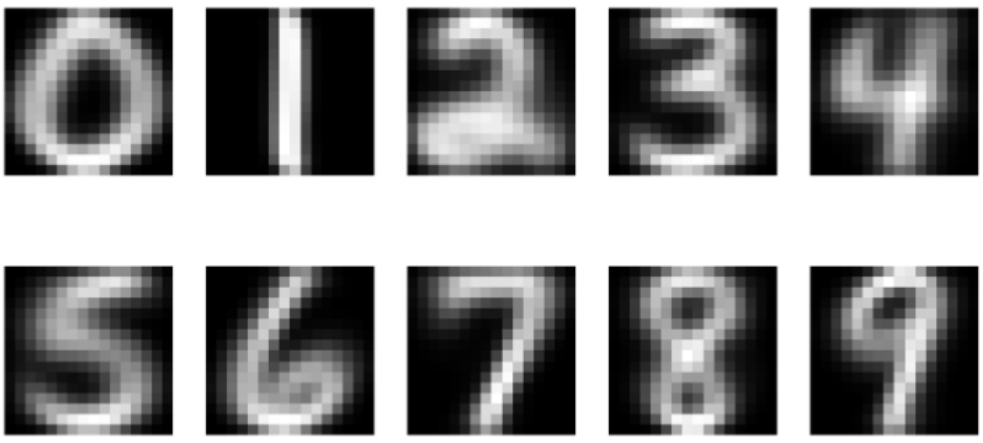
In order to calculate the mean values and variances for all digits, we define the *mean_and_var_for_all_digits()* function, where we iterate from 0 to 9 and call the predefined functions *digit_mean(), digit_var()* and then reuse some code (*plot_digits_samples()*) to plot them.

We then calculate the mean values and variances for each digit. A small peak at the results is below.

```
In [33]: mean, var = mean_and_var_for_all_digits(X_train, y_train, plot
         print(mean[0, :20], var[0, :20])

         [-0.99862814 -0.99539782 -0.98492295 -0.94125126 -0.83334255 -(
          -0.13158459  0.15260804  0.04628392 -0.35370101 -0.74124874 -(
          -0.98502513 -0.99718258 -0.99993467 -1.         -0.99823451 -(
```

While plotting the mean values of the digits, we get the following results;

```
In [47]: mean_value_for_all_digits(X_train, y_train)
```



Step 10

First we define a function *euclidean_distance* to calculate the euclidean distance between two vectors. It simply returns the norm of their difference, by using the np.linalg.norm() function. Then, we define a function, the *euclidean_distance_classifier()* that classifies an array of a digit to one of the classes 0-9, based on its euclidean distance with the mean values of every digit. It is classified to the class that the euclidean distance is minimized.

For the row 101, we get the following result.

```
In [76]: mean = mean_value_for_all_digits(X_train, y_train, plot=False, return_mean=True)
         euclidean_distance_classifier(X_train[101, :], mean)

Out[76]: 0
```

It is the correct prediction as we can verify below;

```
In [77]: print(y_train[101])

         0.0
```

Step 11

In order to classify all the data of the test set, we define a new function, called *euclidean_classifier_predictions()*. After calculating the distance from the mean value of every digit, using the *euclidean_distance_classifier()*, the algorithm returns that index with the smaller value.

We get the following results;

```
In [43]: mean = mean_and_var_for_all_digits(X_train, y_train, plot=False, return_values=True)[0]
         euclidean_classifier_predictions(X_test, mean)

Out[43]: array([9, 2, 3, ..., 4, 0, 1], dtype=int64)
```

We then define another function for calculating the score of our classifier. The function *calculate_euclidean_classifier_accuracy()* calls the euclidean classifier to get the predictions, and then counts the number of correct classifications. Only four decimals are shown, since the number quite large (with a lot of decimals).

```
In [62]: calculate_euclidean_classifier_accuracy(X_test, y_test, mean_and_var_for_all_digits
                                                 (X_test, y_test, plot=False, return_values=True)[0])

         Accuracy of the euclidean classifier is 81.4150 %
```

Step 12

In order to implement the Euclidean Classifier as a sklearn estimator we have to define three methods of the class (exceplt the __init__ method); fit(), predict(), and score(). After defining many functions before, we can use them to hepl us for this task.

The __init__ method initializes an empty X_mean array (initially valued as None), The fit method calculates the values of the array, which are the mean values of the digits. This variable is then used in predict() method to predict the classification of a new digit, with the help of *euclidean_distance_classifier()* function. The score method calculates the accuracy of the estimator on a whole test set, by calling the predict method to find the predictions, and count the correct ones, by comparing the predictions with the true labels.

