Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Αναγνώριση Προτύπων

Εξάμηνο 9ο (Εαρινό Εξάμηνο 2022-2023)

1η Εργαστηριακή Άσκηση

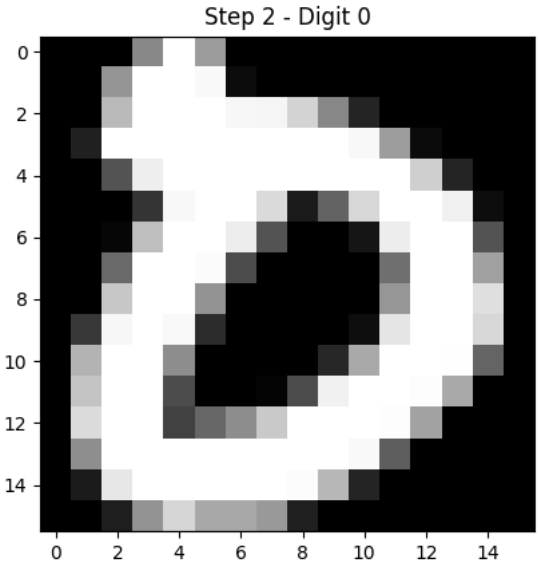
Χάιδος Νικόλαος el18096, Σπανός Νικόλαος el18822

Βήμα 1

Φορτώσαμε τα δεδομένα χρησιμοποιώντας την συνάρτηση $np.loadtxt$, και κρατήσαμε μόνο την πρώτη στήλη από τα train και test, για να τα χωρίσουμε σε features και σε labels.

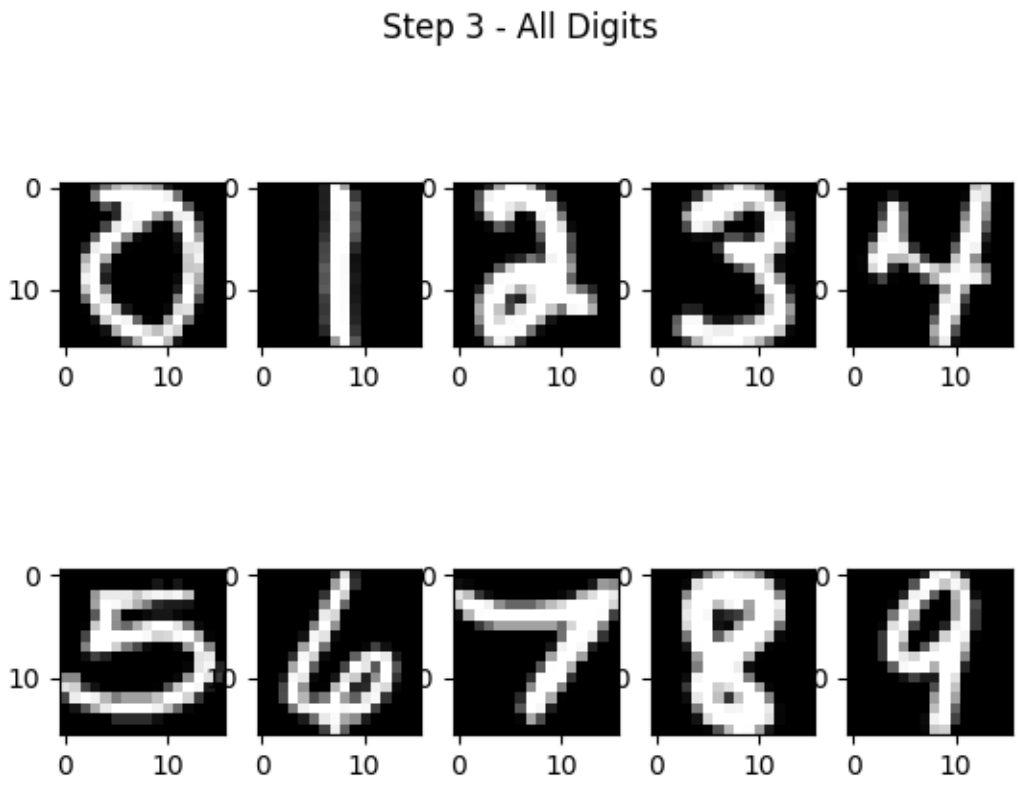
Βήμα 2

Από τον πίνακα train, πήραμε το sample νούμερο 131, το οποίο (μετά από reshape σε $16x16$) είναι το εξής:



Βήμα 3

Με την ίδια διαδικασία, σχεδιάσαμε ένα τυχαίο δείγμα για κάθε ένα από τα δέκα ψηφία (η τυχαία επιλογή έγινε με την συνάρτηση $np.random.choice$ στα επιμέρους κομμάτια του πίνακα train):



Βήμα 4

Χρησιμοποιώντας την συνάρτηση $np.mean$, πήραμε τον μέσο όρο από όλα τα pixel στην θέση (10,10) των ψηφίων 0. Το τελικό νούμερο είναι ίσο με $-\mathbf{0.927264}$.

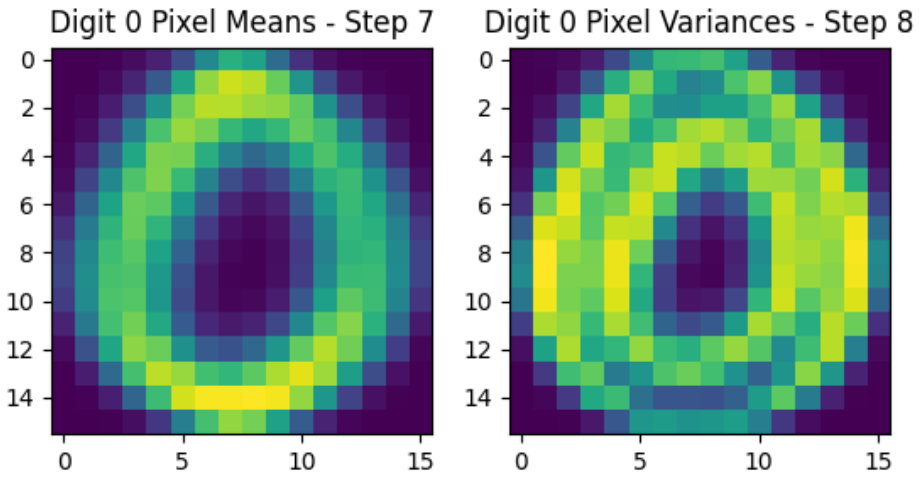
Βήμα 5

Χρησιμοποιώντας την συνάρτηση $np.var$, πήραμε την διασπορά από όλα τα pixel στην θέση (10,10) των ψηφίων 0. Το τελικό νούμερο είναι ίσο με $-\mathbf{0.083924}$.

Βήμα 6-8

Εφαρμόζοντας τις παραπάνω συναρτήσεις σε όλα τα pixel, για όλα τα ψηφία 0, βρίσκουμε τις συνολικές μέσες τιμές και διασπορές.

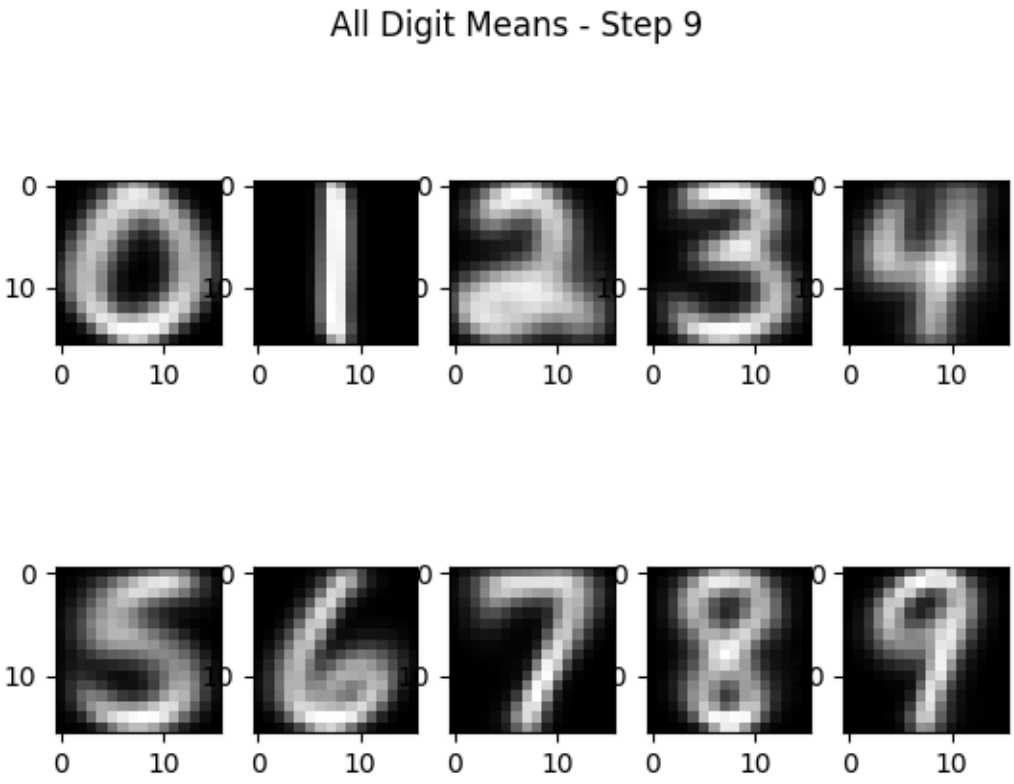
Κάνοντας reshape τις παραπάνω μέσες τιμές και διασπορές που βρήκαμε, παίρνουμε τα εξής διαγράμματα:



Παρατηρούμε ότι οι μέσες τιμές των pixel δίνουν μία καλή προσέγγιση του ψηφίου 0. Παράλληλα, στην σχεδίαση των διασπορών, μπορούμε να δούμε τα pixel τα οποία παρουσιάζουν τις πιο έντονες διαφορές. Σε αυτήν την περίπτωση, όπως είναι και αναμενόμενο, η μεγαλύτερη διασπορά βρίσκεται στα “τοιχώματα” του ψηφίου 0, ενώ στο κέντρο ή στις γωνίες, η διασπορά είναι σχεδόν μηδενική (αφού αυτός ο χώρος αφήνεται συνήθως κενός).

Βήμα 9

Επεκτείνουμε την παραπάνω διαδικασία σε όλα τα ψηφία. Σχεδιάζοντας τις προσεγγίσεις των ψηφίων από τις μέσες τιμές που βρήκαμε, έχουμε τα εξής αποτελέσματα:



Βήμα 10

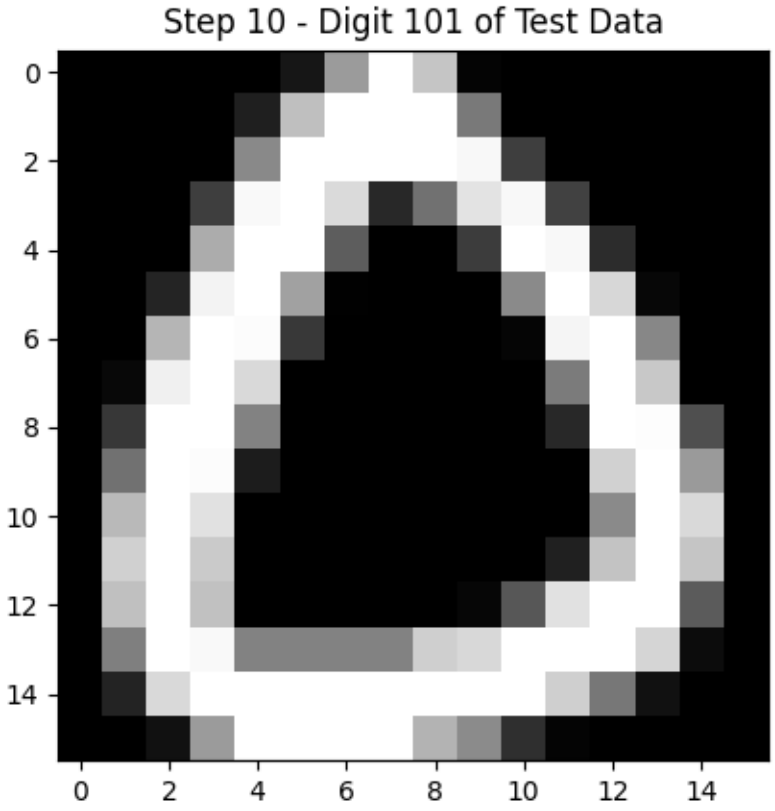
Για την ταξινόμηση του sample νούμερο 101, θα βρούμε την ευκλείδεια απόστασή του από όλα τα Class Means που υπολογίσαμε στο προηγούμενο βήμα. Αυτό μπορεί να γίνει με τις εξής δύο εντολές:

$$\boldsymbol{dist = np.linalg.norm(final\_means - digit, axis = 1, ord = 2)}$$

$$\boldsymbol{pred\_class = np.argmin(dist)}$$

Η πρώτη εντολή θα μάς δώσει την L2 Νόρμα, του τελικού πίνακα των αποστάσεων του sample με τα Class Means. Η L2 Νόρμα ταυτίζεται με την Ευκλείδεια Απόσταση, και με την δεύτερη εντολή, υπολογίζουμε το Class Index που αντιστοιχεί στην μικρότερη απόσταση, το οποίο αντιστοιχεί στο προβλεπόμενο ψηφίο.

Συγκεκριμένα, αν σχεδιάσουμε το sample 101, αυτό δείχνει ως εξής:



Ο ταξινομητής Ευκλείδειας Απόστασης που εξηγήσαμε προηγουμένως, ταξινομεί επιτυχώς το παραπάνω sample στην κλάση 0.

Βήμα 11

Ακολουθώντας την ίδια διαδικασία ταξινομούμε όλα τα samples του Test Set, με βάση την Ευκλείδεια Απόσταση, σε μία από τις δέκα κλάσεις. Ύστερα, υπολογίζουμε το τελικό Accuracy που πέτυχε ο ταξινομητής μας, ως εξής:

$$Accuracy = \frac{Συνολικά\ Σωστά\ Ταξινομημένα\ Samples}{Συνολικά\ Samples}$$

Το τελικό Accuracy του ταξινομητή μας είναι ίσο με $\mathbf{81.415\%}$

Βήμα 12

Η βιβλιοθήκη Scikit-Learn, ζητάει οι ταξινομητές να είναι κλάσεις, που να υλοποιούν τις συναρτήσεις $fit, predict\ και\ score$. Ενσωματώνοντας τις παραπάνω συναρτήσεις και διαδικασίες, δημιουργήσαμε την εξής κλάση (συμβατή με την sklearn):

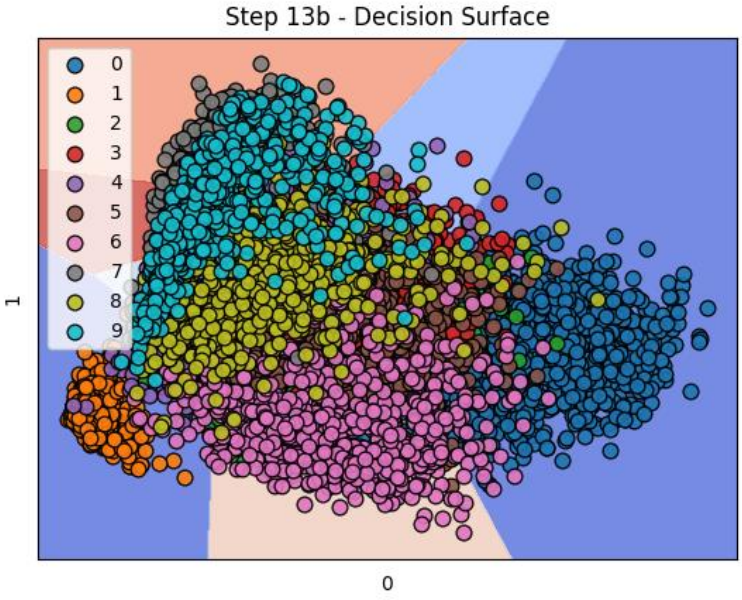
```
#-Creating Euclidean Distance Classifier in accordance to sklearn-#
def euclidean_distance_classifier(X, X_mean):
    final_preds = np.zeros(shape = (X.shape[0],))

    for i in range(X.shape[0]):
      dists = np.linalg.norm(X_mean - X[i], axis = 1, ord = 2)
      final_preds[i] = np.argmin(dists)

    return final_preds


class EuclideanDistanceClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self):
        self.X_mean_ = None


    def fit(self, X, y):
        self.X_mean_ = np.zeros(shape = (10,X.shape[1]))
        for i in range(0,10):
            temp = X[y == i,:]
            self.X_mean_[i] = np.mean(temp, axis = 0)

        return self

    def predict(self, X):
        return euclidean_distance_classifier(X, self.X_mean_)


    def score(self, X, y):
        acc = (self.predict(X) == y).sum() / (y.shape[0])
        return acc
```

