

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ



ΑΝΑΓΝΩΡΙΣΗ ΠΡΟΤΥΠΩΝ
Χειμερινό Εξάμηνο Ε.ΔΕ.Μ.Μ. 2022-23

Προπαρασκευή 1ης Εργαστηριακής Άσκησης:
Οπτική Αναγνώριση Ψηφίων

Παπακωνσταντίνου Άννα 03400187
Κοτσιφάκου Κοντιλένια Μαρία 03400174

Project εργαστηρίου

Σκοπός της παρούσας εργαστηριακής αναφοράς αποτελεί η υλοποίηση ενός συστήματος οπτικής αναγνώρισης ψηφίων. Τα δεδομένα προέρχονται από την US Postal Service (γραμμένα στο χέρι σε ταχυδρομικούς φακέλους και σκαναρισμένα) και περιέχουν τα ψηφία από το 0 έως το 9 και διακρίνονται σε *train* και *test*.

Τα datasets αποτελούνται από γραμμές και στήλες. Οι γραμμές αναπαριστούν τα ξεχωριστά ψηφία (δείγματα) από το 0-9 και οι στήλες τα επιμέρους 256 χαρακτηριστικά τους (*features*). Κάθε ψηφίο, δηλαδή κάθε γραμμή, περιγράφεται πλήρως από 257 τιμές ή διαφορετικά 257 στήλες. Η πρώτη τιμή-στήλη αντιστοιχεί στην αξία του ίδιου του ψηφίου και ανήκει στο εύρος 0-9. Οι υπόλοιπες 256 απαρτίζουν τα *features* του.

Κάθε ψηφίο απεικονίζεται ως ένας πίνακας (16x16), όπου σε κάθε κελί του (*pixel*) υπάρχει μία πραγματική τιμή στο εύρος $[-1, 1]$. Το αριστερό άκρο αντιστοιχεί στο απόλυτα μαύρο χρώμα ενώ το δεξί στο απόλυτα λευκό. Κάθε μία από αυτές τις 256 τιμές αντιστοιχίζεται σε μία απόχρωση του μαύρου, έτσι ώστε συνολικά να φωτίζεται με άσπρο χρώμα το περίγραμμα του ψηφίου.

Βήμα 1

Πρωταρχικό βήμα της υλοποίησης αποτελεί η εισαγωγή των datasets στον κώδικα και η επεξεργασία τους (*processing*). Για τον σκοπό αυτό εισάγονται οι κατάλληλες βιβλιοθήκες:

```
import os
import pandas as pd
```

Εικόνα 1.1. Εισαγωγή *libraries*

Υπάρχουν 2 datasets, το πρώτο αποθηκευμένο στο αρχείο *train.txt* που αντιστοιχεί στα δεδομένα εκπαίδευσης και το δεύτερο στο αρχείο *test.txt* που αντιστοιχεί στα δεδομένα με βάση τα οποία θα αξιολογηθεί η απόδοση του ταξινομητή.

Στόχος είναι το πρόγραμμα που υλοποιήθηκε να διαβάζει τα εν λόγω αρχεία από οποιοδήποτε υπολογιστή. Τα *txt* αρχεία βρίσκονται σε ένα φάκελο με όνομα "*data*" και αυτός ο φάκελος πρέπει να βρίσκεται στο ίδιο *path* με το αρχείο *pattern_recognition.py* για να τρέξει σωστά ο κώδικας.

Στην αρχή του κώδικα, χρησιμοποιείται εντολή που επιστρέφει το *path* του τρέχοντος script (*absolute_path*) και ακολούθως συνδυάζεται με το όνομα του φακέλου (*data_folder*) αλλά και του εκάστοτε αρχείου ώστε τελικά να παραχθεί το *path* από το οποίο θα διαβάζονται τα datasets. Η ιδέα αυτή μεταφράζεται στις εντολές python της Εικόνας 1.2.

```
# Find data path combining file absolute path and data folder
absolute_path = os.path.dirname(__file__)
data_path = ["test.txt", "train.txt"]
data_folder = "data"
full_path_test = os.path.join(absolute_path, data_folder, data_path[0])
full_path_train = os.path.join(absolute_path, data_folder, data_path[1])
```

Εικόνα 1.2 Εντολές εύρεσης *path* του dataset

Στην συνέχεια, διαβάζονται τα datasets και αποθηκεύονται σε *dataframes*. Συνεπώς για το *test set* τα δεδομένα αποθηκεύονται στο *test dataframe* και αντιστοίχως το *training set* αποθηκεύεται στο *train dataframe*. Επόμενο βήμα αποτελεί ο διαχωρισμός των ψηφίων από τα υπόλοιπα 256 χαρακτηριστικά τους τόσο από το *test* όσο και από το *training set*. Τελικά, η αρχική πληροφορία διαμοιράζεται στα

εξής τέσσερα dataframes: X_{train} , y_{train} , X_{test} και y_{test} , από τα οποία τα πρώτα δύο μετατράπηκαν και σε numpy ώστε να γίνεται καλύτερα ο υπολογισμός πράξεων στα επόμενα ερωτήματα. Η υλοποίηση περιγράφεται στην ακόλουθη εικόνα.

```
test = pd.read_csv(full_path_test, sep=' ', header=None, index_col=False)
test.dropna(axis=1, how='all', inplace=True)
y_test = test.iloc[:, 0]
X_test = test.iloc[:, 1:]

train = pd.read_csv(full_path_train, sep=' ', header=None, index_col=False)
train.dropna(axis=1, how='all', inplace=True)
y_train = train.iloc[:, 0]
X_train = train.iloc[:, 1:]

X_train1 = X_train.to_numpy()
X_test1 = X_test.to_numpy()
```

Εικόνα 1.3 Διαχωρισμός features και labels

Βήμα 2

Για την απεικόνιση του ψηφίου που ζητείται, αρχικά εισάγονται οι βιβλιοθήκες numpy και pyplot της matplotlib για την κατάλληλη διαχείριση των δεδομένων και μετέπειτα την εμφάνιση του σχήματος (Εικόνα 2.1).

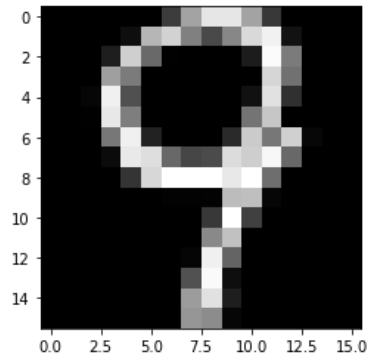
```
import numpy as np
from matplotlib import pyplot as plt
```

Εικόνα 2.1 Εισαγωγή libraries

Από το dataframe των training data επιλέγεται το στοιχείο με index 131 και όλα τα 256 χαρακτηριστικά—στήλες που το απαρτίζουν. Όπως προαναφέρθηκε κάθε ψηφίο αναπαρίσταται ως ένας πίνακας (16x16), συνεπώς μέσω της np.reshape() γίνεται η μετατροπή από ένα πίνακα (1x256) σε ένα πίνακα (16x16). Αφότου υλοποιηθεί αυτή η μετατροπή καλείται η συνάρτηση imshow() με σκοπό την εμφάνιση του ψηφίου στην οθόνη σε ασπρόμαυρη κλίμακα χρωμάτων. Η αναπαράσταση του ψηφίου ακολουθεί στην Εικόνα 2.3 .

```
digit = X_train1[131, :]
digit = np.reshape(digit, (16, 16)) # Reshape digit from 1X256 TO 16X16
plt.imshow(digit, cmap='gray')
plt.show()
```

Εικόνα 2.2 Εντολές απεικόνισης του ψηφίου



Εικόνα 2.3 Απεικόνιση ψηφίου

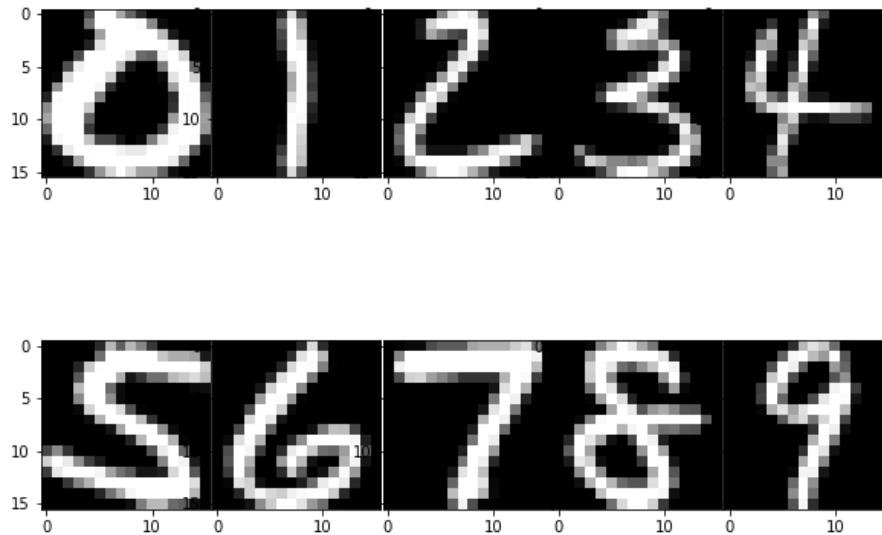
Βήμα 3

Στο βήμα αυτό το πρόγραμμα καλείται να επιλέξει για κάθε ένα από τα δέκα ψηφία (0-9) ένα τυχαίο δείγμα από το `y_train` dataframe. Όπως έχει επεξηγηθεί σε προηγούμενο ερώτημα, το `y_train` dataframe περιλαμβάνει τα ψηφία του training dataset και αντιστοιχεί στα features του `X_train` dataframe. Για το λόγο αυτό σε ένα loop που τρέχει για τα 10 ψηφία, αποθηκεύεται στη `random_sample` μεταβλητή ένα τυχαίο δείγμα και βάση αυτού από το `X_train` dataframe επιλέγεται η αντίστοιχη γραμμή στην οποία βρίσκονται τα 256 χαρακτηριστικά του. Ακολουθεί η μετατροπή του μονοδιάστατου πίνακα (1x256) σε (16x16) και η εισαγωγή του σε μία λίστα `ten_list`. Η διαδικασία αυτή γίνεται συνολικά δέκα φορές, μία για κάθε ψηφίο. Αφού ολοκληρωθεί το loop, με χρήση της συνάρτησης `subplots` σε ένα ενιαίο σχήμα εκτυπώνονται με τη σειρά τα 10 τυχαία δείγματα που επιλέχθηκαν και αποθηκεύτηκαν στην `ten_list`. Η εκτύπωση φαίνεται στην Εικόνα 3.2.

```
ten_list = []
for i in range(10):
    random_sample = y_train[y_train == i].sample(n=1)
    index = random_sample.index
    new = np.reshape(X_train1[index, :], (16, 16))
    ten_list.append(new)

fig, axs = plt.subplots(ncols=5, nrows=2, figsize=(10, 8))
gs1 = gridspec.GridSpec(4, 4)
gs1.update(wspace=0.025, hspace=0.05)
for i, ax in enumerate(axs.ravel()):
    ax.imshow(ten_list[i], cmap='gray')
plt.subplots_adjust(wspace=0.01, hspace=0)
plt.show()
```

Εικόνα 3.1 Εντολές επιλογής και απεικόνισης τυχαίων δειγμάτων



Εικόνα 3.2 Απεικόνιση τυχαίων δειγμάτων για κάθε ψηφίο

Βήμα 4

Για τον υπολογισμό της μέσης τιμής των χαρακτηριστικών του pixel (10, 10) για το ψηφίο 0 με βάση τα train δεδομένα, ορίζεται η μεταβλητή *conv_index*. Το pixel (10,10) κάθε μηδενικού ψηφίου θα πρέπει να μετασχηματιστεί από διάσταση 16x16 σε 1x256. Η μετατροπή αυτή ορίζεται στη μεταβλητή *conv_index*. Επομένως, αφότου περιοριστούν από το συνολικό *X_train* μόνο οι γραμμές που αφορούν το ψηφίο 0, έπειτα απομονώνεται η στήλη που αντιστοιχεί στο ζητούμενο pixel και υπολογίζεται η μέση τιμή όπως δηλώνουν οι εντολές στην Εικόνα 4.1

```
conv_index = (0 - 1) * 16 + 0
zero_pixel_mean = X_train1[y_train == 0][:, conv_index]
mean_val = np.mean(zero_pixel_mean, axis=0)
```

Εικόνα 4.1 Υπολογισμός μέσης τιμής pixel(10,10) για το ψηφίο 0

Βήμα 5

Ακολουθείται ακριβώς η ίδια διαδικασία και για τον υπολογισμό της διασποράς με χρήση της αντίστοιχης συνάρτησης όπως απεικονίζεται στην Εικόνα 5.1

```
zero_pixel_sd = X_train1[y_train == 0][:, conv_index]
var = np.var(zero_pixel_sd, axis=0)
```

Εικόνα 5.1 Υπολογισμός διασποράς pixel(10,10) για το ψηφίο 0

Βήμα 6

Για τον υπολογισμό της μέσης τιμής και διασποράς των χαρακτηριστικών κάθε pixel για το ψηφίο 0 με βάση τα train δεδομένα, απομονώνονται σε πρώτη φάση όλα τα δεδομένα του *X_train* dataframe των μηδενικών ψηφίων. Έπειτα, αφού το dataframe μετατραπεί σε ndarray, εφαρμόζεται πρώτα η συνάρτηση *mean()* και ακολούθως η *var()* σε όλες τις στήλες των χαρακτηριστικών των μηδενικών ψηφίων. Τελικώς, το αποτέλεσμα είναι δύο πίνακες 256 θέσεων, όπου σε κάθε θέση βρίσκεται η μέση

τιμή και αντίστοιχα η διασπορά των τιμών του εκάστοτε pixel των μηδενικών ψηφίων. Η διαδικασία περιγράφεται στην Εικόνα 6.1.

```
# Step 6: mean for each pixel of all the same digits
digit1 = 0
mean_of_digit1 = np.mean(X_train1[y_train == digit1], axis=0)

# Step 6: Standard deviation for each pixel of all the same digits
sd_of_digit1 = np.var(X_train1[y_train == digit1], axis=0)
```