Step 13

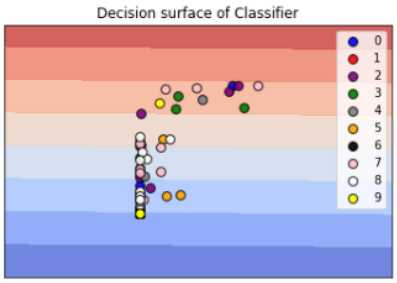
After defining a function that calculates the 5-cross-fold validation score for the euclidean classifier, we call it and get the following score;

```
#Evaluate the classifier

def evaluate_classifier(clf, X, y_true):
    return clf.score(X, y_true)

# Create the classifier
clf = EuclideanDistanceClassifier()
scores = cross_val_score(clf, X_train, y_train,
                         cv=KFold(n_splits=5),
                         scoring=evaluate_classifier)
print("CV error = %f +- %f" % (np.mean(scores)*100, np.std(scores)*100))
```

```
CV error = 84.656143 +- 0.854206
```

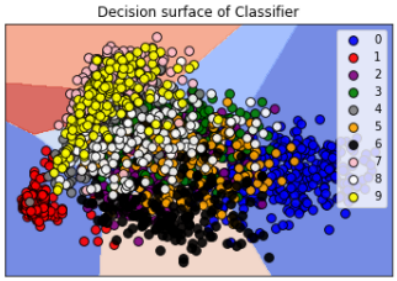
Then, we try a simple classification of the digits based on the first two components of the data of the test set. The decision surfaces that we consider to be strait lines (y=k, for k in 0, 1, ..., 9) make wrong classifications since valuable information is missing. We could experiment with different combinations of 2 components of the test data, which might produce slightly better or worse results, but it is obvious that we need a method that reduces the dimensionality of the data and, thus, exploits all the information provided.

```
clf = EuclideanDistanceClassifier()

plot_clf(clf, X_test, y_test, [i for i in range(10)] )
```



As a dimensionality reduction method, we applied the PCA method.

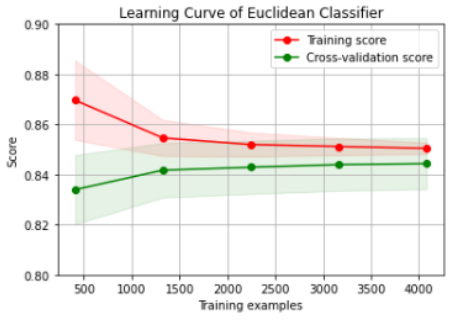
```
clf = EuclideanDistanceClassifier()
clf.fit(X_train_2d, y_train)

plot_clf(clf, X_test_2d, y_test, [i for i in range(10)] )
```



We can see that there are only 5 decision regions, since some digits were more frequent in the dataset than others. Low-frequency digits might also be sparce and overshadowed by the larger -and thus more dense- clusters of points-digits.

After that, with the help of sklearn we plot the learning curve. We see that the training score is bigger that the cross-validation score for a small dataset (as expected, with such a small dataset, all the information is needed for a good prediction).

```
title = "Learning Curve of Euclidean Classifier"
cv = ShuffleSplit(n_splits=100, test_size=0.2, random_state=0)
estimator = EuclideanDistanceClassifier()
plot_learning_curve(estimator, title, X_train, y_train, (0.8, 0.9), cv=cv, n_jobs=-1)

plt.show()
```



Step 14

In order to compute the a-priori probabilities of each class we use the function:

$$a - priori(i) = \frac{N(i)}{N}$$

where, N(i) is the number that the digit-i appears, and N is the training size. We define a function called *calculate_priors()* to calculate these values, with the use of np.unique(), passing as an argument the return_counts=True. To make a quick test we can sum up the probabilities calculated and we will see that the total sum is equal to 1.