Βήμα 13

a) Εφόσον ο ταξινομητής είναι συμβατός με την sklearn, μπορούμε να χρησιμοποιήσουμε απευθείας την συνάρτηση $cross\_val\_score$, που υπολογίζει το K-Fold Cross Validation Score του classifier. Συγκεκριμένα, για τον Ευκλείδειο Ταξινομητή που φτιάξαμε στο προηγούμενο βήμα, το 5-Fold Cross Validation Accuracy Score στα δεδομένα $train$ είναι $\mathbf{84.858}\%$. Παρατηρούμε ότι το score που πετυχαίνει εδώ είναι μεγαλύτερο από το score στο $Test\ Set$, όπως και είναι αναμενόμενο, λόγω της διαφοράς του όγκου $Train\ και\ Test\ Set$, αλλά και επειδή το $CrossValScore$ αποτελεί μέσο όρο 5 διαφορετικών evaluations του μοντέλου.

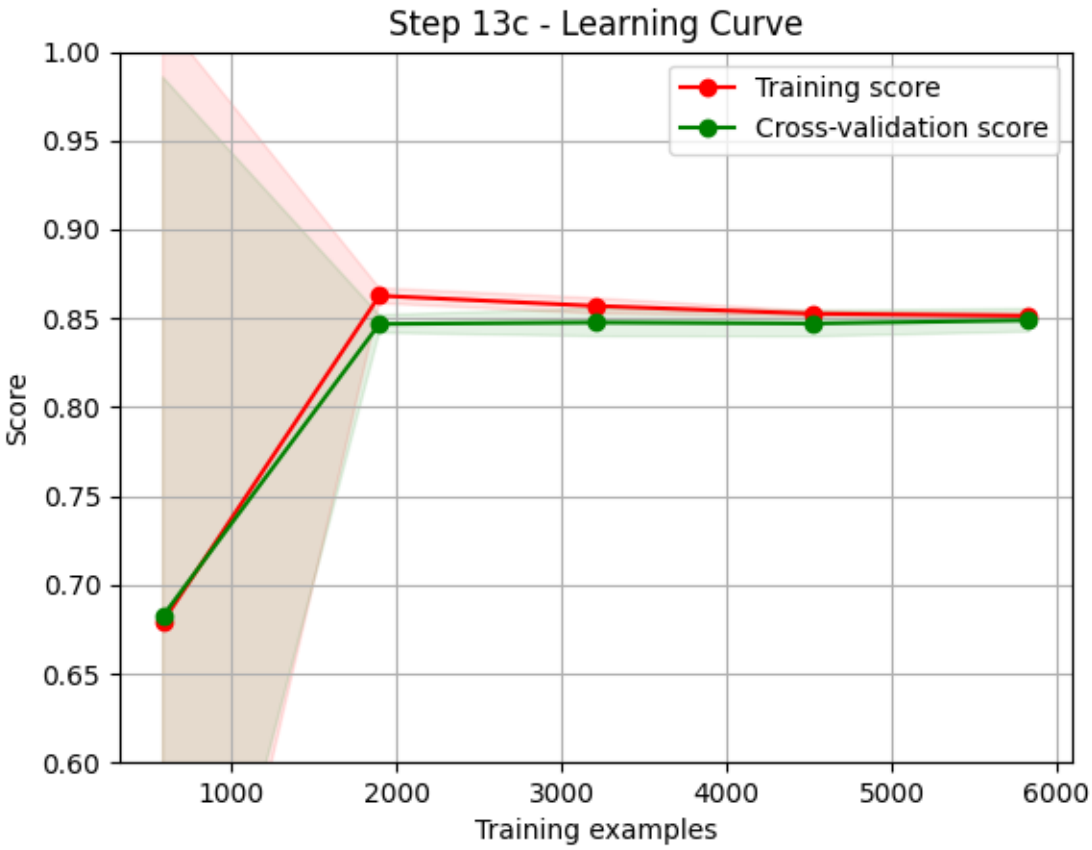
b) Για να σχεδιάσουμε ένα 2-D διάγραμμα της περιοχής απόφασης του ταξινομητή, πρέπει πρώτα να μειώσουμε την διαστατικότητα των δεδομένων από 256 σε 2. Για αυτόν τον σκοπό, χρησιμοποιούμε την τεχνική του PCA (υλοποιημένη στο sklearn), η οποία θα μας δώσει δύο συνδυασμούς των 256 διαστάσεων, οι οποίοι έχουν την μεγαλύτερη διασπορά στα δεδομένα (άρα "συγκεντρώνουν" και την περισσότερη πληροφορία).

