Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Χειμερινό εξάμηνο 2022-23

Αναγνώριση Προτύπων

1η Εργαστηριακή Άσκηση: Οπτική Αναγνώριση Ψηφίων

Δημήτριος Κοκκίνης 03118896

Χριστίνα Ρεντίφη 03118217

Περιγραφή Εργαστηρίου:

Στο εργαστήριο αυτό καλούμαστε να υλοποιήσουμε ένα σύστημα οπτικής αναγνώρισης ψηφίων, δηλαδή ένα σύστημα που θα ταξινομεί εικόνες χειρόγραφων ψηφίων από το 0 ως το 9, σε δέκα κατηγορίες-κλάσεις (που αντιστοιχούν στα δέκα ψηφία 0,1,..,9) ανάλογα με το ψηφίο που απεικονίζουν.

Μας δίνονται δύο αρχεία.txt που περιέχουν τα δεδομένα που θα χρησιμοποιήσουμε για την ανάλυση του συγκεκριμένου εργαστηρίου. Το ένα txt αρχείο αφορά τα train data (train.txt) δηλαδή τα δεδομένα που θα χρησιμοποιήσουμε για την εκπαίδευση του μοντέλου μας. Το άλλο txt αρχείο (test.txt) αφορά τα test data που θα χρησιμοποιήσουμε για να εκτιμήσουμε το πόσο καλά ή όχι πηγαίνει το μοντέλο μας. Τα train δεδομένα, μας είναι γνωστά κατά την ανάλυση ενώ τα test δεδομένα είναι κρυφά και τα χρησιμοποιούμε καθαρά για δοκιμαστικούς σκοπούς που θα μας βοηθήσουν να εκτιμήσουμε την επίδοση του μοντέλου μας πάνω σε τυχαία/άγνωστα δεδομένα.

Τα δεδομένα και στα δύο αρχεία είναι αποθηκευμένα σε μορφή πινάκων, δηλαδή σε καθένα από τα δύο αρχεία περιέχονται samples χειρόγραφων ψηφίων, καθένα από τα οποία αναπαρίσταται υπό την μορφή ενός πίνακα.

Τόσο στα train όσο και στα test δεδομένα, η πρώτη στήλη κάθε πίνακα αναφέρεται στο ίδιο το ψηφίο (label) ενώ οι υπόλοιπες 256 στήλες αναφέρονται στα χαρακτηριστικά (features) του ψηφίου τα οποία δεν είναι τίποτε άλλο από τιμές grayscale . Επομένως κάθε δείγμα-δεδομένο-ψηφίο μπορεί να απεικονιστεί ως ένα 16x16 πλέγμα (grid), κάθε κουτάκι του οποίου αναπαριστά και ένα pixel της εικόνας του ψηφίου. Κάθε ψηφίο που εμφανίζεται στην οθόνη, αποτελείται από ένα σύνολο από τέτοια “κουτάκια” που φωτίζονται με κατάλληλο τρόπο ώστε η συνολική εικόνα που βλέπουμε να απεικονίζει το θεωρούμενο ψηφίο.

Όπως γίνεται αντιληπτό από την παραπάνω περιγραφή, σε αυτήν την άσκηση θα εφαρμόσουμε supervised learning (γνωρίζουμε τις κατηγορίες στις οποίες θα ταξινομηθούν τα δείγματα μας- κλάσεις 0,1,…,9) δοκιμάζοντας διαφορετικά μοντέλα και συγκρίνοντάς τα ξεχωριστά, αλλά και συνδυάζοντας κάποια από τα μοντέλα αυτά για καλύτερες επιδόσεις ελαχιστοποιώντας πιθανά σφάλματα ταξινόμησης (όπως θα δούμε και στη συνέχεια).

Τα 13 πρώτα βήματα που παρουσιάζουμε παρακάτω, αποτελούν την προπαρασκευή του εργαστηρίου, δηλαδή περιέχουν απλή ανάγνωση και αναπαράσταση των δεδομένων μας καθώς και κάποιους βασικούς υπολογισμούς που θα χρησιμοποιηθούν μετέπειτα στο κύριο σκέλος του εργαστηρίου (υπόλοιπα βήματα)

Βήμα 1:

Αρχικά διαβάζουμε τα δεδομένα από τα αρχεία txt που μας δίνονται.

Από το train.txt παίρνουμε τα δεδομένα που θα εκπαιδεύσουμε και τα αποθηκεύουμε σε δύο πίνακες κατά τον εξής τρόπο:

- Στον πίνακα **X_train** περιέχονται τα δείγματα εκπαίδευσης, χωρίς τα labels, επομένως ο πίνακας αυτός είναι διάστασης (n_samples_train x 256_features).
- Στον πίνακα **y_train** περιέχονται τα αντίστοιχα labels επομένως αυτός ο πίνακας είναι μονοδιάστατος μήκους n_samples

Αντίστοιχα από το test.txt παίρνουμε τα test δεδομένα τα οποία θα χρησιμοποιήσουμε για την εκτίμηση του μοντέλου μας. Με την ίδια λογική όπως και στα train data μας, τα δεδομένα του test.txt αποθηκεύονται σε δύο πίνακες:

- Στον πίνακα **X_test** περιέχονται τα δείγματα test ,χωρίς τα labels, επομένως είναι διάστασης (m_samples x 256_features)
- Στον πίνακα **y_test** περιέχονται τα αντίστοιχα labels επομένως είναι ένας μονοδιάστατος πίνακας διάστασης m_samples

Με την συνάρτηση shape ελέγχουμε τον αριθμό των samples σε κάθε set:

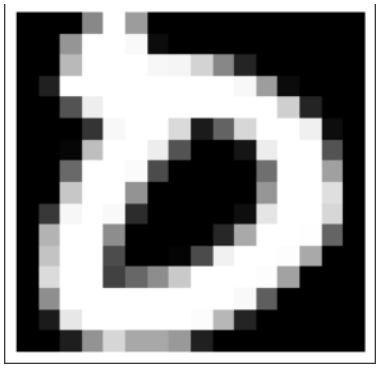
```
number_of_samples_train = 7291
number_of_samples_test = 2007
number_of_features = 256
```

Βήμα 2:

Αφού έχουμε διαβάσει και αποθηκεύσει όλα τα train δεδομένα μας, σχεδιάζουμε το υπ' αριθμόν 131 ψηφίο. Στον κώδικα μας το ψηφίο αυτό βρίσκεται στην θέση 130 του X_train πίνακα εφόσον στην python η αρίθμηση ξεκινά από το 0.

Για το βήμα αυτό, χρησιμοποιούμε τη συνάρτηση reshape της numpy ώστε να οργανώσουμε τα 256 features του 131ου ψηφίου σε ένα πίνακα 16×16, όπου κάθε κελί (pixel) του περιέχει το grayscale value της εικόνας του ψηφίου. Στη συνέχεια, με τη matplotlib.pyplot.imshow εμφανίζουμε στην οθόνη μας το ζητούμενο ψηφίο. Στα ορίσματά της pyplot.imshow παίρνουμε ως ελάχιστη τιμή το -1 και ως μέγιστη το 1 (μεταξύ αυτών των τιμών κυμαίνονται τα grayscale values)

Η εικόνα που προκύπτει είναι η εξής:



Βήμα 3:

Για κάθε ένα από τα δέκα ψηφία (0,1,…,9) σχεδιάζουμε ένα τυχαίο δείγμα που βρίσκουμε στα train data μας. Για να πραγματοποιηθεί αυτό, εντοπίζουμε αρχικά όλες τις θέσεις (indices) που περιέχουν το συγκεκριμένο label του ψηφίου (στο y_train) και βάζουμε σε μια λίστα τα indices αυτά. Ύστερα διαλέγουμε τυχαία ένα index από την λίστα και κάνουμε plot το ψηφίο που βρίσκεται στη θέση αυτή στον X_train πίνακα (εκεί περιέχεται η πληροφορία για την απεικόνισή του).

Η διαδικασία αυτή επαναλαμβάνεται για καθένα από τα 10 ψηφία με την χρήση for-loop, όπως φαίνεται και στον κώδικα που επισυνάπτουμε.

Η εικόνα που παίρνουμε σε κοινό figure για τα ψηφία 0,1,..9 όπως προκύπτουν από τα τυχαία δείγματα είναι η ακόλουθη:

Βήμα 4:

Θα αρχίσουμε από εδώ και έπειτα να χτίζουμε σιγά σιγά τις βασικές έννοιες που θα μας χρειαστούν για την υλοποίηση των διαφόρων ταξινομητών.