Εικόνα 6.1 Υπολογισμός μέσης τιμής και διασποράς όλων των pixels για το ψηφίο 0

Βήμα 7 & 8

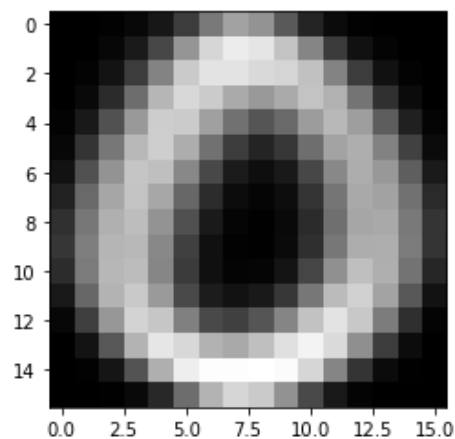
Ακολουθούν οι εντολές για την απεικόνιση του μηδενικού ψηφίου χρησιμοποιώντας τις τιμές της μέσης τιμής αλλά και της διασποράς που υπολογίστηκαν στο βήμα 6.

```
# Step 7
schema_med = np.reshape(mean_of_digit1, (16, 16))
plt.imshow(schema_med, cmap='gray')
plt.show()

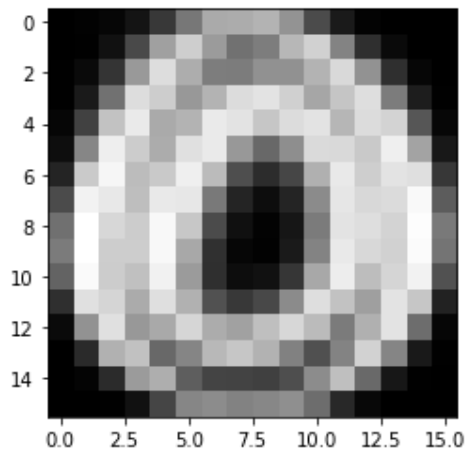
# Step 8
schema_var = np.reshape(variance_of_digit1, (16, 16))
plt.imshow(schema_var, cmap='gray')
plt.show()
```

Εικόνα 7-8.1 Απεικόνιση μέσης τιμής και διακύμανσης του ψηφίου 0

Παρατήρηση: Στην εικόνα 7-8.2 το σχήμα του μηδενικού ψηφίου με χρήση της μέσης τιμής είναι πιο ευδιάκριτο σε σχέση με την Εικόνα 7-8.3 όπου το ψηφίο έχει σχεδιαστεί με τη χρήση της διασποράς των pixels του. Αυτό συμβαίνει διότι το σχήμα της διασποράς αποτυπώνεται με βάση την διαφορά της τιμής των pixels από τη μέση τιμή τους. Συνεπώς στην Εικόνα 7-8.3 αναπαρίσταται το φάσμα γύρω από το οποίο απεικονίζονται τα διάφορα μηδενικά στο training set.



Εικόνα 7-8.2: Μηδενικό ψηφίο με χρήση της μέσης τιμής



Εικόνα 7-8.3: Μηδενικό ψηφίο με χρήση της τιμής διασποράς

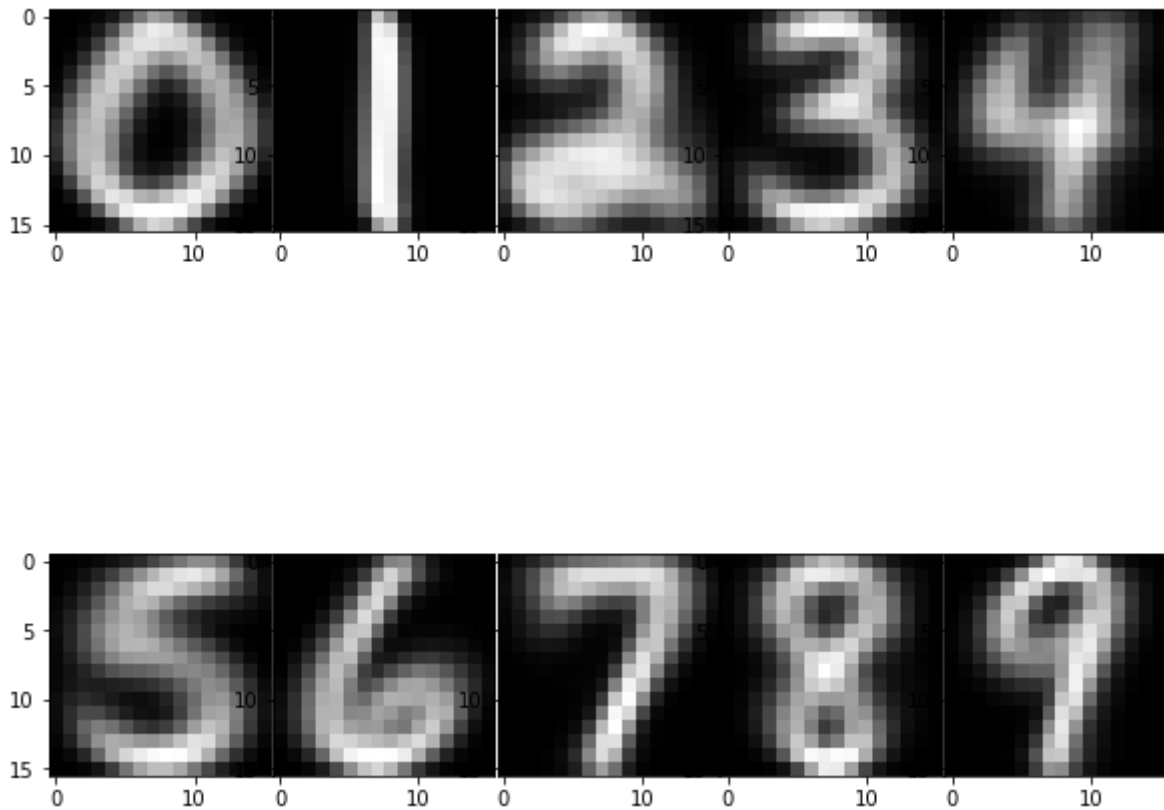
Βήμα 9

Για τον υπολογισμό της μέσης τιμής και διασποράς των χαρακτηριστικών για όλα τα ψηφία (0-9) με βάση τα train δεδομένα, υλοποιήθηκε ένα loop δέκα φόρων, όσα είναι και τα ψηφία. Ο υπολογισμός της μέσης τιμής και της διασποράς αποθηκεύεται στις λίστες *mean_values* και *var_values* αντίστοιχα. Έπειτα, με τη χρήση της συνάρτησης *subplots* και της λίστας με τις μέσες τιμές του καθενός από τα δέκα ψηφία, υλοποιήθηκε η εμφάνιση τους στην οθόνη. Να σημειωθεί πως κάθε στοιχείο της λίστας αποτελεί έναν μονοδιάστατο πίνακα (1x256) έχοντας την πληροφορία της μέσης τιμής των 256 χαρακτηριστικών του ψηφίου, που μετασχηματίζεται σε *ndarray* (16,16).

```
# Step 9 (a)
mean_values = []
var_values = []
for i in range(0, 10):
    mean_variable = np.mean(X_train1[y_train == i], axis=0)
    mean_values.append(mean_variable)
    var_variable = np.var(X_train1[y_train == i], axis=0)
    var_values.append(var_variable)

# Step 9 (b)
fig, axs = plt.subplots(ncols=5, nrows=2, figsize=(10, 10))
gs1 = gridspec.GridSpec(4, 4)
gs1.update(wspace=0.025, hspace=0.05)
for i, ax in enumerate(axs.ravel()):
    ax.imshow(mean_values[i].reshape(16, 16), cmap='gray')
plt.subplots_adjust(wspace=0.01, hspace=0)
plt.show()
```

Εικόνα 9.1 Εντολές υπολογισμού της μέσης τιμής και διασποράς των χαρακτηριστικών για όλα τα ψηφία (0-9)



Εικόνα 9.2 Απεικόνιση των ψηφίων με χρήση της μέσης τιμής τους

Βήμα 10

Για την ταξινόμηση του στοιχείου με index 101 του `y_test` dataframe σε μία από τις 10 κατηγορίες (0-9), ορίζεται η λίστα `euclidian_distance`. Για κάθε μία από τις δέκα κλάσεις (0-9) αποθηκεύεται στην `euclidian_distance` η ρίζα της τετραγωνικής διαφοράς της μέσης τιμής κάθε pixel καθενός από τα 10 ψηφία με τα αντίστοιχα pixels του ζητούμενου ψηφίου ή αλλιώς η ευκλείδεια απόσταση τους. Η ευκλείδεια απόσταση υπολογίζεται μέσω της εντολής `np.linalg.norm()`. Τελικά εκτυπώνεται η κλάση από την οποία το ζητούμενο ψηφίο απέχει την μικρότερη υπολογιζόμενη απόσταση. Στην Εικόνα 10.1 αποτυπώνονται οι εντολές για τον υπολογισμό και την ταξινόμηση του ζητούμενου ψηφίου στην κλάση.

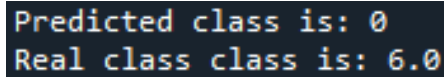
```
euclidian_distance = []
mean_values_np = np.array(mean_values)
for i in range(10):
    # Euclidian distance between mean and the element 101
    euclidian_distance.append(np.linalg.norm(mean_values_np[i] - X_test1[101, :]))

# Select class base on the minimum distance
print("Predicted class is: " + str(euclidian_distance.index(min(euclidian_distance))))
print("Real class class is: " + str(y_test.iloc[101]))
```

Εικόνα 10.1 Εντολές ταξινόμησης στοιχείου σε κλάση

Να σημειωθεί πως τα αποτελέσματα του ευκλείδιου ταξινομητή στη συγκεκριμένη περίπτωση δεν είναι πολύ ενθαρρυντικά, καθώς το ψηφίο με index 131 εντέλει είναι ίσο με 6, ωστόσο το πρόγραμμα το

ταξινόμησε στη μηδενική κλάση, όπως φαίνεται και στην Εικόνα 10.2. Παρολα αυτά, τα ψηφία 0 και 6 έχεις αρκετά όμοιο σχήμα οπότε το σφάλμα αυτό δεν θεωρείται ακραίο.

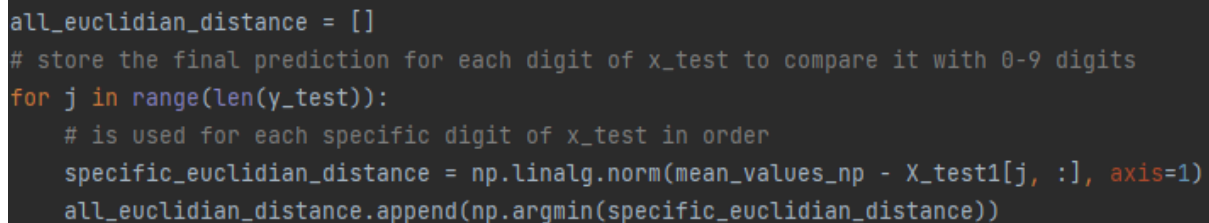


```
Predicted class is: 0
Real class is: 6.0
```

Εικόνα 10.2 Μήνυμα πρόβλεψης κλάσης του ταξινομητή

Βήμα 11

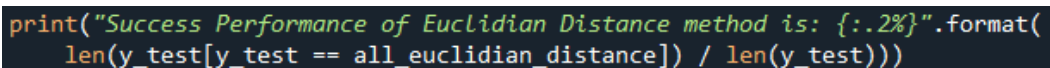
Για την ταξινόμηση όλων των ψηφίων των test δεδομένων σε μία από τις 10 κατηγορίες με βάση την Ευκλείδεια απόσταση, δημιουργείται μία λίστα *all_euclidian_distance*, στην οποία με βάση τη λογική του βήματος 10, αποθηκεύεται για κάθε ένα από τα δείγματα του *y_test*, η κλάση από την οποία απέχει την minimum απόσταση. Η κλάση αυτή ορίζεται ως η πρόβλεψη του ευκλείδειου ταξινομητή για κάθε ένα ψηφίο. Σε αυτή τη διαδικασία, σε κάθε έλεγχο ενός ψηφίου του test set, υπάρχει μία μεταβλητή, η *specific_euclidian_distance*, στην οποία υπολογίζονται και αποθηκεύονται οι αποστάσεις του ψηφίου από κάθε κλάση. Αμέσως μετά, αποθηκεύεται στην *all_euclidian_distance* το index της μικρότερης απόστασης από όλες τις αποστάσεις του ψηφίου που υπάρχουν στην *specific_euclidian_distance*. Το index αυτό μπορεί να έχει τιμές από 0 έως 9 και αντιστοιχεί στις αντίστοιχες κλάσεις 0-9.



```
all_euclidian_distance = []
# store the final prediction for each digit of x_test to compare it with 0-9 digits
for j in range(len(y_test)):
    # is used for each specific digit of x_test in order
    specific_euclidian_distance = np.linalg.norm(mean_values_np - X_test1[j, :], axis=1)
    all_euclidian_distance.append(np.argmin(specific_euclidian_distance))
```

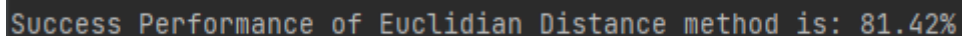
Εικόνα 11.1 Εντολές ταξινόμησης όλων των στοιχείων σε κλάσεις

Για τον υπολογισμό του ποσοστού επιτυχίας, ορίζεται το πηλίκο με αριθμητή τον αριθμό των ψηφίων του *y_test* που ισούνται με τις προβλέψεις του ευκλείδειου ταξινομητή, δηλαδή τις τιμές της λίστας *all_euclidian_distance*, προς το παρονομαστή που ορίζεται από το σύνολο όλων των ψηφίων του *y_test*. Αυτή ονομάζεται και μετρική accuracy. Στη συγκεκριμένη υλοποίηση ήταν ίση με 81.42% (Εικόνα 11.3).



```
print("Success Performance of Euclidian Distance method is: {:.2%}".format(
    len(y_test[y_test == all_euclidian_distance]) / len(y_test)))
```

Εικόνα 11.2 Υπολογισμός του ποσοστού επιτυχίας



```
Success Performance of Euclidian Distance method is: 81.42%
```

Εικόνα 11.3 Εκτύπωση ποσοστού επιτυχίας

Βήμα 12

Σε αυτό το βήμα, δημιουργήθηκε η κλάση *EuclideanDistanceClassifier* σε μορφή scikit-learn estimator για να είναι συμβατή με τις συναρτήσεις του scikit-learn που θα χρησιμοποιηθούν στα επόμενα ερωτήματα. Ο κώδικας που αποτελεί τον σκελετό της κλάσης έχει αντιγραφεί από τα αρχεία που παρέχεται στο github και αφορούν την πρώτη εργασία του μαθήματος.