Αφού κάνουμε την μείωση διαστατικότητας στα δεδομένα, και εκπαιδεύσουμε τον ταξινομητή στα δεδομένα με τα 2 features, μπορούμε να σχεδιάσουμε την περιοχή απόφασης του ταξινομητή σε 2-D διάγραμμα, ως εξής:



Βλέπουμε ότι οι διαφορετικές κλάσεις, σχηματίζουν διακριτά clusters στις αντίστοιχες περιοχές των 2 νέων features, αλλά βλέπουμε ότι υπάρχει και μία μικρή επικάλυψη, άρα και λανθασμένα προβλεπόμενα samples.

c) Για το $Learning\ Curve$ χρησιμοποιούμε την συνάρτηση $learning\_curve$ της sklearn, η οποία παίρνει ως είσοδο τον ταξινομητή, τα δεδομένα και τα Folds για το Cross-Validation. Η τελική καμπύλη εκμάθησης είναι η εξής:



Μπορούμε να δούμε ότι, ο ταξινομητής ξεκινάει να "μαθαίνει" το συγκεκριμένο task, όταν το $Train\ Set$ φτάνει περίπου τα 2000 samples. Από εκείνη την στιγμή και μετά, το score που πετυχαίνει σταθεροποιείται περίπου στο $\mathbf{86\%}$.

Βήμα 14

Ο Naive Bayes Classifier λειτουργεί υπολογίζοντας την πιθανότητα δεδομένου ενός δείγματος x, το x αυτό να βρίσκεται στην κάθε κλάση της ταξινόμησης. Πιο συγκεκριμένα, χρησιμοποιείται για τον υπολογισμό της πιθανότητας το θεώρημα του Bayes:

$$p(\omega_i|x) = \frac{p(\omega_i)p(x|\omega_i)}{p(x)}$$

Ως πρώτο βήμα, υπολογίζουμε τις a-priori πιθανότητες $p(\omega_i)$ για τις κλάσεις, ως:

$$p(\omega_i) = \frac{\#Δειγμάτων\ της\ κλάσης\ \omega_i}{\#Σύνολο\ Δειγμάτων}$$

Ο υπολογισμός έγινε χρησιμοποιώντας την *np.unique*, η οποία επιστρέφει το πλήθος των δειγμάτων για κάθε ξεχωριστή κλάση και διαιρώντας με το συνολικό πλήθος δειγμάτων.

Βήμα 15

Όπως και στο βήμα 12, χρειάζεται να υλοποιήσουμε τον Custom Naive Bayes Classifier σε μορφή συμβατή με την βιβλιοθήκη scikit-learn, δηλαδή να διαθέτει διαδικασίες *fit*,*predict* και *score*.

- **Fit:**
  Η συνάρτηση *fit* υλοποιήθηκε ως εξής:

```
def fit(self, X, y):
    self.final_means = np.zeros(shape = (10,X.shape[1]))
    self.final_vars = np.zeros(shape = (10,X.shape[1]))

    for i in range(0,10):
        temp = X[y == i,:]
        #-Calculating Means and Variances according to data for fitting-#
        self.final_means[i] = np.mean(temp, axis = 0)
        self.final_vars[i] = np.var(temp, axis = 0)

    if self.use_unit_variance:
        self.final_vars = np.ones(self.final_vars.shape)

    self.priors = np.unique(y.astype(int), return_counts=True)[1] / y.shape[0]


    return self
```

Βάσει των δεδομένων εισόδου, υπολογίζονται οι μέσες τιμές και οι διακυμάνσεις για κάθε κλάση, καθώς και οι a-priori πιθανότητες που θα χρειαστούν στο *predict*. Ο Custom Naive Bayes υποστηρίζει και λειτουργία unit_variance, στην οποία ο πίνακας διακυμάνσεων αντί να δημιουργείται από τα δεδομένα, τίθεται ως μοναδιαίος.