Μέση τιμή

Υπολογίζουμε αρχικά τη μέση τιμή του pixel (10, 10) για το ψηφίο «0» βάσει των train δεδομένων μας. Αυτή προκύπτει από όλες τις τιμές του pixel (10,10) για όλα τα διαφορετικά samples των train data που απεικονίζουν το ψηφίο «0». Μετρώντας τις θέσεις, βλέπουμε ότι το pixel (10,10) βρίσκεται στην θέση 170 του πίνακα X_train.

Για τους σκοπούς του βήματος αυτού χρησιμοποιήσαμε τη συνάρτηση της numpy.mean() για να υπολογίσουμε τη μέση τιμή όλων των τιμών για το pixel (10,10).

Η μέση τιμή ορίζεται ως το άθροισμα όλων των τιμών ενός χαρακτηριστικού διά το πλήθος τους: $\frac{\sum_{i=1}^{n} x_i}{n}$ όπου n το πλήθος των samples των οποίων θέλουμε να υπολογίσουμε τη μέση τιμή και x_i η τιμή του pixel (10,10) του i-οστού sample. Αν θέλαμε να υλοποιήσουμε τη συνάρτηση np.mean() σε μορφή δικού μας απλού κώδικα, θα ήταν ως εξής:

```
sum = 0
def mean(n_samples):
        for i in n_samples:
                sum += i
        return sum/len(n_samples)
```

Βρίσκουμε ότι:

```
Mean value of pixel(10,10) of all '0' is: -0.5041884422110553
```

Βήμα 5:

Διασπορά

Υπολογίζουμε τώρα τη διασπορά των χαρακτηριστικών του pixel (10, 10) για το ψηφίο «0» με βάση τα train δεδομένα. Η διασπορά του n_samples ορίζεται ως η μέση τιμή της διαφοράς του n_samples με τη μέση τιμή του n_samples, δηλαδή είναι: $E[(X - E[X])^2]$. Στο κώδικά μας χρησιμοποιήσαμε την έτοιμη συνάρτηση numpy.var() αλλά μια εύκολη υλοποίηση της συνάρτησης αυτής θα ήταν:

```
def var(n_samples):
        return mean(pow((n_samples – mean(n_samples)), 2))
```

Βρίσκουμε ότι:

```
Variance of values of pixel(10,10) of all '0' is: 0.5245221428814929
```

Βήμα 6:

Επεκτείνουμε τώρα τη διαδικασία των δύο προηγούμενων βημάτων, υπολογίζοντας τη μέση τιμή και διασπορά κάθε χαρακτηριστικού (pixel) για το ψηφίο «0» με βάση τα train δεδομένα.

Για τα 256 χαρακτηριστικά έχουμε 256 μέσες τιμές, μία για το καθένα, και 256 τιμές διασποράς.

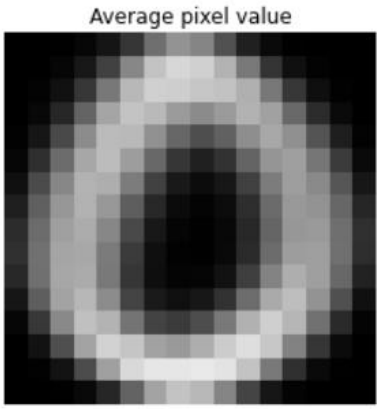
```
Shape of mean values: (1, 256)
Shape of variances: (1, 256)
```

Βήμα 7:

Σε αυτό το βήμα, σχεδιάζουμε το ψηφίο «0» χρησιμοποιώντας τις τιμές της μέσης τιμής για κάθε pixel του, που υπολογίσαμε στο βήμα 6 .

Ουσιαστικά, ακολουθούμε την ίδια διαδικασία με το βήμα 2, μόνο που εδώ χρησιμοποιούμε για κάθε pixel της εικόνας τη μέση τιμή του συγκεκριμένου pixel, που προέκυψε από όλα τα δείγματα για το ψηφίο «0». Επομένως αυτό που παίρνουμε είναι μια μέση τιμή της εικόνας του χειρόγραφου ψηφίου «0» που προκύπτει από όλα τα δείγματα που έχουμε στο train set μας για το ψηφίο αυτό.

Η εικόνα που παίρνουμε είναι η εξής:



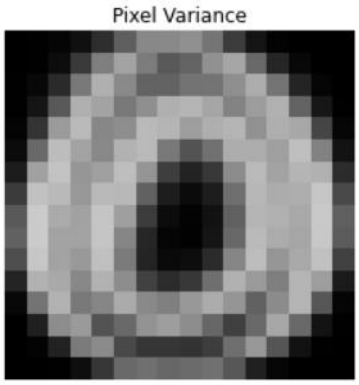
Παρατηρούμε ότι με την διαδικασία αυτή παίρνουμε μια αρκετά καλή εκτίμηση της εικόνας του ψηφίου 0.

Βήμα 8:

Τώρα, σχεδιάζουμε το ψηφίο «0» χρησιμοποιώντας αυτή τη φορά τις τιμές της διασποράς που υπολογίσαμε στο βήμα 6.

Ακόμη σημειώνουμε ως μια διαφορά στην υλοποίηση, ότι στη συνάρτηση pyplot.imshow βάλαμε ως ελάχιστη τιμή vmin τη τιμή 0 διότι η διασπορά παίρνει τιμές από 0 έως 1.

Παίρνουμε την εξής απεικόνιση:



Αυτό που παρατηρούμε, είναι οτι η εικόνα που προκύπτει από την διασπορά κάθε pixel του ψηφίου περιέχει περισσότερο θόρυβο σε σχέση με αυτήν της μέσης τιμής των pixel του βήματος 7.

Με την διασπορά διακρίνουμε ουσιαστικά όλες τις δυνατές μορφές του ψηφίου “0” που μπορούν να εντοπιστούν μέσα στο training set μας, ενώ με την μέση τιμή όλες αυτές οι πιθανές μορφές συνοψίζονται σε μια μέση εικόνα του ψηφίου “0” που παρατηρούμε.

Η μεγάλη διασπορά που βλέπουμε να εμφανίζεται, δείχνει ουσιαστικά τους πολλούς διαφορετικούς τρόπους γραφής με τους οποίους το ψηφίο “0” μπορεί να εμφανίζεται στα διάφορα samples που έχουμε στη διάθεσή μας.

Μπορούμε να πούμε πως με την μέση τιμή έχουμε μια καλύτερη εικόνα του ψηφίου, κάτι που αναμέναμε άλλωστε διότι μέσω μαθηματικών σχέσεων από τη θεωρία προκύπτει πως ο αριθμητικός μέσος όρος είναι πράγματι η καλύτερη εκτίμηση.

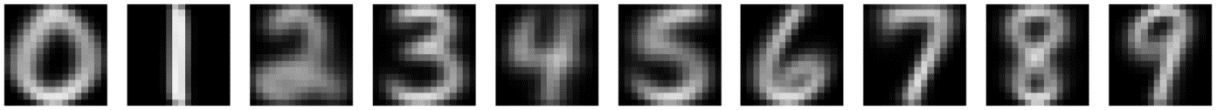
Σε κάθε περίπτωση όμως (είτε χρησιμοποιώντας την μέση τιμή είτε την διασπορά) μπορούμε να διακρίνουμε ποιο είναι το ψηφίο που απεικονίζεται.

Βήμα 9:

Το βήμα αυτό αποτελεί ένα συνδυασμό όλων των προηγούμενων βημάτων.

Υπολογίζουμε τις μέσες τιμές όλων των pixel για κάθε ψηφίο όπως προκύπτει από τα διαφορετικά δείγματα που έχουμε για αυτό από το train set, τις τοποθετούμε έπειτα στον πίνακα all_mean και ύστερα κατά τα γνωστά απεικονίζουμε τα 256 αυτά features κάθε ψηφίου σε μια εικόνα 16x16. Βάζουμε τις εικόνες σε κοινό figure χρησιμοποιώντας την συνάρτηση figure.add_subplot.