The results are shown below.

```
apriori = calculate_priors(X_train, y_train)
for k, v in apriori.items():
    print("Digit " + str(k) + ": " + str(v))
```

```
Digit 0: 0.16376354409546015
Digit 1: 0.13784117405019888
Digit 2: 0.10026059525442327
Digit 3: 0.09024825126868742
Digit 4: 0.08942531888629818
Digit 5: 0.07625840076807022
Digit 6: 0.09107118365107666
Digit 7: 0.08846523110684405
Digit 8: 0.07433822520916199
Digit 9: 0.08832807570977919
```

A verification of the sum of the values;

```
print("Sum of a-priori is : " + str(sum(apriori.values())))
```

```
Sum of a-priori is : 1.0
```

Step 15

The implementation of the Naive Bayes classifier as a sklearn estimator demands the definition of three functions (instread of init); fit, predict, score. The fit method calculates all the self variables that are needed for the calculation of the multivariable density of the Gaussian Distribution. Then, the predict method calculates this function for a particular datum and the score method calculates the overall accuracy of the model. For every digit, we calculate a vector with the mean values and another with the variances. By assembling all these vectors, we create two np.arrays that are extremely useful for the calculations. The covariance matrix is diagonal (for all digits), since we assume that pixels are independent from one another. It is worth noting that some digits had zero variance in some pixels. This is problematic, since the derivative of the covariance matrix will be zero (zero division Exception). Thus, we replace zeros at this np.array with a small value, called smooth.

For a random value for smooth, we show some predictions of our model.

```
gaussNB = CustomNBClassifier()
gaussNB.fit(X_train, y_train, smooth=0.8)
```

```
]: CustomNBClassifier()
```

```
print("Predictions on the first 5 elements of the X_test")
for i in range(5):
    print(f"The prediction of our model is {gaussNB.predict(X_test[i])} while the true label is {y_test[i]}")
```

```
Predictions on the first 5 elements of the X_test
The prediction of our model is 9 while the true label is 9.0
The prediction of our model is 6 while the true label is 6.0
The prediction of our model is 3 while the true label is 3.0
The prediction of our model is 0 while the true label is 6.0
The prediction of our model is 6 while the true label is 6.0
```

Then, we try different values for smooth on a validation set to estimate which value is better. We do not try very small values since the determinant will be infinitely small. The best choice was found to be for 0.75. We calculate then the score of our classifier for that value, and the 5-fold-cross-validation.

```
#Define the classifier
gaussNB = CustomNBClassifier()
smooth = [0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]
new_X_train, val_set, new_y_train, y_val = train_test_split(X_train, y_train, test_size=0.33, random_state=42)
for sm in smooth:
    gaussNB.fit(new_X_train, new_y_train, sm)
    print(f"The score for smooth value equal to {sm} is {gaussNB.score(val_set, y_val)}")
```

```
The score for smooth value equal to 0.5 is 0.8122143747403406
The score for smooth value equal to 0.75 is 0.813876194432904
The score for smooth value equal to 1 is 0.8130452845866224
The score for smooth value equal to 1.25 is 0.8130452845866224
The score for smooth value equal to 1.5 is 0.8134607395097632
The score for smooth value equal to 1.75 is 0.8126298296634815
The score for smooth value equal to 2 is 0.8134607395097632
```

```
gaussNB = CustomNBClassifier()
gaussNB.fit(X_train, y_train, 0.75)
print(f"The score of our classifier for the test set is {gaussNB.score(X_test, y_test)}")
```

```
The score of our classifier for the test set is 0.7807673143996013
```

```
evaluate_custom_nb_classifier(X_train, y_train, use_unit_variance=False)
```

```
CV error = 82.046002 +- 2.079542
```

Meanwhile the score of the Gaussian estimator of sklearn was slightly lower. The cause might be the optimization that we did in the parameters of the classifier.

```
#Define the sklearn classifier (default smoothing is 1e-9)
gaussNB = GaussianNB()

gaussNB.fit(X_train, y_train)

print("Accuracy of sklearn GaussianNB classifier")
print()

print(gaussNB.score(X_test, y_test))
```

```
Accuracy of sklearn GaussianNB classifier

0.7194818136522172
```

Step 16

The implementation of our classifier allows us to create a new object of that class with its variance array have all values be equal to one. The 5-fold-cross-validation score is bigger than before. We see that the assumption that all pixels are independent from one another leads to lower performance. Even a uniform value for the covariances results in a better model.

```
evaluate_custom_nb_classifier(X_train, y_train, use_unit_variance=True)
```

```
CV error = 84.830591 +- 0.232547
```

Step 17

In the next step we are asked to compare the performance of Naive Bayes, Nearest Neighbors, SVM by using different kernels. We firstly, defined the Nearest Neighbors classifier using and performed the evaluation for 3,6 and 9 neighbors. The results of the accuracy achieved is:

```
# Define Nearest Neighbors classifier using 3,6  and 9 neighbors

neigh3 = KNeighborsClassifier(n_neighbors=3)
neigh3.fit(X_train, y_train)

neigh6 = KNeighborsClassifier(n_neighbors=6)
neigh6.fit(X_train, y_train)

neigh9 = KNeighborsClassifier(n_neighbors=9)
neigh9.fit(X_train, y_train)

print("Accuracy of Nearest Neighbors using different k neighbors")
print()

print("Neighbors = 3: " + str(neigh3.score(X_test, y_test)))
print("Neighbors = 6: " + str(neigh6.score(X_test, y_test)))
print("Neighbors = 9: " + str(neigh9.score(X_test, y_test)))
```

```
Accuracy of Nearest Neighbors using different k neighbors

Neighbors = 3: 0.9446935724962631
Neighbors = 6: 0.9387144992526159
Neighbors = 9: 0.9372197309417041
```