Η παραπάνω κλάση περιέχει το property *X_mean* όπου αρχικοποιείται στον constructor της κλάσης και χρησιμοποιείται για να αποθηκεύσει την μοναδική παράμετρο του ταξινομητή ευκλείδειας

απόστασης, δηλαδή την μέση τιμή κάθε pixel για κάθε μία από τις κλάσεις. Περιέχονται επίσης, τρεις μέθοδοι που θα αναλυθούν στην συνέχεια. Η πρώτη μέθοδος είναι η *fit* στην οποία δίνονται ως ορίσματα τα χαρακτηριστικά και τα labels των δεδομένων εκπαίδευσης. Σε αυτήν αρκεί να υπολογιστεί η μέση τιμή όλων των pixel από όλες τις κλάσεις. Ο κώδικας υπολογισμού μέσης τιμής που περικλείεται μέσα στην συνάρτηση είναι ο ίδιος που περιγράφηκε στο Βήμα 9. Μοναδική διαφορά αποτελεί η μετατροπή του ορίσματος των χαρακτηριστικών σε ndarray σε περίπτωση που είναι άλλου τύπου ώστε να γενικευτεί η συνάρτηση.

```
class EuclideanDistanceClassifier(BaseEstimator, ClassifierMixin):
    """Classify samples based on the distance from the mean feature value"""

    def __init__(self):
        self.X_mean_ = None

    def fit(self, X, y):
        """
        This should fit classifier. All the "work" should be done here.
        Calculates self.X_mean_ based on the mean
        feature values in X for each class.
        self.X_mean_ becomes a numpy.ndarray of shape
        (n_classes, n_features)
        fit always returns self.
        """
        mean_value = []
        if not isinstance(X, np.ndarray):
            X = X.to_numpy()
        for i in range(0, 10):
            mean_variables = np.mean(X[y == i], axis=0)
            mean_value.append(mean_variables)
        self.X_mean_ = np.array(mean_value)
        return self
```

Εικόνα 12.1 Ορισμός κλάσης EuclideanDistanceClassifier (α μέρος)

Η δεύτερη μέθοδος που απαιτείται είναι η *predict* που δέχεται ως όρισμα τα χαρακτηριστικά για κάποια δεδομένα. Αφού μετατραπούν σε ndarray (σε περίπτωση που δεν είναι), γίνεται η πρόβλεψη για κάθε ένα από τα δοθέντα δεδομένα ακριβώς όπως περιγράφηκε στο Βήμα 11. Σε αυτή την μέθοδο, επιστρέφονται όλα τα labels που προβλέφθηκαν.

```
def predict(self, X):
    """
    Make predictions for X based on the
    euclidean distance from self.X_mean_
    """
    predictions = []
    # store the final prediction for each digit of x_test to compare it with 0-9 digits
    if not isinstance(X, np.ndarray):
        X = X.to_numpy()
    for i in range(X.shape[0]):
        # is used for each specific digit of x_test in order
        euclidian_distances = np.linalg.norm(self.X_mean_ - X[i, :], axis=1)
        predictions.append(np.argmin(euclidian_distances))
    return predictions
```

Εικόνα 12.2 Ορισμός κλάσης EuclideanDistanceClassifier (β μέρος)

Η τρίτη και τελευταία μέθοδος της κλάσης είναι η *score*. Αυτή η μέθοδος παίρνει τα χαρακτηριστικά κάποιων δεδομένων και τις πραγματικές ετικέτες και υπολογίζει την ακρίβεια (accuracy) της πρόβλεψης του μοντέλου όπως περιγράφηκε στο Βήμα 11. Αξίζει να αναφερθεί ότι το τελευταίο όρισμα της μεθόδου (*sample_weight*) προστέθηκε αργότερα καθώς δεν μπορούσε να γίνει σωστά override η μέθοδος *score* από τις κλάσεις που κληρονομεί η *EuclideanDistanceClassifier* δηλαδή τις *BaseEstimator* και *ClassifierMixin*.

Μετά τον ορισμό της, δημιουργήθηκε ένα αντικείμενο της κλάσης αυτής, όπου εκπαιδεύτηκε με τα training δεδομένα και στην συνέχεια εκτιμήθηκε η ακρίβεια της πρόβλεψης του μοντέλου αυτού στα testing δεδομένα. Η ακρίβεια φαίνεται στην Εικόνα 12.4 και εμφανίζεται ακριβώς η ίδια ακρίβεια σε σχέση με το Βήμα 11, καθώς ο κώδικας δεν άλλαξε παρα μόνο οργανώθηκε στην συγκεκριμένη κλάση.

```
def score(self, X, y, sample_weight=None):
    """
    Return accuracy score on the predictions
    for X based on ground truth y
    """
    pred = self.predict(X)
    return len(y[y == pred]) / len(y)

simple_model = EuclideanDistanceClassifier()
model = simple_model.fit(X_train, y_train)
score = simple_model.score(X_test, y_test)
print("Success Performance of Euclidian Distance method is: {:.2%}".format(score))
```

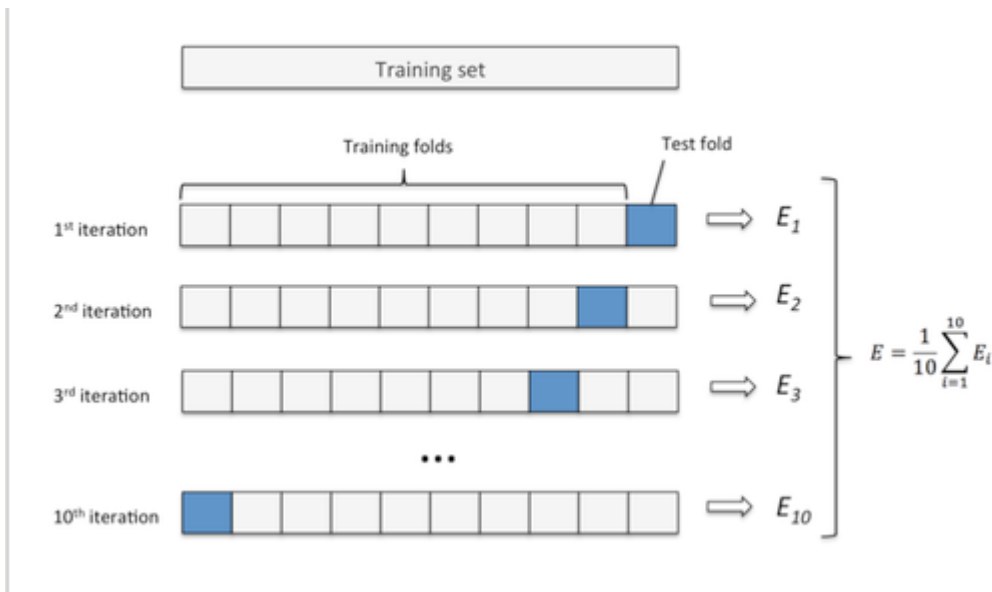
Εικόνα 12.3 Ορισμός κλάσης *EuclideanDistanceClassifier* (γ μέρος)

```
Success Performance of Euclidian Distance method is: 81.42%
```

Εικόνα 12.4 Εκτύπωση ποσοστού επιτυχίας

Βήμα 13

Όσον αφορά το θεωρητικό κομμάτι, το Cross Validation (CV) παρέχει τη δυνατότητα εκπαίδευσης του μοντέλου σε διαφορετικά train και validation set δεδομένων. Μ' αυτό τον τρόπο μπορεί να αποδοθεί μια μέση εκτίμησης της απόδοσης του μοντέλου σε πραγματικά άγνωστα δεδομένα. Αρχικά, στο Cross Validation, το training set χωρίζεται σε έναν αριθμό "πτυχών" (folds). Συνηθισμένες τιμές είναι το 5 και το 10 (5-fold και 10-fold CV). Στη συνέχεια, για κάθε k-fold, θεωρείται ότι τα k-1 folds απαρτίζουν το training set και ότι το fold που έμεινε εκτός του training procedure, είναι το validation set. Υπολογίζεται η μετρική σφάλματος στο validation set που ορίζει το fold. Η διαδικασία επαναλαμβάνεται για τα k folds για κάθε τιμή των υπερπαραμέτρων και υπολογίζεται η μέση τιμή της μετρικής του σφάλματος. Με αυτό τον τρόπο, αφενός επιτυγχάνεται η αμεροληψία στην αξιολόγηση αφήνοντας τελείως έξω το test set και αφετέρου χρησιμοποιούνται αποτελεσματικά τα δεδομένα εκπαίδευσης.



Για τον υπολογισμό του score του ευκλείδειου ταξινομητή με χρήση 5-fold cross-validation, αρχικά δημιουργείται ένα νέο μοντέλο κλάσης *EuclideanDistanceClassifier*. Έπειτα καλείται η συνάρτηση *cross_val_score* με τα κατάλληλα ορίσματα δηλαδή το μοντέλο ευκλείδειου ταξινομητή, τα *X_train* και *y_train* dataframes και θέτοντας επίσης την παράμετρο *cv folds* ίση με 5. Το ποσοστό επιτυχίας του ταξινομητή με την 5-folds Cross Validation υπολογίστηκε ίσο με 84.89%. Ελαφρώς καλύτερο σε σχέση με την προηγούμενη μέτρηση.

```
cv_model = EuclideanDistanceClassifier()
scores = cross_val_score(cv_model, X_train, y_train, cv=5)
```

Εικόνα 13.1 Υπολογισμός score του ευκλείδειου ταξινομητή

Decision Surface

Για την σχεδίαση της περιοχή απόφασης του ευκλείδειου ταξινομητή, θα πρέπει να ληφθεί υπόψη η αναγκαιότητα επιλογής των δύο επικρατέστερων χαρακτηριστικών-features του κάθε δείγματος καθώς δεν υπάρχει η δυνατότητα αναπαράστασης και των 256 features κάθε ψηφίου στις δύο διαστάσεις. Ένας από τους πιο συνηθισμένους τρόπους για την επίτευξη Μείωσης Διαστάσεων (Dimensionality Reduction) είναι η Εξαγωγή Χαρακτηριστικών (Feature Extraction), στην οποία πραγματοποιείται μείωση του αριθμού των διαστάσεων αντιστοιχίζοντας έναν χώρο χαρακτηριστικών υψηλότερων διαστάσεων σε έναν χώρο χαρακτηριστικών χαμηλότερης διάστασης. Η πιο δημοφιλής τεχνική εξαγωγής χαρακτηριστικών είναι η Ανάλυση Κύριων Συνιστωσών, Principal Component Analysis, (PCA). Για το λόγο αυτό εισάγεται από την *sklearn* η κλάση PCA και εφαρμόζεται για την απομόνωση των 2 βασικών χαρακτηριστικών του dataset όπως φαίνεται στις Εικόνες 13.2 και 13.3.

```
from sklearn.decomposition import PCA # to apply PCA
import seaborn as sns # to plot the heat maps
```

Εικόνα 13.2 Εισαγωγή libraries

```

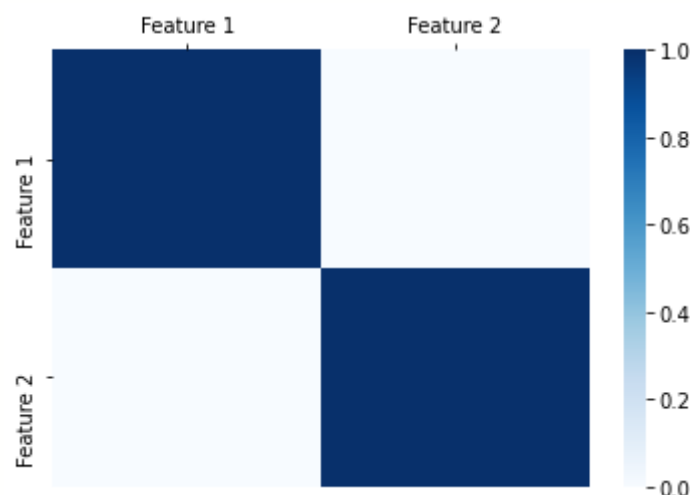
pca = PCA(n_components=2)
pca.fit(X_train)
X_train_curtailed = pca.transform(X_train)
pca.fit(X_test)
X_test_curtailed = pca.transform(X_test)

# Checking Co-relation between features after PCA
Train_curtailed = pd.DataFrame(X_train_curtailed, columns=['Feature 1', 'Feature 2'])
new = sns.heatmap(Train_curtailed.corr(), cmap="Blues")
new.xaxis.tick_top()
plt.show()

```

Εικόνα 13.3 Εφαρμογή PCA στα datasets

Ένας τρόπος για την οπτικοποίηση της συσχέτισης μεταξύ των διαφόρων χαρακτηριστικών δίνεται από τον χάρτη θερμότητας μέσω της συνάρτησης `heatmap()`. Η χρωματική κλίμακα στο πλάι του `heatmap` βοηθά στον προσδιορισμό του μεγέθους της συσχέτισης. Στη συγκεκριμένη περίπτωση της Εικόνας 13.4, μια πιο σκούρα απόχρωση αντιπροσωπεύει μεγαλύτερη συσχέτιση, ενώ μια πιο ανοιχτή απόχρωση αντιπροσωπεύει λιγότερη συσχέτιση. Η διαγώνιος του χάρτη θερμότητας αντιπροσωπεύει τη συσχέτιση ενός χαρακτηριστικού με τον εαυτό του η οποία είναι πάντα 1,0, επομένως, η διαγώνιος του χάρτη θερμότητας είναι της πιο σκούρης απόχρωσης.