Λαμβάνουμε την εξής εικόνα:



Χρησιμοποιώντας τις μέσες τιμές για την απεικόνιση, βλέπουμε, όπως διαπιστώσαμε και παραπάνω, ότι η εικόνα που παίρνουμε για καθένα από τα 10 ψηφία είναι αρκετά ευδιάκριτη, δηλαδή χωρίς δυσκολία μπορούμε να δούμε για ποιο ψηφίο γίνεται λόγος κάθε φορά. Αυτό συμβαίνει καθώς απεικονίζουμε τη «μέση εικόνα» για κάθε ψηφίο, δηλαδή εξομαλύνουμε οποιαδήποτε «περίεργα» χειρόγραφα υπάρχουν για κάθε ψηφίο και κρατάμε μια μέση εικόνα από αυτά.

Βήμα 10:

Θέλουμε να ταξινομήσουμε το υπ΄αριθμόν 101 ψηφίο των test δεδομένων σε μία από τις 10 κατηγορίες βάσει της Ευκλείδειας απόστασης.

Για τον σκοπό αυτό χρησιμοποιούμε τις τιμές που βρήκαμε στο ερώτημα 9α, δηλαδή τις μέσες τιμές κάθε κλάσης (class_means) και με τη συνάρτηση euclidean_distances της sklearn υπολογίζουμε την ευκλείδεια απόσταση του test δεδομένου της θέσης 101 από κάθε κλάση.

Εναλλακτικά, μπορούμε να ορίσουμε μόνοι μας μια συνάρτηση που να υπολογίζει την ευκλείδεια απόσταση του test δεδομένου μας από την μέση τιμή κάθε κλάσης και να κρατάει την ελάχιστη απόσταση που προκύπτει, βάσει της σχέσης

$c^* = \text{argmin} ||x - \mu_c||_2$

όπου c=0,1,..,9 οι κλάσεις μας και μ_c η μέση τιμή της κάθε κλάσης.

Ορισμός Ευκλείδειας απόστασης:

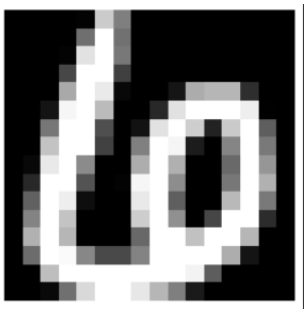
$$d_{ecdn} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Το test δεδομένο μας ταξινομείται στη κλάση από την οποία έχει τη μικρότερη απόσταση.

Ακολουθώντας την παραπάνω διαδικασία το ψηφίο μας ταξινομήθηκε εσφαλμένα στην κλάση 0 (αναγνωρίστηκε δηλαδή ως το ψηφίο 0), ενώ η πραγματική κλάση στην οποία ανήκει είναι η 6 (το ψηφίο 6)

```
Prediction for test digit No 101 is 0
The real value is 6
```

Η πραγματική εικόνα του ψηφίου στην θέση 101 των test data:



Βλέπουμε λοιπόν πως ο Ευκλείδειος ταξινομητής εμπεριέχει πιθανότητα λάθους εκτίμησης.

Βήμα 11:

Εφαρμόζουμε τον ευκλείδειο ταξινομητή για όλα τα ψηφία των test δεδομένων και τα κατατάσσουμε έτσι σε μία από τις 10 κατηγορίες (ψηφίο 0,1,…,9).

Αφού γίνει η διαδικασία αυτή, υπολογίζουμε την ακρίβεια (accuracy) της ταξινόμησης που εφαρμόσαμε σε ποσοστό επί τοις εκατό, η οποία εκφράζεται ως ο λόγος των σωστών προβλέψεων προς το συνολικό πλήθος των ψηφίων μας.

Το αποτέλεσμα που λάβαμε είναι το εξής:

```
Accuracy of Euclidean classifier in test set: 81.41504733432984 percent
```

Μπορούμε να πούμε πως με τον ευκλείδειο ταξινομητή έχουμε ένα αρκετά υψηλό score, όμως δεν μπορούμε να θεωρήσουμε ασήμαντο και το ποσοστό 19% περίπου των εσφαλμένων εκτιμήσεων (όπως είδαμε και στο προηγούμενο ερώτημα πήραμε λάθος εκτίμηση για το υπό εξέταση ψηφίο μας).

Μάλιστα αναλογιζόμενοι την σχετική απλότητα του προβλήματός μας, δεδομένου οτι τα χαρακτηριστικά που αναλύουμε είναι απλές τιμές grayscale, θα μπορούσαμε να φανταστούμε πως για πιο πολύπλοκα προβλήματα (πχ. ταξινόμηση εικόνων με πολλά χρώματα) το ποσοστό των εσφαλμένων εκτιμήσεων θα είναι πολύ μεγαλύτερο.

Βήμα 12:

Υλοποιούμε τώρα τον ταξινομητή ευκλείδειας απόστασης σαν ένα scikit-learn estimator, βασιζόμενοι στο lib.py αρχείο που μας δόθηκε σαν βοηθητικό υλικό.

Αρχικά λίγα λόγια για τους sklearn estimators:

Ένας estimator είναι ένα αντικείμενο που κάνει fit ένα μοντέλο με βάση κάποια training δεδομένα και είναι ικανός να δουλέψει και με νέα δεδομένα που μπορεί να του βάλουμε ως είσοδο.

Ένας sklearn estimator περιέχει 4 μεθόδους:

- Μια μέθοδο *__init__* που καλείται με την δημιουργία αντικειμένου της κλάσης και η οποία πρέπει να είναι απλή και να περιέχει απλά τις παραμέτρους που χρειαζόμαστε.
- Στη συνέχεια έχει μια μέθοδο ***fit*** με την οποία γίνεται η εκτίμηση των παραμέτρων του μοντέλου. Στη περίπτωσή μας, στη μέθοδο fit υπολογίζουμε τη μέση τιμή των κλάσεων από τα training δεδομένα.
- Επίσης έχει τη μέθοδο ***predict***, η οποία παίρνει ως είσοδο τα test δεδομένα ως δισδιάστατο numpy.array και προβλέπει τη κλάση στην οποία ανήκουν τα δεδομένα αυτά.
- Τέλος, υπάρχει η μέθοδος **score** η οποία προβλέπει την ακρίβεια της ταξινόμησης που έγινε από τον classifier.

Η κλάση EuclideanDistanceClassifier, κληρονομεί τις συναρτήσεις get_params και set_params από τη BaseEstimator, οι οποίες είναι αναγκαίες για έναν sklearn estimator. Επιπρόσθετα, κληρονομούμε και τη κλάση ClassifierMixin ώστε να πάρουμε παραπάνω στοιχεία ταξινομητών.

Βήμα 13:

α)Υπολογίζουμε το score του ευκλείδειου ταξινομητή με χρήση 5-fold cross-validation.

Για τον σκοπό αυτό χρησιμοποιούμε την συνάρτηση **KFold()** της sklearn βιβλιοθήκη, κάνοντας split σε 5 folds τον αρχικό πίνακα train_arr που περιέχει τα samples του train.txt αρχείου (lables και features). Έπειτα εφαρμόζουμε την συνάρτηση cross_val_score στον ευκλείδειο ταξινομητή μας για καθένα από τα 5 διαφορετικά σύνολα εκπαίδευσης που προκύπτουν από τα 5 splits (κάθε φορά κρατάμε ένα διαφορετικό fold ως test δεδομένα και τα υπόλοιπα 4 folds ως train δεδομένα, όπως ορίζεται από την μέθοδο 5-fold cross validation) και το ποσοστό επιτυχίας που παίρνουμε είναι :

```
Score of Euclidean classifier with 5-fold cross validation: 84.85803550358166 percent
```

Το ποσοστό αυτό είναι καλύτερο από αυτό που λάβαμε στο βήμα 11 για τον ευκλείδειο ταξινομητή χωρίς τα 5-folds.

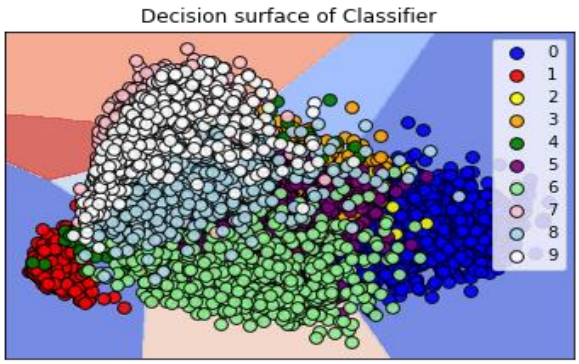
β) Σχεδιάζουμε τώρα την περιοχή απόφασης του ευκλείδειου ταξινομητή.