- **Predict:**
  Η συνάρτηση *predict* υλοποιήθηκε ως εξής:

```
def predict(self, X):
    posterior = np.empty((X.shape[0],10))

    #-Calculating Mutlivariate Normal Distribution for each pixel of all digits for posterior probabilities-#
    for digit in range(10):

        #-Calculating Posterior Probabilities by multiplying distribution with priors-#
        posterior[:, digit] = scipy.stats.multivariate_normal(self.final_means[digit,:], self.final_vars[digit,:],
                                                              allow_singular=True).pdf(X[:, :]) * self.priors[digit]

    #-Selecting maximum digit probability for final prediction-#
    preds = np.argmax(posterior, axis=1)

    return preds
```

Ο Naive Bayes θεωρεί ότι τα επιμέρους στοιχεία των samples είναι ανεξάρτητα μεταξύ τους, οπότε με απλό πολλαπλασιασμό μπορούμε να υπολογίσουμε τις a-posteriori πιθανότητες. Επίσης, θεωρείται ότι το κάθε pixel του κάθε ψηφίου ακολουθεί κανονική κατανομή με μέση τιμή και διασπορά τις υπολογιζόμενες από την συνάρτηση *fit*. Υλοποιούμε τις κατανομές με χρήση της *multivariate_normal* και πολλαπλασιάζοντας με τις a-priori πιθανότητες κάθε ψηφίου, υπολογίζουμε τις a-posteriori. Για την συνάρτηση *multivariate_normal* η παράμετρος *allow_singular* τέθηκε true ώστε να μπορεί να υποστηρίζει διακυμάνσεις τιμής 0 (οι οποίες δίνουν τιμή πιθανότητας 1). Επίσης, ως είσοδος δίνεται ένα vector από διακυμάνσεις, τον οποίο η συνάρτηση μετατρέπει από μόνη της σε διαγώνιο πίνακα (κάνοντας την υπόθεση ανεξάρτητων features). Τέλος, χρησιμοποιώντας την συνάρτηση argmax παίρνουμε την προβλεπόμενη κλάση για το αντίστοιχο ψηφίο, βάσει της μεγαλύτερης a-posteriori πιθανότητας.

- **Score:**
  Η *score* υλοποιήθηκε όπως και σε προηγούμενα βήματα:

```
def score(self, X, y):
    preds = self.predict(X)
    acc = (self.predict(X) == y).sum() / (y.shape[0])

    return acc
```

Υπολογίζοντας το score του Custom Naive Bayes στο test set πήραμε αποτέλεσμα $\mathbf{77.479}\%$.

Η έτοιμη ρουτίνα της scikit-learn έδωσε αποτέλεσμα $\mathbf{71.948}\%$. Η custom υλοποίηση λειτούργησε καλύτερα από την έτοιμη, καθώς η υλοποίηση της scikit-learn χρησιμοποιεί μια μη-μηδενική τιμή για να κάνει smoothing τον πίνακα διακυμάνσεων, ενώ η custom μέθοδος υποστηρίζει και μηδενικές διακυμάνσεις.

Βήμα 16

Υπολογίζοντας ξανά το score, αλλά αυτήν την φορά χρησιμοποιώντας μοναδιαίο πίνακα διακυμάνσεων, πήραμε αποτέλεσμα $\mathbf{81.266}\%$. Το γεγονός ότι η απόδοση του classifier είναι καλύτερη με unit variance μπορεί να οφείλεται στο ότι οι τιμές διακύμανσης που υπολογίζονται από τα pixel δεν αντιπροσωπεύουν σωστά τα ψηφία από τα δεδομένα, με αποτέλεσμα να οδηγούνται, λόγω της κατανομής, σε λανθασμένα αποτελέσματα. Σε περίπτωση που διαθέταμε περισσότερα δεδομένα, οι διακυμάνσεις μπορεί να σύγκλιναν καλύτερα στις πραγματικές τιμές, σε σχέση με αυτές που υπολογίζονται.