Εικόνα 13.4 Heatmap των κύριων χαρακτηριστικών μετά την εφαρμογή PCA

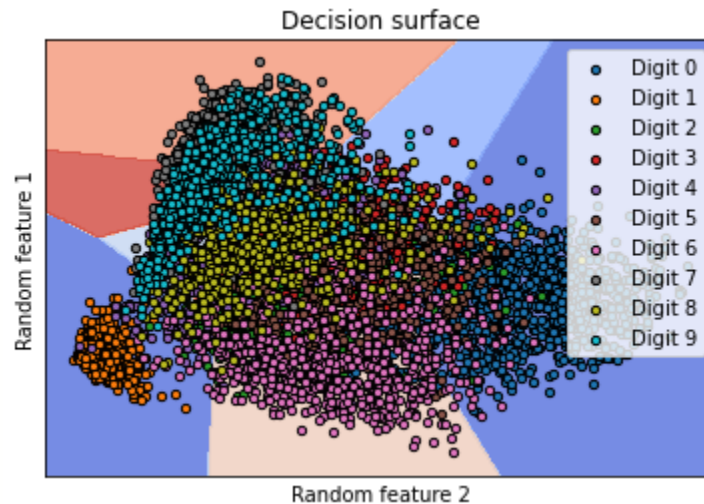
Το παραπάνω heatmap απεικονίζει ξεκάθαρα ότι δεν υπάρχει συσχέτιση μεταξύ των δύο λαμβανόμενων κύριων στοιχείων. Από τον χώρο χαρακτηριστικών υψηλότερων διαστάσεων το πρόβλημα μετακινήθηκε σε έναν χώρο χαρακτηριστικών χαμηλότερης διάστασης, διασφαλίζοντας παράλληλα ότι δεν υπάρχει συσχέτιση μεταξύ των χαρακτηριστικών που λαμβάνονται με αυτόν τον τρόπο. Ως εκ τούτου, έχει επιτύχει ο στόχος της PCA.

Κατόπιν της PCA, εκπαιδεύεται εκ νέου το μοντέλο του *EuclideanDistanceClassifier* με το νέο μικρότερο `X_train_curtailed` των δύο βασικών χαρακτηριστικών και το `y_train` όπως αποτυπώνεται στην Εικόνα 13.5.

```
pca_model = EuclideanDistanceClassifier()
pca_model = pca_model.fit(X_train_curtailed, y_train)
plot_clf(pca_model, X_train_curtailed, y_train)
```

Εικόνα 13.5 Εκπαίδευση του *EuclideanDistanceClassifier* με τα δεδομένα κατόπιν της PCA

Τέλος για την απεικόνιση τελικά της επιφάνειας απόφασης χρησιμοποιείται η συνάρτηση *plot_clf* την οποία ορίζεται το μοντέλο εκπαίδευσης, και τα *X_train_curtailed* και *y_train*. Να σημειωθεί πως η συνάρτηση *plot_clf* δόθηκε στο φροντιστήριο pythοn του μαθήματος και χρησιμοποιήθηκε με κάποιες προσαρμογές.

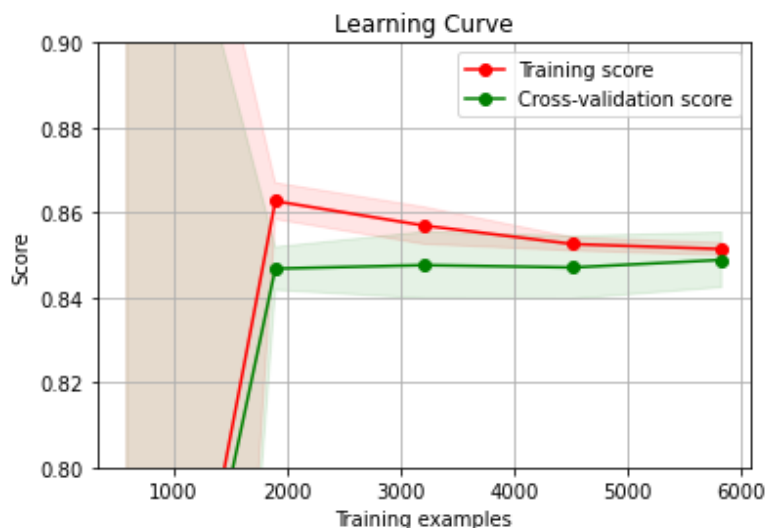


Εικόνα 13.6 Απεικόνιση επιφάνειας απόφασης

Learning Curve

Μια καμπύλη μάθησης (learning curve) είναι μια γραφική παράσταση της απόδοσης μάθησης του μοντέλου σε σχέση με την εμπειρία ή το χρόνο. Το μοντέλο μπορεί να αξιολογηθεί στο σύνολο δεδομένων εκπαίδευσης (training set) και σε ένα σύνολο δεδομένων επικύρωσης (validation set) μετά από κάθε ενημέρωση κατά τη διάρκεια της εκπαίδευσης και μπορούν να δημιουργηθούν γραφικές παραστάσεις της μετρούμενης απόδοσης για την εμφάνιση καμπυλών μάθησης.

Στη συγκεκριμένη υλοποίηση, χρησιμοποιήθηκε η συνάρτηση *plot_learning_curve* που δόθηκε στο φροντιστήριο pythοn του μαθήματος και χρησιμοποιήθηκε με κάποιες προσαρμογές. Η αποτύπωση της καμπύλης μάθησης φαίνεται στην Εικόνα 13.7.



Εικόνα 13.7 Απεικόνιση *learning curve*

Είναι εμφανές από την εικόνα, πως η καμπύλη μάθησης που αφορά τα δεδομένα εκπαίδευσης παρουσιάζει μεγαλύτερο ποσοστό ακρίβειας σε μικρό πλήθος δεδομένων και φθίνει καθώς αυξάνονται. Η πτώση της ακρίβειας συμβαίνει καθώς το μοντέλο καλείται να περιγράψει περισσότερη πληροφορία. Παρολα αυτά, αυτή η επιπλέον πληροφορία δεδομένων επιτρέπει στο μοντέλο να γενικεύσει την γνώση αυτή και να αναγνωρίζει καλύτερα νέα δεδομένα που δεν έχει εκπαιδευτεί. Γι' αυτό η καμπύλη των *validation* δεδομένων αυξάνεται ελαφρώς όσο αυξάνεται και το πλήθος των δεδομένων εκπαίδευσης. Το σημείο που συγκλίνουν οι δύο καμπύλες μαρτυρά το πλήθος των δεδομένων που αρκούν για έχει το μοντέλο την καλύτερη δυνατή ακρίβεια.

Βήμα 14

Για τον υπολογισμό των *a-priori* πιθανοτήτων κάθε κατηγορίας (*class priors*), δημιουργήθηκε μία λίστα *all_apriori*, στην οποία για κάθε ένα από τα δέκα ψηφία, στο εύρος 0-9, αποθηκεύτηκε ο λόγος όλων των ψηφίων από το *training set* που ισούνται με το εν λόγω ψηφίο (0-9) προς το συνολικό άθροισμα όλων των ψηφίων. Οι εντολές φαίνονται στην Εικόνα 14.1

```
all_apriori = []
[all_apriori.append(len(y_train[y_train == i]) / len(y_train)) for i in range(0, 10)]
```

Εικόνα 14.1 Εντολές υπολογισμού *a-priori* πιθανοτήτων

Βήμα 15-16

Η βασική ιδέα λειτουργίας του Naive Bayesian ταξινομητή είναι ο γνωστός νόμος του Bayes

$$P(A|B) = \frac{P(B|A) P(A)}{P(B)}$$

και η (*naive*) υπόθεση ότι τα χαρακτηριστικά είναι όλα ανεξάρτητα μεταξύ τους (δεν ισχύει γενικά, αλλά ο ταξινομητής είναι πρακτικά καλός σε πολλές περιπτώσεις).

Με δεδομένα μια μεταβλητή κατηγορίας (κλάσης) y και ένα εξαρτώμενο διάνυσμα χαρακτηριστικών x_1 μέχρι και x_n , σύμφωνα με το θεώρημα του Bayes θα ισχύει:

$$P(y|x_1, \dots, x_n) = \frac{P(x_1, \dots, x_n|y) P(y)}{P(x_1, \dots, x_n)} \quad (1)$$

Ισχύει ότι $P(x_1, \dots, x_n|y) = \prod_{i=1}^n P(x_i|y, x_1 \dots x_{i-1}, x_{i+1}, \dots, x_n)$

και κάνοντας την αφελή υπόθεση ότι το χαρακτηριστικό x_i για κάθε i εξαρτάται μόνο από την κλάση y και όχι από οποιοδήποτε άλλο χαρακτηριστικό τότε:

$$P(x_i|y, x_1 \dots x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i|y)$$

οδηγώντας στην απλοποίηση της σχέσης (1):

$$P(y|x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^n P(x_i|y)}{P(x_1, \dots, x_n)}$$

Με δεδομένη είσοδο, το $P(x_1, \dots, x_n)$ είναι σταθερό. Συνεπώς μπορεί να χρησιμοποιηθεί ο ακόλουθος κανόνας ταξινόμησης:

$$P(y|x_1, \dots, x_n) \simeq P(y) \prod_{i=1}^n P(x_i|y) \Rightarrow \hat{y} = \underset{y}{\operatorname{argmax}} P(y) \prod_{i=1}^n P(x_i|y)$$

Το $P(y)$ είναι η υπόθεση και ισούται με τη σχετική συχνότητα της κλάσης y στο training set.

Το $P(x_i|y)$ είναι η πιθανοφάνεια δηλαδή η πιθανότητα του δείγματος με δεδομένη την υπόθεση και μπορεί επίσης να υπολογιστεί απλά από το training set.

Οι διάφοροι Naive Bayes classifiers διαφοροποιούνται κυρίως από τις υποθέσεις που κάνουν ως προς την κατανομή $P(x_i|y)$.

Η κλάση \hat{y} που ανατίθεται σε ένα νέο δείγμα είναι αυτή που μεγιστοποιεί το δεξί μέλος της σχέσης.

Στο συγκεκριμένο πρόβλημα η υπόθεση για την κατανομή $P(x_i|y)$ έγκειται στη θεώρηση ότι η κατανομή κάθε χαρακτηριστικού ως προς κάθε κλάση ακολουθεί την κανονική κατανομή:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi}\sigma_y} e^{\frac{-(x_i - \mu_y)^2}{2\sigma_y^2}}$$

Ο συγκεκριμένος ταξινομητής είναι ο Gaussian Naive Bayes. Πρακτικά, με τα δεδομένα του training set, για κάθε κλάση υπολογίζεται η μέση τιμή μ_y και η διακύμανση σ_y^2 κάθε χαρακτηριστικού για τη συγκεκριμένη κλάση. Πρακτικά, όσο πιο κοντά στη μέση τιμή του (ως προς το σύνολο του train set) είναι ένα χαρακτηριστικό ενός δείγματος, τόσο πιο κοντά στη μονάδα θα είναι η πιθανοφάνεια του χαρακτηριστικού και αντίστροφα.

Αρχικά για την υλοποίηση αυτού του ερωτήματος ορίζεται εξωτερικά η συνάρτηση `calculate_probability` η οποία παίρνει σαν ορίσματα το εκάστοτε ψηφίο από το training set, μία μέση

τιμή και μία διασπορά. Ο υπολογισμός που γίνεται στην εν λόγω συνάρτηση αφορά τη συνάρτηση πυκνότητας πιθανότητας (probability density function) για το εκάστοτε ψηφίο δοσμένων της μέσης τιμής και διακύμανσης, όπως αποτυπώνεται και στην Εικόνα 15-6.1. Να σημειωθεί πως για αυτή την υλοποίηση εισήχθη η βιβλιοθήκη `scipy.stats` (Εικόνα 15-6.2)

```
def calculate_probability(x, mean, var):  
    return sts.norm.pdf(x, mean, np.sqrt(var))
```

Εικόνα 15-16.1 Υπολογισμός της pdf

```
import scipy.stats as sts
```

Εικόνα 15-16.2 Εισαγωγή `scipy.stats` library

```
class NaiveBayesClassifier(BaseEstimator, ClassifierMixin):  
  
    def __init__(self):  
        self.X_mean_ = None  
        self.X_var_ = None  
        self.X_apriori_ = None  
        self._var_smoothing_ = 1e-5
```

Εικόνα 15-16.3 Customized Naive Bayes

Έπειτα δημιουργείται η customized κλάση του *Gaussian Naive Bayes* estimator σε μορφή `scikit-learn` estimator για να είναι συμβατή με τις συναρτήσεις του `scikit-learn` που θα χρησιμοποιηθούν στα επόμενα ερωτήματα. Η κλάση περιέχει τα εξής properties, την `X_mean_` που χρησιμοποιείται για την αποθήκευση της μέσης τιμής κάθε pixel ενός ψηφίου από το training set, την `X_var_` για την αποθήκευση της διασποράς του, την `X_apriori_` για τις πιθανότητες εμφάνισης καθενός από τα δέκα ψηφία (0-9) και την `_var_smoothing`. Αναφορικά με την `_var_smoothing`, αρχικοποιείται με μία πολύ μικρή τιμή στον constructor της κλάσης, ίση με $1e-5$. Η ύπαρξη της εξυπηρετεί φαινόμενα μηδενισμού του variance. Στο documentation της `sklearn` για τον Gaussian Naive Bayes classifier, ορίζεται αντίστοιχα η παράμετρος **`var_smoothing` (default=1e-9)** η οποία πολλαπλασιάζεται με τη μέγιστη διακύμανση όλων των χαρακτηριστικών. Έπειτα αυτό το γινόμενο προστίθεται σε όλες τις διακυμάνσεις και κατ'αυτό τον τρόπο αποφεύγονται φαινόμενα αστάθειας και επιτυγχάνεται σταθερότητα στον υπολογισμό. Είναι σημαντικό να τονίσουμε πως η προσθήκη αυτής της παραμέτρου ήταν απαραίτητη γιατί τα δεδομένα παράγουν διακυμάνσεις που είναι ίσες με 0 και δημιουργείται πρόβλημα στον τύπο υπολογισμού της PDF.