Για τον σκοπό αυτό χρησιμοποιούμε την συνάρτηση plot_clf, όπου για τις 10 διαφορετικές κλάσεις μας (ψηφία 0,1,...,9) προκύπτουν οι 10 διαφορετικές περιοχές απόφασης.

Ο "σκελετός" της συνάρτησης plot_clf βρίσκεται στο Lab 0.3 Scikit-learn.ipynb αρχείο που μας δίνεται ως βοηθητικό υλικό στα πλαίσια του μαθήματος.

Οπτικοποιούμε καλύτερα τις περιοχές απόφασης εφαρμόζοντας έναν pca μετασχηματισμό που μετατρέπει κάθε διάνυσμα 256 διαστάσεων σε ένα νέο διάνυσμα 2 διαστάσεων.

Έτσι έχουμε την εξής εικόνα:



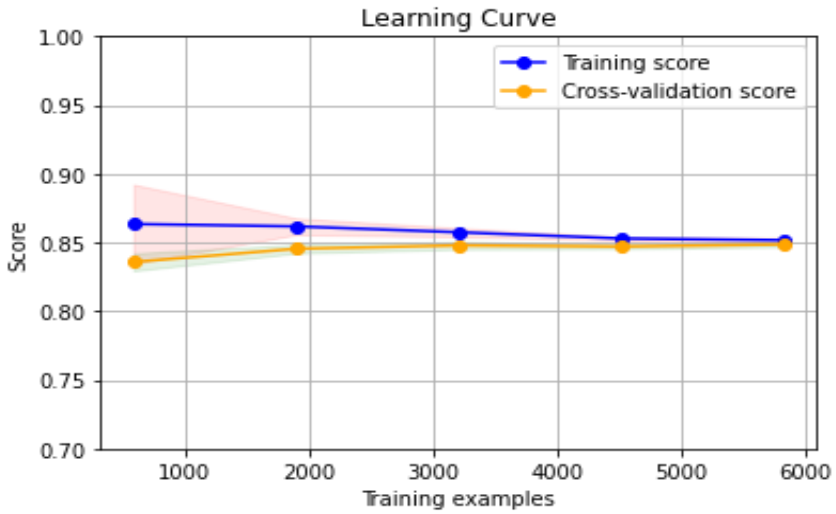
γ) Σχεδιάζουμε την καμπύλη εκμάθησης του ευκλείδειου ταξινομητή (learning curve).

Για τον σκοπό αυτό χρησιμοποιούμε την συνάρτηση plot_learning_curve που βρίσκεται στο βοηθητικό υλικό Lab 0.3 Scikit-learn.ipynb που μας δίνεται ως βοηθητικό εργαλείο.

Επίσης, χρησιμοποιούμε τη συνάρτηση της sklearn, learning_curve που επιστρέφει 3 πίνακες:

- Έναν πίνακα **train_size** που περιέχει τα μεγέθη των training samples που θα χρησιμοποιήσουμε για ταξινόμηση
- Έναν πίνακα **train_scores** που περιέχει τα scores για τον ευκλείδειο ταξινομητή που χρησιμοποιεί ως test data τα δεδομένα που περιέχονται στο test.txt αρχείο που μας δίνεται αρχικά για το εργαστήριο, και ως train data ένα μέρος των δεδομένων που περιέχονται στο train.txt αρχείο.
- Έναν πίνακα **test_scores** που περιέχει τα scores για τον ευκλείδειο ταξινομητή με 5-fold cross-validation, που χρησιμοποιεί σαν train data τα 4 folds που προκύπτουν από το split και σαν test data το άλλο fold (5 folds συνολικά)

Προκύπτει η εξής εικόνα:



Από την εικόνα καταλήγουμε στα εξής συμπεράσματα:

Αρχικά για μικρό αριθμό training samples παρατηρούμε ότι το cross validation score είναι μικρότερο από το training score, κάτι που πιθανότατα υπονοεί ότι το μοντέλο μας είναι underfit και επομένως χρειάζεται περισσότερα training samples.

Καθώς αυξάνεται ο αριθμός των train δεδομένων μας (samples), παρατηρούμε ότι οι δύο καμπύλες συγκλίνουν σε μια σταθερή τιμή. Από αυτό το σημείο και έπειτα δεν έχει επομένως νόημα να "τροφοδοτούμε" με καινούρια train data το μοντέλο μας καθώς η training διαδικασία έχει φτάσει το ανώτατο score που μπορεί να επιτευχθεί.

Βήμα 14

Από εδώ και έπειτα θα αρχίσουμε να "χτίζουμε" έναν Naive-Bayes ταξινομητή.

Αρχικά υπολογίζουμε τις a-priori πιθανότητες (ή αλλιώς prios) κάθε κατηγορίας-κλάσης που έχουμε.

Ο υπολογισμός μας βασίζεται στην μαθηματική σχέση

$$P(Y_c) = \frac{N_c}{N}$$

που ορίζει την a-priori πιθανότητα μιας κλάσης c ως το πηλίκο του πλήθους N_c των στοιχείων που ανήκουν σε αυτήν την κλάση, προς το πλήθος N των στοιχείων όλων των κλάσεων συνολικά.

Η σχέση αυτή προφανώς αναφέρεται στα train δεδομένα μας, τα οποία θα χρησιμοποιήσουμε ακολούθως για την εκπαίδευση του Naive Bayes ταξινομητή μας. Δηλαδή N_c και N είναι μεγέθη που αναφέρονται στα train data μας και συγκεκριμένα

στον y_train πίνακα εφόσον αυτός μας παρέχει την πληροφορία για τα labels των δειγμάτων μας.

Παίρνουμε τα ακόλουθα αποτελέσματα:

```
A priori for 0 = 0.16376354409546015
A priori for 1 = 0.13784117405019888
A priori for 2 = 0.10026059525442327
A priori for 3 = 0.09024825126868742
A priori for 4 = 0.08942531888629818
A priori for 5 = 0.07625840076807022
A priori for 6 = 0.09107118365107666
A priori for 7 = 0.08846523110684405
A priori for 8 = 0.07433822520916199
A priori for 9 = 0.08832807570977919
```

Βήμα 15

α) Κατασκευάζουμε έναν Naive-Bayes ταξινομητή σύμφωνα με τους βασικούς κανόνες της sklearn, όπως αυτοί αναφέρθηκαν στο Βήμα 12.

Για την υλοποίηση του ταξινομητή μας αναφέρουμε το μαθηματικό υπόβαθρο στο οποίο στηριχθήκαμε:

Οι ταξινομητές Naive Bayes είναι ένα σύνολο supervised αλγορίθμων εκμάθησης βασισμένοι στο θεώρημα Bayes

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

που κάνουν όμως την "αφελή" υπόθεση ότι τα features, στην προκειμένη περίπτωση τα pixels, είναι ανεξάρτητα μεταξύ τους.

Έχοντας μια μεταβλητή κατηγορίας (κλάσης) y και ένα εξαρτώμενο διάνυσμα χαρακτηριστικών x1 μέχρι xn, σύμφωνα με το θεώρημα του Bayes θα ισχύει:

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) P(x_1, \dots, x_n \mid y)}{P(x_1, \dots, x_n)}$$

Χρησιμοποιώντας την naive υπόθεση περί ανεξαρτησίας των x1,...,xn χαρακτηριστικών, η παραπάνω σχέση μετασχηματίζεται ως

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

και καθώς ο παρονομαστής είναι σταθερός, καταλήγουμε στη σχέση

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

Το **P(y)** είναι η υπόθεσή μας και εκφράζει την πιθανότητα το υπό εξέταση στοιχείο να ανήκει στην κατηγορία y (στο πρόβλημά μας σε μία από τις 10 κλάσεις 0,1,...,9). Διατυπώνοντάς το διαφορετικά, μπορούμε να πούμε οτι το P(y) εκφράζει τη σχετική συχνότητα του label y στο training set μας (a-priori πιθανότητα).

Οι a-priori πιθανότητες (τα P(y) δηλαδή των παραπάνω σχέσεων) για το πρόβλημά μας υπολογίστηκαν στο Βήμα 14.

Το **P(xi|y)** είναι η πιθανότητα του xi χαρακτηριστικού με δεδομένη την υπόθεσή μας (ότι το υπό ταξινόμηση ψηφίο ανήκει στην κατηγορία y) και μπορεί επίσης να υπολογιστεί απλά από το training set.