Βήμα 17

Για την σύγκριση των διαφορετικών ταξινομητών χρησιμοποιήσαμε τις έτοιμες υλοποιήσεις των *Naive Bayes*, *Nearest Neighbors* και *SVM* και πήραμε τα εξής αποτελέσματα:

Classifier	Score
SKLearn Naive Bayes	71.948 %
SKLearn 3-NN	94.469 %
SKLearn 5-NN	94.469 %
SKLearn 7-NN	94.170 %
SKLearn SVM (Linear)	92.626 %
SKLearn SVM (Poly)	95.366 %
SKLearn SVM (RBF)	94.718 %

Από όλους τους ταξινομητές παρατηρούμε ότι ο Naive Bayes έχει την χειρότερη απόδοση ενώ ο K-NN και ο SVM έχουν παρόμοιες αποδόσεις, με τον SVM να είναι ελαφρώς καλύτερος με πολυωνυμικό πυρήνα.

Βήμα 18

Με σκοπό να βελτιώσουμε την απόδοση των ταξινομητών, χρησιμοποιήσαμε δύο ensembling μεθόδους, την VotingClassifier και την BaggingClassifier.

- **Voting Classifier:**
  Για την μέθοδο αυτή χρησιμοποιήσαμε hard voting, καθώς για soft voting πρέπει όλοι οι ταξινομητές να υπολογίζουν πιθανότητες συμμετοχής σε κλάση, ενώ ο Custom Naive Bayes δεν έχει τέτοια δυνατότητα. Χρησιμοποιώντας τους ταξινομητές *Custom Naive Bayes, SKLearn Naive Bayes, 3-NN, 5-NN* και *PolySVM* υπολογίσαμε score $\mathbf{94.569}\%$. Το score βγήκε μικρότερο, καθώς η χρήση περισσότερων ταξινομητών μαζί με τον ήδη αποδοτικό *PolySVM*, οδήγησε σε σφάλματα, λόγω της χαμηλότερης απόδοσης των υπολοίπων.

- **Bagging Classifier:**
  Για την μέθοδο αυτή χρησιμοποιήσαμε των Custom Naive Bayes για να παρατηρήσουμε αν μπορεί να βελτιωθεί η αποδοσή του. Χρησιμοποιώντας αριθμό estimators ίσο με 20, πετύχαμε score $\mathbf{78.376}\%$, δηλαδή αύξηση της τάξης του $\mathbf{1}\%$.

Βήμα 19

Για να μπορέσουμε να επεξεργαστούμε τα δεδομένα με την βιβλιοθήκη *pytorch*, τα φορτώσαμε σε μορφή *Pytorch Dataloader* με μέγεθος batch ίσο με 16 samples και για τα 3 set (train, test, validation). Το validation set αποτελεί το 20% των δειγμάτων του train set.

Για το νευρωνικό χρησιμοποιήθηκαν 3 hidden layers (300, 200, 100), συνάρτηση ενεργοποίησης ReLU για τα hidden layers και Softmax για το output layer (ώστε να παίρνουμε κατανομή πιθανοτήτων). Παράλληλα, για την εκπαίδευση χρησιμοποιήσαμε optimizer AdamW, CrossEntropyLoss και scheduler LambdaLR με λάμδα 0.95 για το learning rate. Στην διάρκεια μίας εποχής κάναμε update το running_loss ανά 100 batches (τόσο για το train όσο και για το validation loss). Για 30 εποχές, υπολογίζοντας το train loss και το validation loss παρατηρήσαμε χαμηλές τιμές, δηλαδή δεν είχαμε overfitting στην εκπαίδευση του μοντέλου. Ενδεικτικά, παρουσιάζονται παρακάτω οι 10 τελευταίες εποχές:

```
EPOCH 20, Train Loss 1.485, Validation Loss 1.532
EPOCH 21, Train Loss 1.501, Validation Loss 1.516
EPOCH 22, Train Loss 1.487, Validation Loss 1.519
EPOCH 23, Train Loss 1.499, Validation Loss 1.497
EPOCH 24, Train Loss 1.482, Validation Loss 1.533
EPOCH 25, Train Loss 1.492, Validation Loss 1.504
EPOCH 26, Train Loss 1.489, Validation Loss 1.496
EPOCH 27, Train Loss 1.485, Validation Loss 1.495
EPOCH 28, Train Loss 1.484, Validation Loss 1.494
EPOCH 29, Train Loss 1.495, Validation Loss 1.512
EPOCH 30, Train Loss 1.481, Validation Loss 1.494
```

Τελικά, υπολογίσαμε score για το μοντέλο στο test set ίσο με $\mathbf{93.124}\%$.