Η πρώτη μέθοδος είναι η `fit` στην οποία δίνονται ως ορίσματα τα χαρακτηριστικά και τα labels των δεδομένων εκπαίδευσης και η διακύμανση `nb_var` (Εικόνα 15-6.4α). Ο λόγος που δίνεται η διακύμανση ως όρισμα είναι η υλοποίηση να μπορεί να συνδυαστεί με το ερώτημα 16 όπου ο χρήστης εισάγει ως τιμή διακύμανσης την μονάδα στον ταξινομητή. Η default τιμή ορίζεται ως "None".

```
def fit(self, X, y, nb_var=None):  
    mean_value = []  
    var_value = []  
    apriori_value = []
```

Εικόνα 15-16.4α Customized Naive Bayes - fit function

Μετέπειτα γίνεται έλεγχος για την τιμή της και στην περίπτωση που δεν έχει εισαχθεί κάποια τιμή κατά την κλήση της κλάσης, τότε υπολογίζεται και καταχωρείται στην μεταβλητή *small_number* ο πολλαπλασιασμός της μέγιστη διακύμανσης όλων των χαρακτηριστικών με την σταθερά *_var_smoothing*. Τέλος η *small_number* αθροίζεται σε όλες τις διακυμάνσεις (Εικόνα 15-6.4β).

```
small_number = 0.0
if nb_var is None:
    small_number = self._var_smoothing * var_value.max()
var_value += small_number
```

Εικόνα 15-16.4β Customized Naive Bayes - fit function

Ωστόσο θα πρέπει πρώτα να έχουν οριστεί για κάθε ένα από τα ψηφία 0-9 οι μέσες τιμές αλλά και οι διακυμάνσεις καθενός από τα pixels τους. Υπολογίζονται και αποθηκεύονται στις λίστες *mean_values* και *var_values* αντίστοιχα, αυτό αποτυπώνεται στην Εικόνα 15-6.4γ. Μάλιστα να σημειωθεί πως στην περίπτωση που η *nb_var* ισούται με μονάδα, η λίστα *var_values* γεμίζει με 256 άσσους και δεν γίνεται η διαδικασία υπολογισμού βάσει των training data.

```
for i in range(0, 10):
    mean_variables = np.mean(X[y == i], axis=0)
    mean_value.append(mean_variables)
    if nb_var is None:
        var_variables = np.var(X[y == i], axis=0)
        var_value.append(var_variables)
    else:
        var_value.append(np.ones(256) * nb_var)
    apriori_value.append(len(y[y == i]) / len(y))
```

Εικόνα 15-16.4γ Customized Naive Bayes - fit function

Επόμενη συνάρτηση αποτελεί η predict. Ορίζονται δύο λίστες, η *probabilities* και η *predictions*. Για κάθε ένα από τα ψηφία (i) του training set (first loop), γίνεται έλεγχος με κάθε ένα από τα δέκα ψηφία (k) 0-9 (second loop) με κάθε ένα από τα χαρακτηριστικά features (j) του ψηφίου (third loop). Πρακτικά καλείται η συνάρτηση *calculate_probability* και υπολογίζεται η pdf για κάθε pixel του ψηφίου με παραμέτρους το εκάστοτε pixel, τη μέση τιμή και τη διασπορά του pixel του προς σύγκριση ψηφίου (0-9). Πολλαπλασιάζονται οι pdf για όλα τα features του ψηφίου και στη συνέχεια το γινόμενο αυτό πολλαπλασιάζεται με την apriori πιθανότητα εμφάνισης του k-οστού ψηφίου (0-9) με το οποίο γίνεται και η σύγκριση του i-οστού ψηφίου του training set. Αυτή η πιθανότητα αποθηκεύεται προσωρινά στην λίστα *probabilities*, στην οποία συνολικά θα αποθηκευτούν 10 πιθανότητες. Κάθε μία από αυτές θα ερμηνεύει την πιθανότητα το i-οστο ψηφίο του training set να ανήκει σε μία από τις κλάσεις από το 0-9. Τέλος στην *predictions* αποθηκεύεται για κάθε i ψηφίο, η μέγιστη από αυτές τις πιθανότητες που τελικά αποτελεί και την πρόβλεψη του ταξινομητή. Οι εντολές περιγράφονται στην Εικόνα 15-6.5

```
def predict(self, X):
    probabilities = []
    predictions = []
    # store the final prediction for each digit of x_test
    # to compare it with 0-9 digits
    if not isinstance(X, np.ndarray):
        X = X.to_numpy()
    for i in range(X.shape[0]):
        for k in range(0, 10):
            a = 1
            for j in range(X.shape[1]):
                a = a * calculate_probability(X[i, j], self.X_mean_[k, j], self.X_var_[k, j])
            probabilities.append(a * self.X_apriori_[k])
            predictions.append(np.argmax(probabilities))
            probabilities.clear()
    return predictions
```

Εικόνα 15-16.5 Customized Naive Bayes - predict function

Τελική συνάρτηση της κλάσης αποτελεί η *score*. Η υλοποίηση της είναι ακριβώς ίδια με αυτή που περιγράφηκε και στον Ευκλείδειο ταξινομητή και αποτυπώνεται στην Εικόνα 15-6.6.

```
def score(self, X, y, sample_weight=None):
    pred = self.predict(X)
    return len(y[y == pred]) / len(y)
```

Εικόνα 15-16.6 Customized Naive Bayes - score function

Στην συνέχεια δημιουργείται ένα αντικείμενο της παραπάνω κλάσης και αφού εκπαιδευτεί το μοντέλο, γίνεται πρόβλεψη της ακρίβειας (Εικόνα 15-6.7).

```
Bayes = NaiveBayesClassifier()
Bayes = Bayes.fit(X_train, y_train)
Bayes_score = Bayes.score(X_test, y_test)
```

Εικόνα 15-16.7 Χρήση του μοντέλου Naive Base

Οι προσπάθειες δημιουργίας του ταξινομητή χωρίς την προσθήκη της παραμέτρου *var_smoothing* ήταν απογοητευτικά χαμηλά ή/και προκαλούσαν σφάλματα στην Python λόγω μηδενικής διακύμανσης. Είναι αξιοσημείωτο το γεγονός πως μέσα από αυτές τις προσπάθειες παρατηρήθηκε πως η συνάρτηση PDF της κανονικής κατανομής επιστρέφει τιμές μεγαλύτερες της μονάδας όταν υπάρχει μικρή διακύμανση κι αυτό ήταν και το έναυσμα για εύρεση κάποιας αποδοτικότερης λύσης. Με την χρήση της παραμέτρου αυτής φαίνεται πως ο ταξινομητής Bayes έχει ένα επαρκές ποσοστό ακρίβειας (Εικόνα 15-6.8). Πάντως, η συνάρτηση PDF εξακολουθούσε να παράγει τιμές μεγαλύτερες της μονάδας ακόμα και με αυτή την λύση ωστόσο εξαλείφθηκε η μηδενική διακύμανση. Τέλος, θα ήταν καλό να σχολιαστεί ότι το ποσοστό ακρίβειας είναι χαμηλότερο από το αντίστοιχο ποσοστό του Ευκλείδειου ταξινομητή. Το γεγονός αυτό είναι πιθανό να ευθύνεται στην υπόθεση του Naive Bayes για ανεξαρτησία των χαρακτηριστικών ή και στις τιμές της PDF λόγω μικρών διακυμάνσεων.

```
Success Performance of Naive Bayes method is: 74.59%
```

Εικόνα 15-16.8 Ακρίβεια του ταξινομητή Naive Base

Κατά την εκπαίδευση του ταξινομητή με υπόθεση πως η διακύμανση ισούται με 1 παρατηρείται πως η ακρίβεια βελτιώνεται σε μια τάξη του 7% περίπου (Εικόνα 15-16.10). Αυτό το αποτέλεσμα επιβεβαιώνει τις υποθέσεις μας σχετικά με την επιρροή της μικρής διακύμανσης στον υπολογισμό των πιθανοτήτων και την ταξινόμηση του Naive Bayes.

```
Bayes2 = NaiveBayesClassifier()  
Bayes2 = Bayes2.fit(X_train, y_train, nb_var=1)  
Bayes_score2 = Bayes2.score(X_test, y_test)
```

Εικόνα 15-16.9 Χρήση του μοντέλου Naive Base με διακύμανση 1

Success Performance of Naive Bayes method (for variance equals 1) is: 81.27%

Εικόνα 15-16.10 Ακρίβεια του ταξινομητή Naive Base με διακύμανση 1

Βήμα 17

Ακολουθεί μια σύντομη ανάλυση των ταξινομητών αλλά και διάφορων σημαντικών ορισμών που θα χρησιμοποιηθούν σε αυτό το ερώτημα.

k Nearest Neighbors Classifier (kNN)

Ο kNN είναι ένας μη παραμετρικός ταξινομητής βασισμένος σε παραδείγματα (instance-based). Η αρχή λειτουργίας του είναι η εξής: για ένα νέο δείγμα προς ταξινόμηση, πρώτα υπολογίζονται οι k πλησιέστεροι γείτονές του (στον ν-διάστατο χώρο των χαρακτηριστικών εισόδου) με βάση κάποια συνάρτηση απόστασης, συνήθως ευκλείδεια:

$$d(x, x') = \sqrt{(x_1 - x'_1)^2 + (x_2 - x'_2)^2 + \dots + (x_n - x'_n)^2}$$

Η κλάση του νέου δείγματος θα είναι η κλάση της πλειοψηφίας των k γειτόνων (επιλέγεται k περιπτώ), είτε απλά υπολογισμένη (άθροισμα) είτε (αντίστροφα) ζυγισμένη με βάση την απόσταση του κάθε γείτονα.

Ο kNN δεν έχει πρακτικά κατάσταση εκπαίδευσης. Ωστόσο, ένα νέο δείγμα για να ταξινομηθεί στην test κατάσταση, πρέπει να συγκριθεί η απόστασή του με κάθε δείγμα του train set. Αυτό σημαίνει ότι για την ταξινόμηση είναι απαραίτητα όλα τα δείγματα εκπαίδευσης (εξού και η ονομασία "instance-based", ενώ στον Naive Bayes χρειάζονται μόνο 2 παράμετροι, η μέση τιμή μ και η διακύμανση σ^2). Αυτό σημαίνει ότι ο kNN είναι πιο απαιτητικός και σε χώρο (αποθήκευση όλων των δειγμάτων) και σε χρόνο (υπολογισμός όλων των αποστάσεων για κάθε νέο δείγμα).

Επιλογή υπερπαραμέτρου k

Παρότι Ο kNN είναι ένας μη παραμετρικός ταξινομητής, το k της γειτονιάς του kNN είναι μια υπερ παράμετρος του ταξινομητή. Μια άλλη υπερ παράμετρος για παράδειγμα είναι η συνάρτηση της απόστασης.

Στην περίπτωση του kNN το k ελέγχει το trade-off μεταξύ μεταξύ απόκλισης και διακύμανσης. Εάν τεθεί μικρό k, πχ k=1 προκύπτει ένας ταξινομητής με υψηλή διακύμανση και χαμηλή απόκλιση. Ο ταξινομητής τείνει να αγνοεί τη συνολική κατανομή και αποφασίζει μόνο από το κοντινότερο δείγμα. Στην περίπτωση k=1 το σύνορο απόφασης (decision boundary) περνά από τις μεσοκαθέτους

γειτονικών δειγμάτων διαφορετικής κλάσης. Αντιθέτως, αν τεθεί μεγαλύτερο k , προκύπτει ένας ταξινομητής με χαμηλότερη διακύμανση και υψηλότερη απόκλιση. Θα ταξινομήσει λάθος περισσότερα αποκλίνοντα δείγματα (outliers) αλλά θα σέβεται περισσότερο τη συνολική κατανομή.

Εμπειρικά επιλέγεται συνήθως ως $k < \sqrt{N}$, όπου N το πλήθος των παραδειγμάτων εκπαίδευσης.

Support Vector Machine (SVM)

Το Support Vector Machine (SVM) είναι ένας supervised machine learning αλγόριθμος που χρησιμοποιείται τόσο για ταξινόμηση (classification) όσο και για παλινδρόμηση (regression). Ωστόσο είναι πιο αποδοτικός για ταξινόμηση. Ο στόχος του αλγορίθμου SVM είναι να βρει ένα υπερεπίπεδο σε ένα χώρο N -διάστατων που ταξινομεί ευδιάκριτα τα σημεία δεδομένων. Η διάσταση του υπερεπίπεδου εξαρτάται από τον αριθμό των χαρακτηριστικών. Εάν ο αριθμός των χαρακτηριστικών εισόδου είναι δύο, τότε το υπερεπίπεδο είναι απλώς μια γραμμή. Εάν ο αριθμός των χαρακτηριστικών εισόδου είναι τρία, τότε το υπερεπίπεδο γίνεται δισδιάστατο επίπεδο.

SVM kernel:

Ο πυρήνας SVM είναι μια συνάρτηση που καταλαμβάνει χώρο εισόδου χαμηλών διαστάσεων και τον μετατρέπει σε χώρο υψηλότερης διάστασης, δηλαδή μετατρέπει το μη διαχωρισμό πρόβλημα σε διαχωρίσιμο πρόβλημα. Είναι κυρίως χρήσιμο σε προβλήματα μη γραμμικού διαχωρισμού. Εν ολίγοις ο πυρήνας, κάνει μερικούς εξαιρετικά πολύπλοκους μετασχηματισμούς δεδομένων και, στη συνέχεια, ανακαλύπτει τη διαδικασία διαχωρισμού των δεδομένων με βάση τα labels ή τις εξόδους που ορίζονται.

SVM linear Kernel

Ο linear πυρήνας χρησιμοποιείται όταν τα δεδομένα είναι γραμμικά διαχωρίσιμα (ή σχεδόν γραμμικά διαχωρίσιμα), δηλαδή μπορούν να διαχωριστούν χρησιμοποιώντας μία μόνο γραμμή. Είναι ένας από τους πιο συνηθισμένους πυρήνες που χρησιμοποιούνται. Χρησιμοποιείται κυρίως όταν υπάρχει μεγάλος αριθμός χαρακτηριστικών σε ένα συγκεκριμένο σύνολο δεδομένων. Ένα από τα παραδείγματα όπου υπάρχουν πολλά χαρακτηριστικά, είναι η ταξινόμηση κειμένου, καθώς κάθε αλφάβητο είναι ένα νέο χαρακτηριστικό. Έτσι χρησιμοποιούμε κυρίως linear πυρήνα στην ταξινόμηση κειμένων (text classification).

Πλεονεκτήματα της χρήσης SVM linear Kernel:

- Η εκπαίδευση ενός SVM με γραμμικό πυρήνα είναι ταχύτερη από ό,τι με οποιοδήποτε άλλο πυρήνα.
- Κατά την εκπαίδευση ενός SVM με Γραμμικό Πυρήνα, απαιτείται μόνο η βελτιστοποίηση της παραμέτρου C Regularization. Από την άλλη πλευρά, όταν εκπαιδεύεται με άλλους πυρήνες, υπάρχει ανάγκη βελτιστοποίησης της παραμέτρου γ που σημαίνει ότι η εκτέλεση μιας αναζήτησης πλέγματος συνήθως απαιτεί περισσότερο χρόνο

SVM RBF Kernel

Όταν το σύνολο δεδομένων είναι μη γραμμικό, συνιστάται η χρήση συναρτήσεων πυρήνα όπως το RBF. Για ένα γραμμικά διαχωρίσιμο σύνολο δεδομένων (γραμμικό σύνολο δεδομένων) θα μπορούσε κανείς να χρησιμοποιήσει γραμμική συνάρτηση πυρήνα (kernel = "linear").