Η διαφορά μεταξύ των διαφόρων Naive Bayes ταξινομητών είναι στην υπόθεση που κάνουν σχετικά με τη κατανομή της P(xi|y)

Στην περίπτωσή μας, τα χαρακτηριστικά (τιμές pixels) παίρνουν συνεχείς τιμές, οπότε θα υλοποιήσουμε έναν Gaussian Naive Bayes, όπου η πιθανοφάνεια των features υποθέτουμε ότι είναι Gaussian, δηλαδή:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

Επειδή στον παρονομαστή της συνάρτησης πυκνότητας πιθανότητας, υπάρχει η διακύμανση σ^2 που μπορεί να πάρει την τιμή 0, για να αποφύγουμε τις απροσδιοριστίες που θα προέκυπταν λόγω μηδενισμού του, προσθέτουμε ένα κοινό μικρό offset της τάξης του 10^-9 που είναι αρκετά μικρότερο από τις διακυμάνσεις που παρατηρούμε για κάθε pixel κάθε κλάσης μας οπότε η συμβολή του στα αποτελέσματα που παίρνουμε μπορεί να θεωρηθεί ασήμαντη.

Για να βεβαιωθούμε πως η άθροιση αυτού του offset δεν θα επηρεάσει το αποτέλεσμα που προκύπτει, δοκιμάζουμε διάφορες τιμές του και βλέπουμε τα διαφορετικά score που λαμβάνουμε. Συνοψίζουμε τα αποτελέσματα αυτά στον ακόλουθο πίνακα:

OFFSET	SCORE
10^{-9}	71,99%
10^{-7}	73,04%
10^{-6}	73,89%

Παρατηρούμε πράγματι πως για πολύ μικρές τιμές του offset, παρατηρούνται ελάχιστες αλλαγές στο score. Με ελάχιστα μεγαλύτερο offset (από 10^-9 σε 10^-7 και μετά σε 10^-6) αυξάνεται ελάχιστα και το score, κάτι που είναι άλλωστε λογικό καθώς αύξηση του offset σημαίνει αύξηση στην διακύμανση, δηλαδή αύξηση στην "ανοχή" του ταξινομητή μας σε αποκλίσεις από την μέση παρατηρούμενη εικόνα κάθε ψηφίου (κλάσης) και επομένως μπορεί να αναγνωρίζει και ταξινομεί σωστά και πιο περίεργους γραφικούς χαρακτήρες (που αποκλίνουν από τον μέσο γραφικό χαρακτήρα που παρατηρείται για κάθε ψηφίο).

Κρατάμε το μικρότερο offset ώστε να έχουμε όσο το δυνατό πιο αντικειμενικό αποτέλεσμα με βάση τα δεδομένα μας.

Παρατηρήσεις στον κώδικα:

1) Μέσα στη μέθοδο fit για τον υπολογισμό της a priori πιθανότητας κάθε κλάσης χρησιμοποιήσαμε τη μέθοδο numpy.bincount η οποία επιστρέφει τις συχνότητες εμφάνισης κάθε πιθανού sample τις οποίες μετά διαιρούμε με τον συνολικό αριθμό των samples. Στο βήμα 14, για να αποφύγουμε τη χρήση αυτής της έτοιμης συνάρτησης παραθέσαμε έναν δικό μας απλό κώδικα. Και οι δύο εκτελέσεις έχουν τα ίδια αποτελέσματα.
2) Περνώντας στη συνάρτηση argmax ως παράμετρο το result ήρθαμε αντιμέτωποι για το β ερώτημα με το error: operands could not be broadcast together with shapes (10,2007) (10,). Ψάχνοντας στο διαδίκτυο, βρήκαμε ότι φταίει η διάσταση του πίνακα και μια λύση είναι να περάσουμε τον transpose του (result.T)

β) Το score που παίρνουμε τελικά για τον **CustomNBClassifier** ταξινομητή που φτιάξαμε εφαρμόζοντας όλα τα παραπάνω είναι:

```
Score: 0.7199800697558545
```

γ) Χρησιμοποιούμε τώρα την έτοιμη υλοποίηση **GaussianNB** της sklearn για Naive-Bayes ταξινομητή.

Το αποτέλεσμα που παίρνουμε είναι το εξής:

```
Accuracy of sklearn GaussianNB classifier

0.7194818136522172
```

Βλέπουμε πως το score αυτό παρουσιάζει ελάχιστη απόκλιση από το score της δικής μας υλοποίησης. Επιβεβαιώνουμε έτσι πως το offset που προσθέσαμε για αποφυγή πιθανών απροσδιοριστιών, πράγματι δεν διαστρεβλώνει τα αποτελέσματα που παίρνουμε και θεωρείται απειροελάχιστο και ασήμαντο.

Βήμα 16

Στο ερώτημα αυτό, υλοποιούμε ξανά τον Naive-Bayes ταξινομητή που φτιάξαμε προηγουμένως, χρησιμοποιώντας αυτή τη φορά διασπορά ίση με 1 για όλα τα χαρακτηριστικά όλων των κλάσεων.

Το score που λαμβάνουμε τώρα είναι το ακόλουθο:

```
Score: 0.8126557050323866
```

Βλέπουμε ότι αυτό είναι καλύτερο από ότι αυτό που παίρνουμε χρησιμοποιώντας την πραγματική διασπορά των χαρακτηριστικών κάθε κλάσης, η οποία είναι σε κάθε περίπτωση μικρότερη του 1.

Όπως γνωρίζουμε η διασπορά εκφράζει το μέσο όρο των αποκλίσεων από τη μέση τιμή. Όσο αυξάνεται η διασπορά τόσο περισσότερα διαφορετικά και πιο “περίεργα” χειρόγραφα δείγματα ψηφίων μπορούν να αναγνωριστούν και να ταξινομηθούν στην κλάση όπου πράγματι ανήκουν, με αποτέλεσμα να αυξάνεται και το score. Ουσιαστικά με αυτόν τον τρόπο δίνεται η δυνατότητα να ταξινομηθούν σωστά και οι πιο περίεργοι γραφικοί χαρακτήρες ψηφίων. Ωστόσο η μέση τιμή αποτελεί πάντοτε την καλύτερη εκτίμηση καθώς καλύπτει την γενική περίπτωση και έτσι ταξινομεί με μεγαλύτερη βεβαιότητα τα περισσότερα ψηφία αλλά χάνει τις ειδικές περιπτώσεις όπου τα ψηφία έχουν γραφεί με πιο “ιδιαίτερο-περίεργο” γραφικό χαρακτήρα.

Πρέπει να σημειώσουμε πως αυξάνοντας κατά πολύ την διασπορά, υπάρχει ο κίνδυνος να ταξινομούνται μεν σωστά οι ειδικές και πιο σπάνιες περιπτώσεις δειγμάτων (με πολύ περίεργο γραφικό χαρακτήρα) αλλά να μην αναγνωρίζονται πλέον σωστά οι πιο συνηθισμένες περιπτώσεις που προσεγγίζουν την μέση εικόνα κάθε ψηφίου. Με αυτόν τον τρόπο καταλήγουμε δηλαδή να χάνουμε την γενική περίπτωση και να επικεντρωνόμαστε στις πολύ ειδικές κάτι που θα λειτουργήσει εντέλει επιβαρυντικά για τον ταξινομητή μας μειώνοντας τελικά το score του. Το φαινόμενο αυτό είναι γνωστό ως overfitting.

Βήμα 17

Χρησιμοποιούμε έτοιμες υλοποιήσεις της sklearn για τους ταξινομητές Nearest Neighbors, SVM και GaussianNB και παρατηρούμε το score που παίρνουμε κάθε φορά.