Then we defined the SVM Classifier using the linear, poly, rbf and sigmoid kernels.

```
# Define SVM Classifier using different kernels

svm_lin = SVC(kernel="linear", probability=True)
svm_lin.fit(X_train, y_train)

svm_poly = SVC(kernel="poly", probability=True)
svm_poly.fit(X_train, y_train)

svm_rbf = SVC(kernel="rbf", probability=True)
svm_rbf.fit(X_train, y_train)

svm_sigmoid = SVC(kernel="sigmoid", probability=True)
svm_sigmoid.fit(X_train, y_train)

print("Accuracy of SVM using different kernels")
print()

print("Kernel = linear: " + str(svm_lin.score(X_test, y_test)))
print("Kernel = poly: " + str(svm_poly.score(X_test, y_test)))
print("Kernel = rbf: " + str(svm_rbf.score(X_test, y_test)))
print("Kernel = sigmoid: " + str(svm_sigmoid.score(X_test, y_test)))
```

```
Accuracy of SVM using different kernels

Kernel = linear: 0.9262580966616841
Kernel = poly: 0.953662182361734
Kernel = rbf: 0.9471848530144494
Kernel = sigmoid: 0.8505231689088192
```

We can see that all Nearest Neighbors implementations along with the poly SVM have approximately 93.8-95.3% accuracy. As we expected, the Naive Bayes classifiers perform worse than the other classifiers, due to the simplifications made in the algorithm.

Step 18

In this step we are asked to combine some of the different classifiers performed in Step 17 to achieve higher accuracy. We can combine different classifiers with the ensembling technique. In this technique it is important that the combined classifiers misclassify different classes. So at first we take a quick look and check which digits each classifier misplaces and misclassifies.

```
The number of digits mispredicted and not correctly classified for the above classifiers:

neigh3
[ 4.  6. 15. 13. 17. 16.  7.  9. 15.  9.]

neigh6
[ 5.  6. 16. 11. 17. 18.  8.  9. 23. 10.]

neigh9
[ 5.  5. 18. 12. 21. 15.  9. 10. 22.  9.]

svm_lin
[ 8.  8. 17. 20. 20. 24. 10. 12. 22.  7.]

svm_poly
[ 4.  7. 13. 15. 11. 10.  8.  8. 10.  7.]

svm_rbf
[ 4. 10. 14. 19. 12.  9. 11.  9. 11.  7.]

svm_sigmoid
[35.  9. 53. 32. 26. 48. 23. 18. 40. 16.]

gaussNB_np
[62.  5. 53. 35. 50. 38. 27. 30. 39. 37.]

Most 3 mispredicted digits:

neigh3: 4 5 8
neigh6: 8 5 4
neigh9: 8 4 2
svm_lin: 5 8 4
svm_poly: 3 2 4
svm_rbf: 3 2 4
svm_sigmoid: 2 5 8
gaussNB_np: 0 2 4

Least 3 mispredicted digits:

neigh3: 0 1 6
neigh6: 0 1 6
neigh9: 0 1 6
svm_lin: 9 0 1
svm_poly: 0 1 9
svm_rbf: 0 9 5
svm_sigmoid: 1 9 7
gaussNB_np: 1 6 7
```

We can see that as expected the Naive Bayes is the worst classifier along with the sigmoid kernel of th SVM classifier. To have a better look we check the digits that are mostly misclassified in each classifier (second half-image above).