Οι πυρήνες RBF είναι η πιο γενικευμένη μορφή των kernels και είναι ένας από τους πιο ευρέως χρησιμοποιούμενους πυρήνες λόγω της ομοιότητάς του με την κατανομή Gauss. Η συνάρτηση πυρήνα RBF για δύο σημεία x_1 και x_2 υπολογίζει την ομοιότητα ή πόσο κοντά είναι μεταξύ τους.

Αυτός ο πυρήνας μπορεί να αναπαρασταθεί μαθηματικά ως εξής:

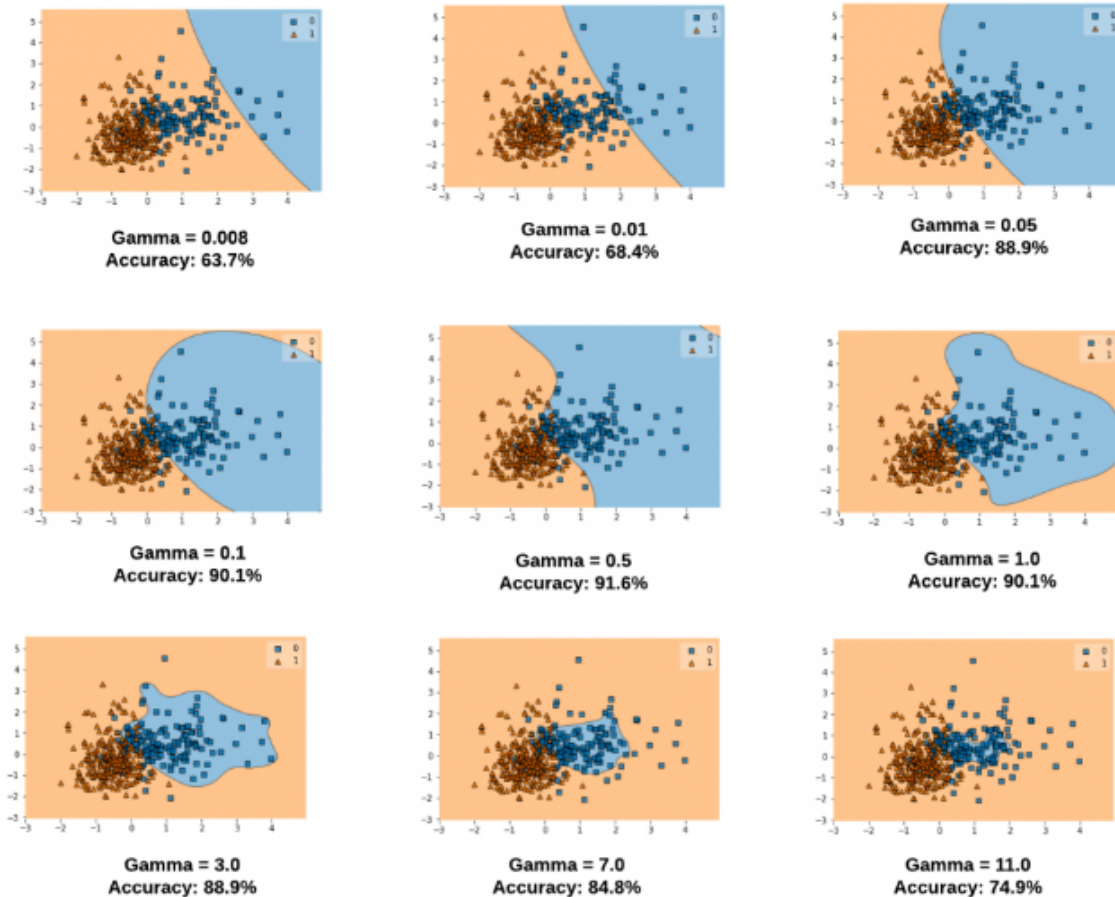
$$K(X_1, X_2) = e^{-\frac{\|X_1 - X_2\|^2}{2\sigma^2}}$$

Sigmoid Kernel:

Προτιμάται κυρίως για νευρωνικά δίκτυα. Αυτή η λειτουργία πυρήνα είναι παρόμοια με ένα μοντέλο perceptron δύο επιπέδων του νευρωνικού δικτύου, το οποίο λειτουργεί ως συνάρτηση ενεργοποίησης για νευρώνες.

Kernel Parameter - Gamma Values:

Η παράμετρος γάμμα καθορίζει πόσο μακριά φτάνει η επιρροή ενός μεμονωμένου training sample, με τις χαμηλές τιμές να σημαίνουν «μακριά» και τις υψηλές τιμές να σημαίνουν «κοντά». Οι χαμηλότερες τιμές γάμμα έχουν ως αποτέλεσμα μοντέλα με χαμηλότερη ακρίβεια και το ίδιο ακριβώς συμβαίνει με τις υψηλότερες τιμές γάμμα. Οι ενδιάμεσες τιμές του γάμμα οδηγούν σε SVM μοντέλο με καλά όρια απόφασης.



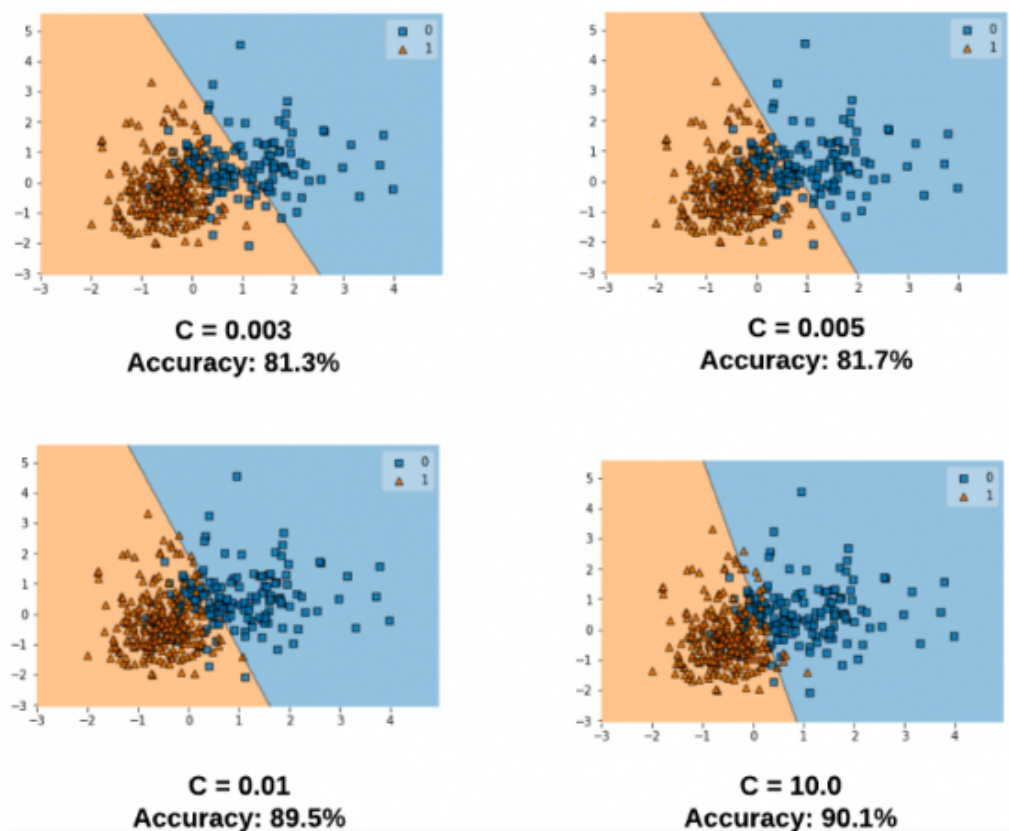
Decision boundaries για διαφορετικές τιμές γάμμα για RBF SVM

Για πολύ μικρές τιμές του γάμμα (0.008 or 0.01) το μοντέλο είναι πολύ περιορισμένο και δεν μπορεί να συλλάβει την πολυπλοκότητα ή το "σχήμα" των δεδομένων. Η περιοχή επιρροής οποιουδήποτε SVM θα περιλαμβάνει ολόκληρο το training set. Το μοντέλο που προκύπτει θα συμπεριφέρεται παρόμοια με ένα γραμμικό μοντέλο.

Για μεγαλύτερες τιμές γάμμα (3.0, 7.0, 11.0) η ακτίνα της περιοχής επιρροής των διανυσμάτων υποστήριξης (SVM) περιλαμβάνει μόνο το ίδιο το SVM και κανένας βαθμός regularization με το C δεν θα μπορεί να αποτρέψει την υπερπροσαρμογή (overfitting).

Kernel Parameter - C Values

Η παράμετρος C είναι μια παράμετρος regularization - κανονικοποίησης που χρησιμοποιείται για τον καθορισμό της ανοχής του μοντέλου ώστε να επιτρέπεται η εσφαλμένη ταξινόμηση των σημείων δεδομένων προκειμένου να επιτευχθεί χαμηλότερο σφάλμα γενίκευσης. Όσο υψηλότερη είναι η τιμή του C, μικρότερη είναι η ανοχή και ο ταξινομητής που εκπαιδεύεται είναι μέγιστου περιθωρίου (maximum-margin). Όσο μικρότερη είναι η τιμή του C, τόσο μεγαλύτερη είναι η ανοχή της εσφαλμένης ταξινόμησης και ο ταξινομητής που εκπαιδεύεται είναι soft-margin που γενικεύει καλύτερα από τον ταξινομητή maximum-margin. Η τιμή C λειτουργεί ως penalty της εσφαλμένης ταξινόμησης. Ωστόσο, μετά από ένα ορισμένο σημείο ($C = 1.0$), η ακρίβεια παύει να αυξάνεται.



Decision boundaries για διαφορετικές τιμές C για Linear Kernel

Στο συγκεκριμένο ερώτημα, χρησιμοποιήθηκαν οι εξής ταξινομητές: Gaussian Naive Bayes, k-Nearest Neighbors Classifier, SVM linear kernel, SVM rbf kernel και SVM sigmoid kernel όπως φαίνεται και στην Εικόνα 17.1 αλλά και τον Πίνακα 17.1

```

classifiers = {}
#Naive Bayes
gnb = GaussianNB()
gnb.fit(X_train, y_train)
classifiers['Naive Bayes'] = gnb.score(X_test, y_test)
#KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5) #default value
knn.fit(X_train, y_train)
classifiers['KNeighborsClassifier'] = knn.score(X_test, y_test)
#SVM linear kernel
svc_model_linear = SVC(C=1.0, random_state=1, kernel='linear', probability=True)
svc_model_linear.fit(X_train, y_train)
classifiers['SVM linear kernel'] = svc_model_linear.score(X_test, y_test)
#SVM RBF kernel
svm_rbf = SVC(kernel='rbf', random_state=1, gamma=0.02, C=1, probability=True)
svm_rbf.fit(X_train, y_train)
classifiers['SVM RBF kernel'] = svm_rbf.score(X_test, y_test)
#SVM sigmoid kernel
svm_sigmoid = SVC(kernel='sigmoid', probability=True)
svm_sigmoid.fit(X_train, y_train)
classifiers['SVM sigmoid kernel'] = svm_sigmoid.score(X_test, y_test)

sorted_accuracy = [(k, classifiers[k]) for k in sorted(classifiers, key=classifiers.get, reverse=True)]
print("Success of each classifier follows below in sorted list:")
for k, v in sorted_accuracy:
    print(k,v)

```

Εικόνα 17.1 Εντολές δημιουργίας και απόδοσης ταξινομητών

Δημιουργήθηκαν 5 διαφορετικά μοντέλα των προαναφερθεισών ταξινομητών, εκπαιδεύτηκαν με τα training data και σε τελικό στάδιο αποθηκεύτηκε σε ένα dictionary *classifiers* το όνομα καθενός ταξινομητή ως key, και ως value το score του στα test data. Μάλιστα, το dictionary *classifiers* ταξινομήθηκε σε φθίνουσα σειρά με βάση το value κάθε στοιχείου, δηλαδή στην προκειμένη περίπτωση το score.

Classifier	Score (%)
Gaussian Naive Bayes	71.95
k Nearest Neighbors Classifier (k=5, default)	94.47
SVM linear kernel	92.62
SVM rbf kernel	94.82
SVM sigmoid kernel	85.05

Πίνακας 17.1 Σύγκριση αποτελεσμάτων επίδοσης ταξινομητών

Παρατήρηση:

Η μεγαλύτερη διαφορά μεταξύ των μοντέλων που κατασκευάστηκαν από την άποψη των features είναι ότι ο Naive Bayes τα αντιμετωπίζει ως ανεξάρτητα, ενώ το SVM εξετάζει τις αλληλεπιδράσεις μεταξύ τους σε έναν ορισμένο βαθμό, εφόσον χρησιμοποιείται μη γραμμικός πυρήνας (rbf, sigmoid κ.λπ.). Επομένως, εάν υπάρχουν αλληλεπιδράσεις και, δεδομένου του προβλήματος, πιθανότατα ένα SVM θα είναι καλύτερο στο να τις καταγράψει, επομένως καλύτερο στην εργασία ταξινόμησης. Επίσης να σημειωθεί πως στα μοντέλα SVM προσαρμόστηκαν καταλλήλως οι παράμετροι C regularization και gamma ώστε να επιτευχθεί το καλύτερο δυνατό score. Τέλος η ίδια διαδικασία

ακολουθήθηκε και για το μοντέλο knn, όπου κατόπιν δοκιμών η παράμετρος k τέθηκε ίση με την default τιμή 5 ($k=5$).

Βήμα 18

Ensemble σημαίνει μια ομάδα στοιχείων που αντιμετωπίζονται ως σύνολο και όχι μεμονωμένα. Μια μέθοδος Ensemble δημιουργεί πολλαπλά μοντέλα και τα συνδυάζει για να το λύσει. Οι μέθοδοι συνόλου βοηθούν στη βελτίωση της ευρωστίας/γενίκευσης του μοντέλου.