Για τον SVM πειραματιζόμαστε για διαφορετικές τιμές της υπερπαραμέτρου kernels (linear, polynomial, rbf, sigmoid) καθώς επίσης και για τον KNN χρησιμοποιούμε ενδεικτικά δύο τιμές για το k, k=5 και k=10. Συνοψίζουμε τα αποτελέσματα που πήραμε στο ακόλουθο πινακάκι (περιέχει και το score του δικού μας, CustomNB ταξινομητή):

Classifier	Score
GaussianNB	71,94%
CustomNB $\left(\text{offset} = 10^{-9}\right)$	71,99%
CustomNB $\left(\text{offset} = 10^{-3}\right)$	77,42%
Custom_NB_var1	81,26%
SVM $\left(\text{sigmoid}\right)$	85,05%
SVM $\left(\text{linear}\right)$	92,62%
Nearest Neighbors $\left(k = 10\right)$	93,57%
Nearest Neighbors $\left(k = 5\right)$	94,46%
SVM $\left(\text{rbf}\right)$	94,71%
SVM $\left(\text{polynomial}\right)$	95,36%

Βλέπουμε ότι για SVM με polynomial υπερπαράμετρο έχουμε τις καλύτερες επιδόσεις, επιτυγχάνοντας το υψηλότερο score που ανέρχεται στο 95,36%. Αμέσως μετά ακολουθεί η υλοποίηση του ίδιου ταξινομητη με rbf υπερπαράμετρο και πολύ κοντινές επιδόσεις έχουμε και με την KNN μέθοδο ταξινόμησης. Από την άλλη τα αποτελέσματα που λαμβάνουμε από τους Naive-Bayes ταξινομητές είναι λιγότερο ικανοποιητικά με χειρότερο να καταγράφεται το score του GaussianNB σε ποσοστό 71,94%. Τα αποτελέσματα αυτά για τους τελευταίους ταξινομητές δεν μας εκπλήσσουν καθώς όπως αναλύσαμε και στην θεωρία παραπάνω (Βήμα 15) ο Naive-Bayes κάνει την αφελή υπόθεση πως τα features, εδώ τα pixels κάθε εικόνας, είναι ανεξάρτητα μεταξύ τους, κάτι που προφανώς δεν ισχύει.

Βήμα 18

Σε αυτό το σημείο, συνδυάζουμε ταξινομητές που έχουμε δει στα προηγούμενα ερωτήματα με σκοπό την δημιουργία ενός “μετα-ταξινομητή” που θα επιτυγχάνει καλύτερη επίδοση από τις επιμέρους επιδόσεις των ταξινομητών που χρησιμοποιήσαμε. Η τεχνική αυτή ονομάζεται **ensembling**, καθώς όπως υποδηλώνει και το ίδιο το όνομα, το τελικό αποτέλεσμα, δηλαδή η τελική ταξινόμηση, προκύπτει έπειτα από τον συνδυασμό και συμψηφισμό των αποτελεσμάτων των επιμέρους ταξινομητών.

Χρησιμοποιούμε δύο διαφορετικές υλοποιήσεις του ensembling:

α) VotingClassifier

β) BaggingClassifier

Και οι δύο τύποι classifiers υλοποιούνται χρησιμοποιώντας τα αντίστοιχα εργαλεία της βιβλιοθήκης sklearn.

VotingClassifier

Ο VotingClassifier είναι ένας μετα-ταξινομητής που συνδυάζει επιμέρους ταξινομητές βάζοντάς τους να ψηφίσουν για το αποτέλεσμα. Κάθε ταξινομητής δηλαδή, ψηφίζει για κάποια κλάση και στο τέλος η κλάση που συγκέντρωσε τις περισσότερες ψήφους "κερδίζει". Όπως μπορεί να γίνει αντιληπτό, για αυτήν την υλοποίηση θα συνδυάσουμε μονό αριθμό ταξινομητών ώστε να αποφευχθεί το ενδεχόμενο της ισοπαλίας και να υπάρχει πλειοψηφία (majority) για κάποια κλάση, η οποία εντέλει θα κερδίσει ("the majority wins").

Επιλέγουμε επίσης ως τύπο ψηφοφορίας το hard voting, όπου η ψήφος κάθε classifier είναι ισοδύναμη και η πλειοψηφία θα επικρατήσει τελικά.

Για να εξασφαλίσουμε μια αρκετά καλή επίδοση στον VotingClassifier, προσπαθούμε να συνδυάσουμε ταξινομητές που τείνουν να έχουν διαφορετικό τύπο λαθών, ώστε να μειώσουμε την "προκατάληψη" να επιλέγεται εσφαλμένα κάποιο συγκεκριμένο ψηφίο και να έχουμε έτσι στο σύνολο τις ελάχιστες δυνατές λανθασμένες ταξινομήσεις (απλοϊκά θα μπορούσαμε να πούμε πως θέλουμε οι λάθος προβλέψεις του ενός ταξινομητή για κάποιο ψηφίο να «διορθώνονται» από τις σωστές προβλέψεις του άλλου ταξινομητή για το ίδιο ψηφίο).

Για να διακρίνουμε τον τύπο λαθών που τείνει να κάνει καθένας από τους ταξινομητές που είδαμε στα προηγούμενα ερωτήματα, υλοποιούμε την συνάρτηση test_scores που εμφανίζει το ποσοστό των επιτυχημένων προβλέψεων για κάθε ψηφίο από κάθε ταξινομητή.

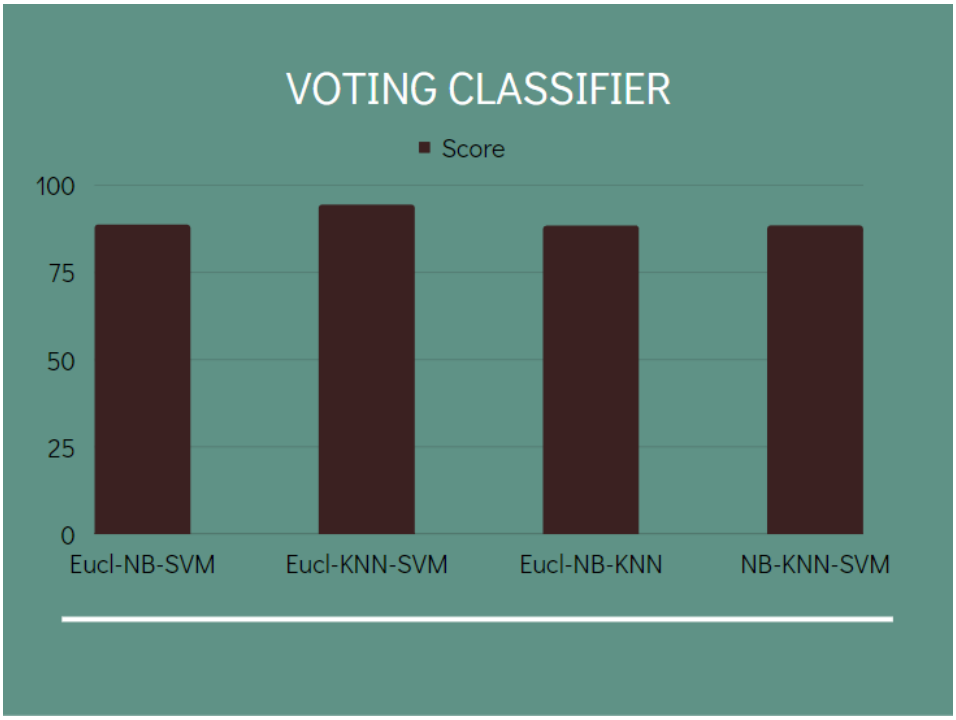
Βλέποντας συγκεντρωτικά ποια ψηφία τείνει να ταξινομεί "περισσότερο" ή "λιγότερο" σωστά ο κάθε ταξινομητής, εκτιμούμε πως ο καλύτερος συνδυασμός που μπορούμε να κάνουμε είναι με τους Euclidean, Naive-Bayes και SVM.

Το score που λαμβάνουμε για τον VotingClassifier που συνδυάζει τους τρεις παραπάνω ταξινομητές είναι:

```
Hard Voting Score: 0.8858993522670653
```

Αυτό είναι πράγματι καλύτερο από τα επιμέρους score για τον Euclidean και τον Naive-Bayes αντίστοιχα, αλλά όχι καλύτερο από το score του SVM με polynomial υπερπαράμετρο.

Πειραματιζόμαστε, συνδυάζοντας διαφορετικές τριάδες ταξινομητών και τα αποτελέσματα που παίρνουμε τα συνοψίζουμε στο παρακάτω γράφημα :



Σε όλες τις περιπτώσεις το συνολικό score που παίρνουμε από τον μετα-ταξινομητή μας είναι αρκετά υψηλό και καλύτερο από κάποια επιμέρους score. Ο καλύτερος συνδυασμός που βλέπουμε ότι προκύπτει είναι ο Euclidean-KNN-SVM.