By taking into consideration the above results and after making a few test we conclude that the combination svm_poly + neigh3 + svm_lin seems like a good choice.

```
# svm_poly && neigh3 && svm_lin
clf6 = VotingClassifier(estimators = [('svm_poly', svm_poly), ('neigh6', neigh6), ('svm_lin', svm_lin)] ,voting = 'hard')
clf6.fit(X_train, y_train)
print("Hard : " + str(clf6.score(X_test, y_test)))
```

```
Hard : 0.9481813652217239
```

```
# svm_poly && neigh3 && svm_lin
clf7 = VotingClassifier(estimators = [('svm_poly', svm_poly), ('neigh3', neigh3), ('svm_lin', svm_lin)] ,voting = 'soft')
clf7.fit(X_train, y_train)
print("Soft : " + str(clf7.score(X_test, y_test)))
```

```
Soft : 0.9531639262580966
```

Next, we chose the svm "poly" classifier and used the BaggingClassifier to create an ensemble. The bagging technique is used to split the train set in random subsets and fit a classifer in each set.

```
bagging_3 = BaggingClassifier(svm_poly, n_estimators=3)
bagging_6 = BaggingClassifier(svm_poly, n_estimators=6)
bagging_9 = BaggingClassifier(svm_poly, n_estimators=9)


bagging_3.fit(X_train, y_train)
bagging_6.fit(X_train, y_train)
bagging_9.fit(X_train, y_train)

print("Bagging svm_poly using 3 estimators: " + str(bagging_3.score(X_test, y_test)))
print("Bagging svm_poly using 6 estimators: " + str(bagging_6.score(X_test, y_test)))
print("Bagging svm_poly using 9 estimators: " + str(bagging_9.score(X_test, y_test)))
```

```
Bagging svm_poly using 3 estimators: 0.9501743896362731
Bagging svm_poly using 6 estimators: 0.9531639262580966
Bagging svm_poly using 9 estimators: 0.9526656701544594
```

Generally the Bagging technique reduces the variance and thus the overfitting by averaging the results and voting. We see that it has increased the accuracy of our model.

Step 19

In the last step, we build a Neural Network with the help of Pytorch. At first we create a *DigitData* class, which is necessary for the dataloader. The dataloader is responsible for the data reading and the splitting the data into batches. Thus, the *DigitData* class, feeds the dataloader with any information it needs. Then we define fully connected Neural Networks for different number and sizes of layers and activation functions. We use the ReLU, the tanh and the sigmoid activation function for two and three layer Neural Networks. Then we perform the training of our Neural Network.

We experiment with the number of Hidden Layers (1 or 2) the Activation Type (ReLU, tanh or sigmoid) and the Size of each layer. The results that we took into consideration were the th Epochs the Accuracy (train) and the Accuracy (val).

Hidden Layers	Activation Type	Size of each layer	Epochs	Accuracy (train)	Accuracy (val)
1	ReLU	16	10	95.36	89.58
1	ReLU	32	16	97.41	91.33
1	ReLU	64	7	97.27	91.67
1	ReLU	128	4	96.47	91.18
---	---	---	---	---	---
2	ReLU	32 - 16	5	95.51	90.23
2	ReLU	64 - 32	7	97.78	91.52
2	ReLU	128 - 64	4	96.96	91.87
---	---	---	---	---	---
1	tanh	16	6	95.29	90.23
1	tanh	32	9	97.45	91.62
1	tanh	64	5	96.62	91.42
1	tanh	128	8	98.66	92.07
---	---	---	---	---	---
2	tanh	32 - 16	5	96.33	90.43
2	tanh	64 - 32	5	97.53	91.72
2	tanh	128 - 64	6	98.74	92.17
---	---	---	---	---	---
1	sigmoid	16	10	94.37	88.68
1	sigmoid	32	10	96.10	91.28
1	sigmoid	64	16	98.17	91.97
1	sigmoid	128	8	96.78	91.77
---	---	---	---	---	---
2	sigmoid	32 - 16	19	97.74	91.42
2	sigmoid	64 - 32	16	98.15	92.17
2	sigmoid	128 - 64	16	**98.80**	**92.52**

Then, we evaluate the best perorfming model on the test set and get the following results;

```
# Evaluate the best model on test set
# sigmoid with 2 hidden layers (128 - 64)
correct = 0
total = 0
with torch.no_grad():
    for data in test_loader:
        images, labels = data
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total_val += labels.size(0)
        correct_val += (predicted == labels).sum().item()

print(correct_val / total_val)
```

```
0.9252615844544095
```

At last, we convert the Neural Network that we constructed above, into a scikit-learn compatible estimator and compute the accuracy of it. The fit method trains the neural network and calculates its accuracy on the validation set. The score mathod calculates its accuracy on the test set.

```
print("Sklearn combatible NN " + str(NN_sk.score(X_test, y_test)))
```

```
Sklearn combatible NN 0.9177877428998505
```

We notice that the score is slightly smaller that other methods such as SVM or KNeighbor. We believe that this is because the training set is particularly small. Deep Neural Networks perform better that the classic models when the size of the dataset is not thousands, but millions of data.