Ακολουθούν τα είδη ensembling:

Bagging

Το Bagging, γνωστό και ως bootstrap aggregating, είναι μια μέθοδος ταξινόμησης που στοχεύει στη μείωση της διακύμανσης των εκτιμήσεων υπολογίζοντας τον μέσο όρο πολλαπλών εκτιμήσεων μαζί του ίδιου ταξινομητή. Το Bagging δημιουργεί υποσύνολα από το κύριο σύνολο δεδομένων στο οποίο εκπαιδεύονται οι ταξινομητές. Προκειμένου να συγκεντρωθούν οι προβλέψεις των διαφορετικών ταξινομητών, είτε χρησιμοποιείται ο μέσος όρος για regression, είτε μια προσέγγιση ψηφοφορίας (voting approach) για ταξινόμηση (με βάση την απόφαση της πλειοψηφίας).

Boosting

Οι Boosting αλγόριθμοι είναι ικανοί να λαμβάνουν αδύναμα μοντέλα με χαμηλή απόδοση και να τα μετατρέπουν σε ισχυρά μοντέλα. Η ιδέα πίσω από τους αλγόριθμους ενίσχυσης είναι ότι εκχωρούνται από το χρήστη πολλά αδύναμα μοντέλα μάθησης στα σύνολα δεδομένων και, στη συνέχεια, τα βάρη για τα εσφαλμένα ταξινομημένα παραδείγματα τροποποιούνται κατά τη διάρκεια των επόμενων κύκλων μάθησης. Οι προβλέψεις των ταξινομητών συγκεντρώνονται και στη συνέχεια γίνονται οι τελικές προβλέψεις μέσω σταθμισμένου αθροίσματος στην περίπτωση παλινδρόμησης ή σταθμισμένης πλειοψηφίας (weighted majority vote) στην περίπτωση ταξινόμησης.

Stacking

Οι αλγόριθμοι στοίβαξης (stacking algorithms) είναι μια μέθοδος εκμάθησης συνόλου που συνδυάζει την απόφαση διαφορετικών αλγορίθμων σε εφαρμογές regression ή classification. Τα επιμέρους μοντέλα εκπαιδεύονται σε ολόκληρο το σύνολο δεδομένων εκπαίδευσης. Αφού εκπαιδευτούν τα επιμέρους μοντέλα, συναρμολογείται ένα μετα-μοντέλο από τα διαφορετικά μοντέλα και στη συνέχεια εκπαιδεύεται στις εξόδους των επιμέρους μοντέλων.

Βασικές προσεγγίσεις ensembling:

1. Μέθοδος υπολογισμού μέσου όρου (Averaging method):

Χρησιμοποιείται κυρίως για προβλήματα παλινδρόμησης. Η μέθοδος αποτελείται από την κατασκευή πολλαπλών ανεξάρτητων μεταξύ τους μοντέλων και την απόδοση του μέσου όρου της πρόβλεψης όλων των μοντέλων. Γενικά, η συνδυασμένη έξοδος είναι καλύτερη από μια μεμονωμένη έξοδο επειδή η διακύμανση μειώνεται.

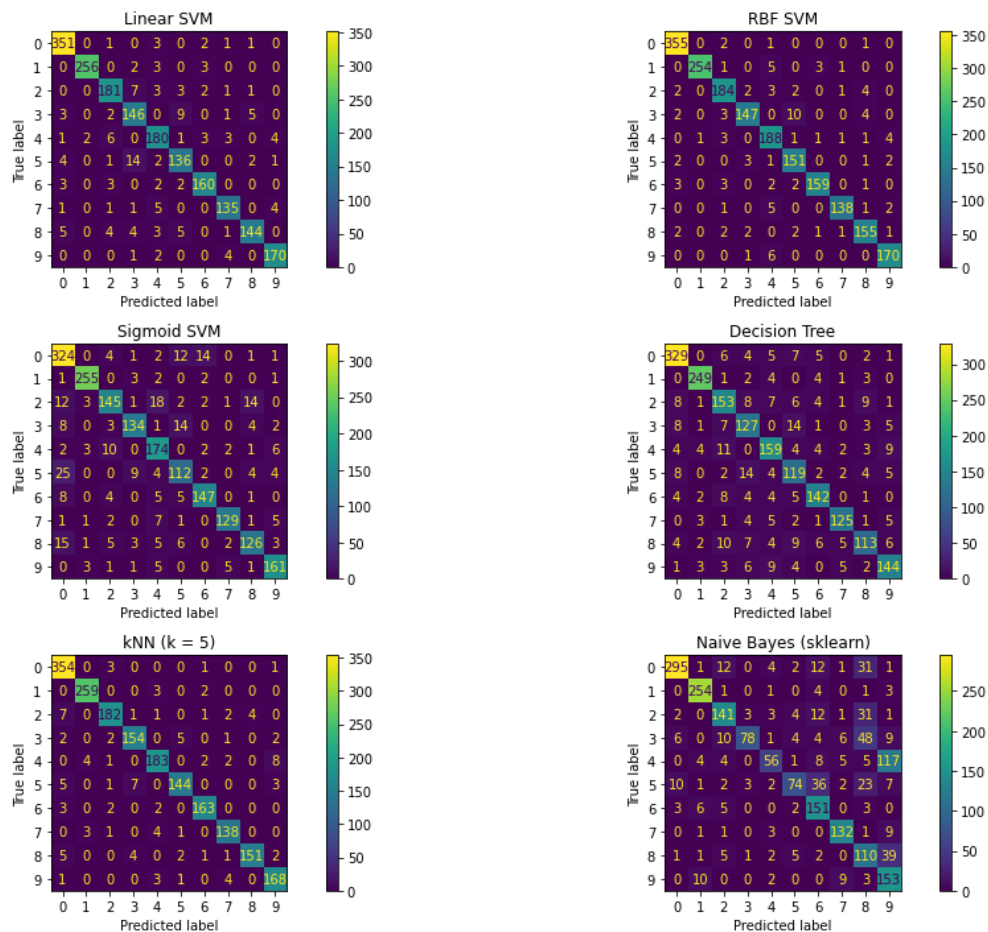
2. Max voting:

Χρησιμοποιείται κυρίως για προβλήματα ταξινόμησης. Η μέθοδος αποτελείται από τη δημιουργία πολλαπλών ανεξάρτητων μεταξύ τους μοντέλων και τη λήψη της ατομικής τους παραγωγής που ονομάζεται «ψήφος» (vote). Η τάξη με τις μέγιστες ψήφους (votes) επιστρέφεται ως αποτέλεσμα.

Συγκεκριμένα, ο VotingClassifier λαμβάνει μια λίστα διαφορετικών εκτιμητών ως ορίσματα και μια μέθοδο ψηφοφορίας. Υπάρχουν δύο διαφορετικοί μέθοδοι - τύποι ταξινομητή ψηφοφορίας για τον συγκεκριμένο. Έχουν ως εξής:

- *Hard voting*, γνωστή και ως *majority voting*, που κάθε μεμονωμένος ταξινομητής ψηφίζει για μια τάξη και η πλειοψηφία κερδίζει.
- *Soft voting*, στην οποία κάθε μεμονωμένος ταξινομητής παρέχει μια τιμή πιθανότητας ότι ένα συγκεκριμένο σημείο δεδομένων ανήκει σε μια συγκεκριμένη κλάση στόχο. Οι προβλέψεις σταθμίζονται με βάση τη σημασία του ταξινομητή και συνοψίζονται. Τότε η ετικέτα στόχος με το μεγαλύτερο άθροισμα σταθμισμένων πιθανοτήτων κερδίζει την ψήφο.

Για την καλύτερη επιλογή των ταξινομητών που θα χρησιμοποιηθούν στον VotingClassifier αναλύθηκε το confusion matrix του καθενός από τους ακόλουθους ταξινομητές: Linear SVM, RBF SVM, Sigmoid SVM, Decision Tree, kNN ($k = 5$), Gaussian Naive Bayes. Όπως διαπιστώθηκε και προηγουμένως από τα αποτελέσματα των scores καλύτερη απόδοση είχαν οι: k Nearest Neighbors Classifier, SVM linear kernel και SVM rbf kernel. Το ίδιο αποτυπώνεται και στην Εικόνα 18.1. Η λογική πίσω από τα confusion matrices, είναι να υπάρχει και η πληροφορία σχετικά με τα ψηφία τα οποία τείνει ο κάθε ταξινομητής να μπερδεύει μεταξύ τους. Συνεπώς θα ήταν λογικό να συνδυαστούν ταξινομητές που εμφανίζουν διαφορές ως προς τα ζεύγη ψηφίων που τείνουν να ταξινομούν εσφαλμένα, ώστε η συνολική απόδοση να βελτιωθεί, διαφορετικά ακόμη και μετά τον συνδυασμό τους θα οδηγούσαν όλοι στη λάθος απόφαση.



Εικόνα 18.1 Confusion matrix κάθε classifier

```

vot_soft = VotingClassifier(estimators = estimator, voting = 'soft')
vot_soft.fit(X_train, y_train)
y_pred1 = vot_soft.predict(X_test)
score1 = accuracy_score(y_test, y_pred1)
print("Soft Voting Score {:.2%}" .format(score1))

vot_hard = VotingClassifier(estimators = estimator, voting = 'hard')
vot_hard.fit(X_train, y_train)
y_pred2 = vot_hard.predict(X_test)
score2 = accuracy_score(y_test, y_pred2)
print("Hard Voting Score {:.2%}" .format(score2))

```

Εικόνα 18.2 Εντολές υλοποίησης VotingClassifier

Κατόπιν της υλοποίησης εντολών για τον Voting Classifier όπως αποτυπώνονται και στην Εικόνα 18.2, το σκορ του αυξήθηκε στο 95.31 % με χρήση soft voting σημειώνοντας αισθητή αύξηση σε σχέση με το προηγούμενο ερώτημα, ενώ με hard voting η απόδοση του classifier άγγιξε μόλις το 94.87% ελάχιστα μεγαλύτερη από την απόδοση του μεμονωμένου SVM με rbf kernel. Συνεπώς όντως παρατηρήθηκε όφελος από τον συνδυασμό των επιμέρους ταξινομητών αυξάνοντας την απόδοση συνολικά (Εικόνα 18.2).

```

Soft Voting Score 95.32%
Hard Voting Score 94.87%

```

Εικόνα 18.3 (α) Αποτελέσματα VotingClassifier

Ακριβώς η ίδια διαδικασία επαναλήφθηκε με μία λίστα 3 διαφορετικών classifiers. Η δεύτερη απόπειρα συμπεριέλαβε τους Gaussian Naive Bayes, SVM sigmoid kernel' και K Nearest Neighbors με $k=5$. Τα αποτελέσματα φαίνονται συγκεντρωμένα στην Εικόνα 18.3 (β).

Soft Voting Score 89.34%
Hard Voting Score 90.28%

Εικόνα 18.3 (β) Αποτελέσματα VotingClassifier

Παρατηρείται πως στην δεύτερη περίπτωση η απόδοση έχει πέσει σημαντικά, όταν μεμονωμένα ο κνη ταξινομητής είχε πετύχει σκορ 94.47%. Συνεπώς έχει μεγάλη σημασία η σωστή επιλογή ταξινομητών. Μάλιστα στη συγκεκριμένη εφαρμογή η μέθοδος hard voting απέδωσε περισσότερο σε σχέση με την soft voting, ενώ στην πρώτη περίπτωση συνέβη το αντίθετο

Bootstrap Aggregation (Bagging)

Ένας ταξινομητής Bagging είναι ένας μετα-εκτιμητής συνόλου που προσαρμόζει έναν από τους βασικούς ταξινομητές σε τυχαία υποσύνολα του αρχικού συνόλου και στη συνέχεια συγκεντρώνει τις μεμονωμένες προβλέψεις τους (είτε με ψηφοφορία είτε με μέσο όρο) για να σχηματίσει μια τελική πρόβλεψη. Ένας τέτοιος μετα-εκτιμητής μπορεί τυπικά να χρησιμοποιηθεί ως τρόπος μείωσης της διακύμανσης ενός black-box εκτιμητή (π.χ. ενός δέντρου αποφάσεων-DecisionTree), εισάγοντας την τυχαιοποίηση στη διαδικασία κατασκευής του και στη συνέχεια δημιουργώντας ένα σύνολο από αυτό.

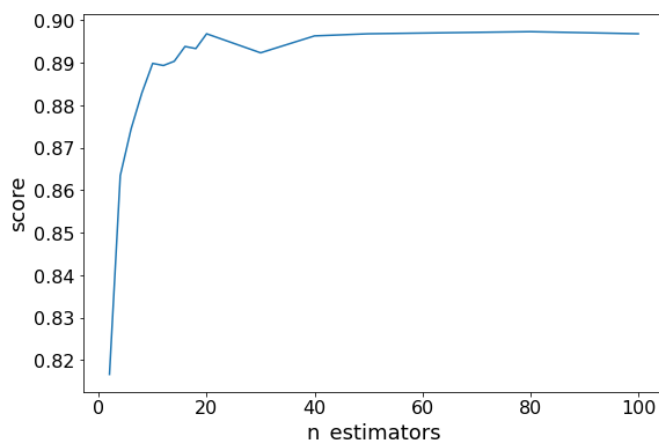
Στο συγκεκριμένο ερώτημα, ο *BaggingClassifier* ($n_estimators=10$) by default παίρνει ως όρισμα του αριθμού των ίδιων εκτιμητών στο σύνολο ensemble το 10. Ωστόσο, δημιουργώντας μία τυχαία λίστα με διάφορους αριθμούς, συμπεριλαμβανομένου και του 10, αναλύθηκε στον default εκτιμητή *DecisionTreeClassifier*, ποιος θα ήταν ο καταλληλότερος αριθμός εκτιμητών με βάση τα training data, όπως φαίνεται και στην Εικόνα 18.3:

```
#Bootstrap Aggregation
estimator_range = [2,4,6,8,10,12,14,16]
models = []
scoring = []
for n_estimators in estimator_range:
    # Create bagging classifier
    clf = BaggingClassifier(n_estimators = n_estimators, random_state = 22)
    # Fit the model
    clf.fit(X_train, y_train)
    # Append the model and score to their respective list
    models.append(clf)
    scoring.append(accuracy_score(y_true = y_test, y_pred = clf.predict(X_test)))
# Generate the plot of scores against number of estimators
plt.figure(figsize=(9,6))
plt.plot(estimator_range, scoring)

plt.xlabel("n_estimators", fontsize = 18)
plt.ylabel("score", fontsize = 18)
plt.tick_params(labelsize = 16)
plt.show()
```

Εικόνα 18.3 Εντολές εύρεσης καταλληλότερου αριθμού εκτιμητών BaggingClassifier

Να σημειωθεί πως ο συνδυασμός αυτής της τεχνικής με Decision Trees δημιουργεί τον ταξινομητή Random Forest. Τελικά, τα αποτελέσματα αποτυπώνονται στην Εικόνα 18.4. Παρατηρείται πως η μέγιστη απόδοση επιτυγχάνεται για $n \approx 24$.