Σχόλιο : Θεωρήσαμε ότι έπρεπε να διαλέξουμε ανάμεσα σε διαφορετικούς ταξινομητές και όχι στους ίδιους με διαφορετική υλοποίηση κάθε φορά (π.χ να χρησιμοποιήσουμε SVM(poly) και SVM(rbf)). Αν η υπόθεση αυτή είναι λανθασμένη, η επόμενη κίνησή μας θα ήταν να πάρουμε τους καλύτερους ταξινομητές που είχαμε βρει βάση αποτελεσμάτων accuracy (βήμα 17) και να συνδυάσουμε αυτούς, αποφεύγοντας να έχουν το ίδιο τύπο λαθών.

BaggingClassifier

Ο BaggingClassifier είναι ένας μετα-ταξινομητής που εφαρμόζεται σε base classifiers με διαφορετικά τυχαία υποσύνολα dataset για training που έχουν προκύψει από το αρχικό training dataset με τυχαίο χωρισμό. Αφού προκύψουν οι επιμέρους προβλέψεις από τους base classifiers, ο BaggingClassifier τις συγκεντρώνει για να διαμορφώσει την τελική πρόβλεψη, που θα προκύψει είτε με ψηφοφορία είτε με averaging.

Επιλέγουμε εδώ σαν base estimator τον SVM polynomial, καθώς αυτός μας έδωσε το καλύτερο score στο Βήμα 17, και το score που παίρνουμε από τον BaggingClassifier είναι

```
0.9506726457399103
```

Συγκρίνοντας τα αποτελέσματα του VotingClassifier και του BaggingClassifier, είναι εμφανές πως ο δεύτερος πετυχαίνει πολύ καλύτερες επιδόσεις με αρκετά υψηλότερο score ταξινόμησης.

Το αποτέλεσμα αυτό είναι κάτι λογικό καθώς με το bagging παίρνουμε κάθε φορά ένα διαφορετικό τυχαίο subset από το αρχικό training dataset, επομένως ο κάθε base estimator που θα συμβάλει στο ensembling εκπαιδεύεται με διαφορετικά και ασυσχέτιστα μεταξύ τους features και έτσι τα αποτελέσματα που θα προκύψουν από τις επιμέρους ταξινομήσεις θα εμφανίζουν αφενός υψηλό score λόγω της ίδιας της υλοποίησης του SVM και αφετέρου θα χαρακτηρίζονται από διαφορετικό τύπο λαθών λόγω της διαφορετικής εκπαίδευσης κάθε φορά. Συνδυάζοντας τις υψηλές επιμέρους επιδόσεις προκύπτει ένας ακόμα καλύτερος μετα-ταξινομητής.

Βήμα 19

α)

Με αυτό το βήμα κάνουμε μια σύντομη εισαγωγή στο χώρο των νευρωνικών δικτύων και βλέπουμε πως αυτά μπορούν να αξιοποιηθούν για την επίλυση προβλημάτων ταξινόμησης.

Γενικά δουλεύοντας με νευρωνικά δίκτυα απαιτείται πολλές φορές και η διαχείριση μεγάλου όγκου δεδομένων. Η βιβλιοθήκη PyTorch, που χρησιμοποιούμε για την ανάπτυξή τους, παρέχει διάφορα εργαλεία για την αποθήκευση, ομαδοποίηση και τη παράλληλη φόρτωση των δεδομένων στη μνήμη κάτι που οδηγεί και σε αύξηση της ταχύτητας του προγράμματος αλλά και εξοικονόμηση της μνήμης.

Ξεκινώντας, χρειάζεται να ορίσουμε ένα **custom Dataset** της PyTorch στο οποίο θα αποθηκεύουμε όλα μας τα δεδομένα. Η κλάση Dataset θα περιέχει δύο μεθόδους, μία μέθοδο len(), η οποία επιστρέφει το μήκος του dataset και μία μέθοδο getitem(), η οποία επιστρέφει το στοιχείο που βρίσκεται σε δοσμένο index.

Στη συνέχεια υλοποιούμε τους τρείς Dataloader μας ένα για το train, έναν για το validation και ένα για το test, οι οποίοι αναλαμβάνουν τη προσπέλαση των δεδομένων μας, τη διαχείριση των batches, το transform των data και πολλά άλλα.

Ορίζουμε ως μέγεθος του batch το 32 (γενικά ένα μικρό νούμερο πολλαπλάσιο του 2), και num_workers το 2 (το νούμερο ήταν σύσταση από ένα error που προέκυψε καθώς προσπαθήσαμε την υλοποίηση με άλλη τιμή, το οποίο μας πρότεινε να χρησιμοποιήσουμε για το σύστημά μας καλύτερα δύο workers)

β)

Ένα νευρωνικό δίκτυο ενός επιπέδου μπορεί να χρησιμοποιηθεί μόνο όταν τα δεδομένα είναι γραμμικά διαχωρίσιμα, δηλαδή να μπορούν να διαχωριστούν με μια απλή ευθεία. Η υλοποίηση αυτή αναφέρεται σε πολύ απλά προβλήματα, τα οποία

ωστόσο δεν συναντώνται τόσο συχνά στην πραγματικότητα. Τα πολυεπίπεδα νευρωνικά δίκτυα λύνουν αυτό το πρόβλημα.

Έτσι στα πλαίσια αυτής της άσκησης θα πειραματιστούμε με δυεπίπεδα και τριεπίπεδα δίκτυα, δηλαδή να έχουν ένα και δύο hidden layers αντίστοιχα.

Αντίστοιχα και για δύο hidden layers. (όταν λέμε για δυεπίπεδο δίκτυο μιλάμε για το hidden και το output layer, καθώς το input layer πολλοί δεν το υπολογίζουν διότι οι νευρώνες σε αυτό το layer δεν εκτελούν κάποιον υπολογισμό).

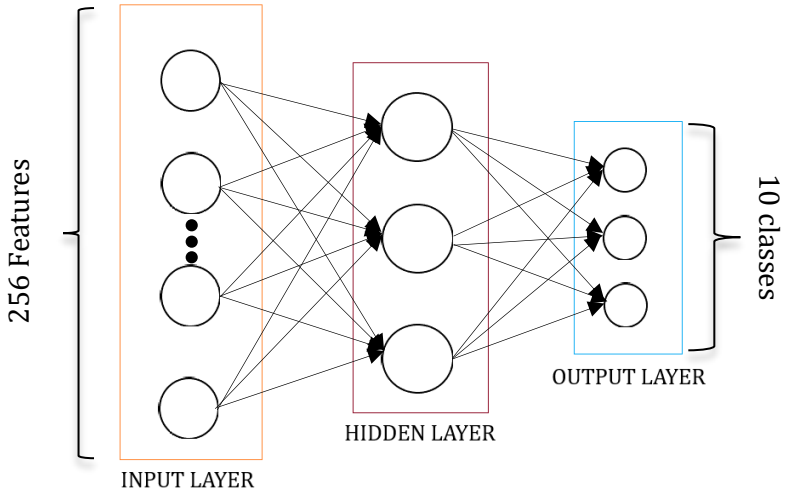
Στα πλαίσια της άσκησης κρατάμε τον αριθμό των hidden layers στα δύο διότι για μεγαλύτερο αριθμό υπάρχει ο κίνδυνος του overfitting όπου το νευρωνικό μας θα αναλύει τα δεδομένα που του εισάγουμε σε υπερβολικό βαθμό, κοιτώντας κάθε λεπτομέρεια, με αποτέλεσμα να μαθαίνει πολύ συγκεκριμένες περιπτώσεις και να μη μπορεί στη συνέχεια να γενικεύει για άλλα δεδομένα.

Όσο αναφορά των αριθμό των νευρώνων, τόσο για το input όσο και για το output layer ο αριθμός των νευρώνων είναι προκαθορισμένος.