Εικόνα 18.4 Εύρεση καταλληλότερου αριθμού εκτιμητών *BaggingClassifier*

Να σημειωθεί πως έγινε απόπειρα προσομοίωσης και εύρεσης του καταλληλότερου αριθμού εκτιμητών και για του υπόλοιπους 5 classifiers, ωστόσο το πρόγραμμα λόγω περισσότερων υπολογισμών έγινε αρκετά πιο χρονοβόρο με αποτέλεσμα να λάβουμε υπόψη τα αποτελέσματα του default εκτιμητή.

Αφού υλοποιήθηκε για κάθε έναν από τους 5 Classifiers η Bagging μέθοδος βάσει της Εικόνας 18.5, συγκεντρώθηκαν τα αποτελέσματα στον Πίνακα 18.1.

```
bagging = {}
#Naive Bayes
bagging_gnb = BaggingClassifier(base_estimator = gnb,n_estimators =24,random_state = 22)
bagging['Naive Bayes'] = cross_val_score(bagging_gnb, X_train, y_train, cv = 5).mean()
#KNeighborsClassifier
bagging_knn = BaggingClassifier(base_estimator = knn,n_estimators =24,random_state = 22) #default value
bagging['KNeighborsClassifier'] = cross_val_score(bagging_knn, X_train, y_train, cv = 5).mean()
#SVM linear kernel
bagging_linear = BaggingClassifier(base_estimator = SVC(C=1.0, random_state=1, kernel='linear'),n_estimators =24,random_st
bagging['SVM linear kernel'] = cross_val_score(bagging_linear, X_train, y_train, cv = 5).mean()
#SVM RBF kernel
bagging_rbf = BaggingClassifier(base_estimator = SVC(kernel='rbf', random_state=1, gamma=0.02, C=1),n_estimators =24,rand
bagging['SVM RBF kernel'] = cross_val_score(bagging_rbf, X_train, y_train, cv = 5).mean()
#SVM sigmoid kernel
bagging_sigmoid = BaggingClassifier(base_estimator = SVC(kernel='sigmoid'),n_estimators =24,random_state = 22)
bagging['SVM sigmoid kernel'] = cross_val_score(bagging_sigmoid, X_train, y_train, cv = 5).mean()

sorted_accuracy_bagging = [(k, bagging[k]) for k in sorted(bagging, key=bagging.get, reverse=True)]
print("Accuracy of each classifier follows below in sorted list:")
for k, v in sorted_accuracy_bagging:
    print("Model: {} has accuracy: {:.2%}".format(k,v))
```

Εικόνα 18.5 Εντολές Bagging κάθε ταξινομητή

Classifier	Score (%)
Gaussian Naive Bayes	74.87
k Nearest Neighbors Classifier (k=5, default)	96.21
SVM linear kernel	96.30

SVM rbf kernel	96.91
SVM sigmoid kernel	89.56

Πίνακας 18.1 Σύγκριση αποτελεσμάτων επίδοσης ταξινομητών με *Bagging*

Να σημειωθεί πως η μέθοδος υπολογισμού ακρίβειας ήταν η Cross Validation με Kfolds=5. Σε σύγκριση με τα ολικά αποτελέσματα του Πίνακα 17.1 χωρίς Bagging, παρατηρήθηκε αύξηση της απόδοσης με την εφαρμογή του *Bagging Classifier* για κάθε έναν από τους επιμέρους ταξινομητές. Βέβαια η μεταβολή αυτή στους SVM sigmoid και Naive Bayes, ταξινομητές των οποίων η απόδοση ήταν αρκετά χαμηλότερη από τους υπόλοιπους, σημειώθηκαν μεγαλύτερη ποσοστιαία αύξηση απόδοσης, ενώ οι Knn, SVM linear και RBF σημείωσαν μικρότερη.

Βήμα 19

Σε αυτό το βήμα χρησιμοποιήθηκε το αρχείο bonus.py ακριβώς τον ίδιο φάκελο με το προηγούμενο αρχείο ώστε να είναι ορατό το dataset.

Χρειάστηκε η φόρτωση διαφορετικών βιβλιοθηκών σε αυτό το αρχείο που αφορούν κυρίως το Pytorch.

```
import os
import pandas as pd
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.optim as optim
```

Εικόνα 19.1 Εισαγωγή βιβλιοθηκών

Τα αρχεία διαβάζονται με παρόμοιο τρόπο με το Βήμα 1.

```
absolute_path = os.path.dirname(__file__)
data_path = ["test.txt", "train.txt"]
data_folder = "data"
full_path_test = os.path.join(absolute_path, data_folder, data_path[0])
full_path_train = os.path.join(absolute_path, data_folder, data_path[1])

test = pd.read_csv(full_path_test, sep=' ', header=None, index_col=False)
test.dropna(axis=1, how='all', inplace=True)

train = pd.read_csv(full_path_train, sep=' ', header=None, index_col=False)
train.dropna(axis=1, how='all', inplace=True)
```

Εικόνα 19.2 Διάβασμα δεδομένων

Δημιουργήθηκε η κλάση ImageData όπου χειρίζεται τα δεδομένα του dataset και κληρονομεί από την κλάση Dataset. Χρησιμοποιείται στην συνέχεια για την δημιουργία των dataloader του train και test set.

```
class ImageData(Dataset):
    def __init__(self, train_data):
        # all the available data are stored the class
        self.data = train_data.values

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx, :]
```

Εικόνα 19.3 Κλάση χειρισμού Dataset

Στην συνέχεια δημιουργήθηκε η υποκλάση NonLinearActivation που περιέχει ένα νευρωνικό δίκτυο ώστε να είναι εύκολη η χρήση διαφορετικών συναρτήσεων ενεργοποίησης. Κληρονομεί την κλάση nn.Module όπως ζητείται στην εκφώνηση.

```
class NonLinearActivation(nn.Module):
    def __init__(self, in_features, out_features, activation):
        super(NonLinearActivation, self).__init__()
        self.f = nn.Linear(in_features, out_features)
        self.a = activation

    def forward(self, x):
        output = self.f(x)
        return self.a(output)
```

Εικόνα 19.4 Κλάση αυτοματοποίησης χρήσης συναρτήσεων ενεργοποίησης

Την προηγούμενη κλάση χρησιμοποιεί η κλάση MyNNNetwork όπου υλοποιεί πλήρως ένα νευρωνικό δίκτυο και χειρίζεται τον αριθμό των νευρώνων και τον αριθμό των layers. Επίσης, αυτή είναι η κλάση όπου θα ορίσει ποια συνάρτηση ενεργοποίησης θα χρησιμοποιηθεί.

```
class MyNNNetwork(nn.Module):
    def __init__(self, layers, n_features, n_classes, activation):
        super(MyNNNetwork, self).__init__()
        layers_in = [n_features] + layers
        layers_out = layers + [n_classes]
        self.f = nn.Sequential(*[
            NonLinearActivation(in_feats, out_feats, activation=activation)
            for in_feats, out_feats in zip(layers_in, layers_out)
        ])

    def forward(self, x):
        y = self.f(x)
        return y
```

Εικόνα 19.5 Κλάση υλοποίησης νευρωνικού

Οι παρακάτω μεταβλητές επιτρέπουν τον πειραματισμό στα epochs του νευρωνικού στο batch size του DataLoader και στο learning rate.

```
EPOCHS = 20
# the mini-batch size
BATCH_SZ = 64
learning_rate = 1e-5
```

Εικόνα 19.6 Παράμετροι εκπαίδευσης

Ο κώδικας πειραματισμού με τις πιο γνωστές συναρτήσεις ενεργοποίησης (ReLU, Sigmoid και tanh) και διάφορα layers και πλήθος νευρώνων.

```
# Test different non linear activate functions
net = MyNNNetwork([500, 20], train.shape[1] - 1, 10, nn.Sigmoid())
# net = MyNNNetwork([50, 100, 20], train.shape[1] - 1, 10, nn.ReLU())
# net = MyNNNetwork([80, 400], train.shape[1]-1, 10, nn.Tanh())
print(f"The network architecture is: \n {net}")
```

Εικόνα 19.7 Διαφορετικά νευρωνικά δίκτυα

Τα αποτελέσματα των πειραματισμών φαίνονται στις επόμενες 3 εικόνες.

```
The network architecture is:
MyNNNetwork(
  (f): Sequential(
    (0): NonLinearActivation(
      (f): Linear(in_features=256, out_features=500, bias=True)
      (a): Sigmoid()
    )
    (1): NonLinearActivation(
      (f): Linear(in_features=500, out_features=20, bias=True)
      (a): Sigmoid()
    )
    (2): NonLinearActivation(
      (f): Linear(in_features=20, out_features=10, bias=True)
      (a): Sigmoid()
    )
  )
)
```

Εικόνα 19.8 Εικόνα μοντέλου (α)


```

The network architecture is:
MyNNNetwork(
  (f): Sequential(
    (0): NoneLinearActivation(
      (f): Linear(in_features=256, out_features=50, bias=True)
      (a): ReLU()
    )
    (1): NoneLinearActivation(
      (f): Linear(in_features=50, out_features=100, bias=True)
      (a): ReLU()
    )
    (2): NoneLinearActivation(
      (f): Linear(in_features=100, out_features=20, bias=True)
      (a): ReLU()
    )
    (3): NoneLinearActivation(
      (f): Linear(in_features=20, out_features=10, bias=True)
      (a): ReLU()
    )
  )
)
)

```

Εικόνα 19.8 Εικόνα μοντέλου (β)

```

The network architecture is:
MyNNNetwork(
  (f): Sequential(
    (0): NoneLinearActivation(
      (f): Linear(in_features=256, out_features=80, bias=True)
      (a): Tanh()
    )
    (1): NoneLinearActivation(
      (f): Linear(in_features=80, out_features=400, bias=True)
      (a): Tanh()
    )
    (2): NoneLinearActivation(
      (f): Linear(in_features=400, out_features=10, bias=True)
      (a): Tanh()
    )
  )
)
)

```

Εικόνα 19.8 Εικόνα μοντέλου (γ)

Στην συνέχεια ορίζεται το loss function, ο optimizer με gradient descent, αρχικοποιούνται τα δεδομένα εκπαίδευσης και ελέγχου ως αντικείμενα της τάξης ImageData και τέλος δημιουργούνται 2 DataLoaders με τα αντίστοιχα αντικείμενα.

```
# define the loss function
criterion = nn.CrossEntropyLoss()

# define the optimizer

optimizer = optim.SGD(net.parameters(), lr=learning_rate)

train_set = ImageData(train)
test_set = ImageData(test)

train_dl = DataLoader(train_set, batch_size=BATCH_SZ, shuffle=True)
test_dl = DataLoader(test_set, batch_size=BATCH_SZ, shuffle=True)
```

Εικόνα 19.9 Αρχικοποίηση σημαντικών αντικειμένων

Το επόμενο κομμάτι κώδικα αφορά την εκπαίδευση του νευρωνικού ανά batches και ανά epochs.

```
net.train()
# for each epoch
for epoch in range(EPOCHS):
    running_average_loss = 0
    # for every batch
    for i, data in enumerate(train_dl):
        all = data
        # get the features and labels
        X_batch, y_batch = all[:, 1:], all[:, 0]
        optimizer.zero_grad()
        out = net(X_batch)
        loss = criterion(out, y_batch)
        loss.backward()
        # update weights
        optimizer.step()

    running_average_loss += loss.detach().item()
    if i % 100 == 0:
        print("Epoch: {} \t Batch: {} \t Loss {}".format(epoch, i, float(running_average_loss) / (i + 1)))
```

Εικόνα 19.10 Εκπαίδευση νευρωνικού

Επιπλέον, δημιουργήθηκε κώδικας για τον έλεγχο της εκπαίδευσης του μοντέλου.

```

# turns off dropout for testing
net.eval()
acc = 0
n_samples = 0
with torch.no_grad():
    for i, data in enumerate(test_dl):
        all = data
        # test data and labels
        X_batch, y_batch = all[:, 1:], all[:, 0]
        # get net's predictions
        out = net(X_batch)
        val, y_pred = out.max(1)
        # get accuracy
        acc += (y_batch == y_pred).sum().detach().item()
        n_samples += X_batch.size(0)

# Print accuracy
print(acc)
print(n_samples)
print(acc / n_samples)

```

Εικόνα 19.11 Έλεγχος ακρίβειας νευρωνικού

Δυστυχώς, δεν καταφέραμε να εκτελέσουμε την εκπαίδευση ώστε να ελέγξουμε και την ακρίβεια που προκύπτει στα νευρωνικά που δημιουργήσαμε ούτε να πειραματιστούμε με τις υπόλοιπες παραμέτρους (learning rate, epochs, batch size, activation functions, layers, αριθμό νευρώνων).

ΒΙΒΛΙΟΓΡΑΦΙΑ

- 1) <https://www.geeksforgeeks.org/reduce-data-dimensionality-using-pca-python/>
- 2) <https://dzone.com/articles/using-jsonb-in-postgresql-how-to-effectively-store-1>
- 3) <https://gist.github.com/WittmannF/60680723ed8dd0cb993051a7448f7805>
- 4) <https://www.geeksforgeeks.org/creating-linear-kernel-svm-in-python/>
- 5) <https://towardsdatascience.com/radial-basis-function-rbf-kernel-the-go-to-kernel-acf0d22c798a>
- 6) <https://www.geeksforgeeks.org/ensemble-methods-in-python/>
- 7) <https://stackabuse.com/ensemble-voting-classification-in-python-with-scikit-learn/>
- 8) <https://vitalflux.com/hard-vs-soft-voting-classifier-python-example/>
- 9) https://www.w3schools.com/python/python_ml_bagging.asp