Στο input θα εισάγουμε τα 256 features της κάθε εικόνας και περιμένουμε από το νευρωνικό να έχει ως έξοδο 10 κλάσεις, οπότε 256 νευρώνες και 10 νευρώνες αντίστοιχα για κάθε layer. Στα τα hidden layers έρχεται η ώρα του πειραματισμού. Δοκιμάζουμε διαφορετικές τιμές για το καθένα εξάγοντας κάθε φορά τα αντίστοιχα αποτελέσματα, τα οποία και θα παρουσιάσουμε παρακάτω. Σε κάθε περίπτωση περιοριζόμαστε στους εξής κανόνες:

1) Το μέγεθος των hidden layers είναι ανάμεσα στο μέγεθος του input και του output layer (256,10) και
2) Καθώς κατευθυνόμαστε προς την έξοδο ο αριθμός των νευρώνων κάθε hidden layer όλο και μειώνεται

Παραθέτουμε μια εικόνα για το πώς μοιάζει το νευρωνικό μας δίκτυο συγκεντρωτικά:



Η **συνάρτηση ενεργοποίησης**, δηλαδή η συνάρτηση που καθορίζει την ενεργοποίηση ή μη του νευρώνα, μας παρέχεται από τη βιβλιοθήκη PyTorch. Θα εξετάσουμε 3 διαφορετικές activation functions:

I. ReLU (Rectified Linear Activation Function)
II. Sigmoid
III. Tanh

Υλοποίηση των μοντέλων:

Ξεκινάμε αναπτύσσοντας όλα τα μοντέλα ως πρότυπα νευρωνικού δικτύου της PyTorch. Συγκεκριμένα έχουμε έξι διαφορετικά μοντέλα

1) (1 Hidden Layer, ReLU)
2) (2 Hidden Layer, ReLU)
3) (1 Hidden Layer, Sigmoid)
4) (2 Hidden Layer, Sigmoid)
5) (1 Hidden Layer, Tanh)
6) (2 Hidden Layer, Tanh)

Καθώς η υλοποίηση μας έχει να κάνει με fully-connected δίκτυο και όχι convolution, τα layer μας θα είναι linear που σημαίνει ότι θα εφαρμόζουν ένα γραμμικό μετασχηματισμό στην είσοδο χρησιμοποιώντας τα αποθηκευμένα weights και biases.

Τέλος, γράφουμε τον κώδικα για το training του νευρωνικού.

Ξεκινάμε δεχόμενοι ως είσοδο από το χρήστη τις παραμέτρους με τις οποίες επιθυμεί να αρχικοποιήσει το μοντέλο οι οποίες είναι ο αριθμός των hidden layers, ο αριθμός των νευρώνων κάθε hidden layer και η συνάρτηση ενεργοποίησης.

Στη συνέχεια ορίζουμε ένα **criterion** με CrossEntropy loss function, δημοφιλής για τη ταξινόμηση σε πολλές κλάσεις. Το criterion είναι χρήσιμο στην εκπαίδευση του νευρωνικού, δοσμένης εισόδου και στόχου υπολογίζει το gradient (απόκλιση) σύμφωνα με τη loss function που του δώσαμε. Επίσης, ορίζουμε και έναν stochastic gradient descent **optimizer** περνώντας του τις παραμέτρους του μοντέλου και ένα learning rate, δηλαδή τον ρυθμό με τον οποίο μαθαίνει το μοντέλο, με τιμή 0.01

Πλέον είμαστε έτοιμοι να εκπαιδεύσουμε το νευρωνικό δίκτυο. Αντί να ορίσουμε συγκεκριμένο αριθμό από epochs, αφήνουμε το νευρωνικό να εκπαιδεύεται μέχρι η ακρίβεια του validation να πάψει να αυξάνεται. Φορτώνουμε από το train_loader τα δεδομένα (features) μαζί με τα labels τους. Το zero_grad() χρησιμοποιήθηκε ώστε να μπορεί να γίνει restart του step χωρίς losses. Αν δεν χρησιμοποιηθεί, το loss θα αυξηθεί και δε θα μειωθεί που είναι το ζητούμενο. Οπότε, υπολογίζουμε το loss και ύστερα κάνουμε optimize, «κουρδίζουμε» δηλαδή το νευρωνικό μας έτσι ώστε στην επόμενη προσπάθειά του να τα πάει καλύτερα από ότι στη προηγούμενη.

Εκτυπώνουμε το accuracy για τα training δεδομένα και συνεχίζουμε την ίδια διαδικασία για το validation set.

Στον παρακάτω πίνακα, συνοψίζονται γραφικά όλα τα αποτελέσματα από τις μετρήσεις που κάναμε «πειράζοντας» τις παραμέτρους. Κρατάμε το σκορ του μοντέλου στη τελευταία εποχή.

Hidden Layers	Activation	Size	Train_Acc	Epochs
1	ReLU	30	94.1529	10
1	ReLU	60	94.8388	14
1	ReLU	90	94.2386	10
1	ReLU	120	94.3930	11
2	ReLU	30-15	93.3813	11
2	ReLU	60-45	94.44	14
2	ReLU	90-75	93.9643	10
2	ReLU	120-105	94.5987	12
1	Sigmoid	30	93.9128	42
1	Sigmoid	60	92.5754	27
1	Sigmoid	90	91.0665	16
1	Sigmoid	120	92.4039	23
2	Sigmoid	30-15	30.4012	11
2	Sigmoid	60-45	24.4684	6
2	Sigmoid	90-75	25.0514	7
2	Sigmoid	120-105	23.95	6
1	Tanh	30	94.1529	12
1	Tanh	60	94.9759	10
1	Tanh	90	94.6502	12
1	Tanh	120	94.4788	12
2	Tanh	30-15	95.4046	25
2	Tanh	60-45	96.0905	22
2	Tanh	90-75	94.6330	13
2	Tanh	120-105	95.1646	15

Οι παρατηρήσεις μας πάνω στα αποτελέσματα που πήραμε είναι οι εξής:

1) Τις καλύτερες επιδόσεις τις πετυχαίνει το νευρωνικό με συνάρτηση ενεργοποίησης tanh
2) Το νευρωνικό έχει καλύτερο accuracy με δύο hidden layer, ωστόσο του παίρνει περισσότερο χρόνο(εποχές) για να καταλήξει σε καλύτερο σκορ
3) Όσο αυξάνεται ο αριθμός των νευρώνων στα layers, τα epochs μειώνονται και η ακρίβεια σε γενικές γραμμές αυξάνεται.
4) Ξεκινήσαμε με τιμή 0.01 για το learning rate, αυξάνοντας τη παράμετρο αυτή βλέπουμε ότι το μοντέλο μας εκπαιδεύεται πιο γρήγορα και γενικά πετυχαίνει μεγαλύτερο accuracy.

Τα διαφορετικά αποτελέσματα που παίρνουμε για τις διαφορετικές συναρτήσεις ενεργοποίησης έχουν να κάνουν με την παράγωγο της κάθε μίας, η οποία εκφράζει ουσιαστικά την κλίση της, από την οποία θα δούμε προς ποια κατεύθυνση και κατά πόσο πρέπει να αλλάξουμε την learning curve μας ώστε να πετύχουμε καλύτερο training.

γ)

Υλοποιούμε το νευρωνικό συμβατό με sklearn. Ουσιαστικά υλοποιούμε τις μεθόδους fit και score, κάτι που έχουμε γράψει σε κώδικα από το προηγούμενο υποερώτημα.

Κάνοντας fit το μοντέλο ουσιαστικά εκπαιδεύουμε το νευρωνικό μας. Τα αποτελέσματα που λάβαμε ωστόσο είναι ασαφή:

```
Epoch: 1
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 2
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 3
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 4
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 5
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 6
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 7
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 8
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 9
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 10
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 11
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 12
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 13
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 14
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
Epoch: 15
Accuracy in train: 15.243484 %
Accuracy in val: 15.146986 %
```

Το train αλλά και το validation accuracy του νευρωνικού φαίνεται να είναι εξαιρετικά χαμηλό, κάτι που αδυνατούμε να εξηγήσουμε.

δ)

Το σκορ που λαμβάνουμε από την αξιολόγηση του νευρωνικού μας καλώντας τη μέθοδο score για τα test δεδομένα, είναι:

```
0.15146985550572994
```

Ένα νούμερο το οποίο προκύπτει πιθανώς από σφάλμα στον κώδικα.

Πηγές

Παρακάτω αναφέρουμε μερικές από τις εξωτερικές πηγές που συμβουλευτήκαμε οι οποίες συντέλεσαν στην ολοκλήρωση της 1ης Εργαστηριακής Άσκησης.

https://pytorch.org/tutorials

https://machinelearningmastery.com

https://towardsdatascience.com

Οι πίνακες σύγκρισης αποτελεσμάτων πραγματοποιήθηκαν στη σελίδα:

https://www.canva.com/graphs

Αρκετή βοήθεια στον κώδικα λάβαμε και από τις σελίδες:

https://stackoverflow.com

www.w3schools.com