Τσαντίκης Γεώργιος
ΑΕΜ: 10722

Αναφορά 2ης υποχρεωτικής εργασίας στο μάθημα «ΝΕΥΡΩΝΙΚΑ ΔΙΚΤΥΑ- ΒΑΘΙΑ ΜΑΘΗΣΗ»

ΠΕΡΙΕΧΟΜΕΝΑ:

1. Εισαγωγή

Στην παρούσα εργασία πραγματοποιήθηκε η υλοποίηση από την αρχή (**from scratch**), με κώδικα **Python**, ενός **Support Vector Machine** (SVM) που εκπαιδεύεται για τον διαχωρισμό όλων των κλάσεων που υπάρχουν στη βάση δεδομένων **CIFAR-10**. Η CIFAR-10 περιέχει 60.000 έγχρωμες (RGB) εικόνες μεγέθους 32x32 pixels, κατανεμημένες σε 10 διαφορετικές κατηγορίες /κλάσεις (π.χ., σκύλος, φορτηγό, αεροπλάνο), από τις οποίες 50.000 εικόνες είναι δείγματα εκπαίδευσης και 10.000 δείγματα ελέγχου.

Η μελέτη επικεντρώνεται στην ανάπτυξη και υλοποίηση ενός SVM, χωρίς τη χρήση έτοιμων συναρτήσεων από βιβλιοθήκες μηχανικής μάθησης (όπως Scikit-Learn), με σκοπό την κατανόηση σε βάθος των μηχανισμών εκπαίδευσης και λειτουργίας ενός SVM.

Χρησιμοποιήθηκαν τεχνικές όπως :

- **Κανονικοποίηση** εισόδων ώστε οι τιμές των pixel να βρίσκονται στο **εύρος [0, 1].**
- Εφαρμογή **PCA (Principal Component Analysis)** για τη μείωση διάστασης των δεδομένων και την αποδοτικότερη εκπαίδευση του SVM, .
- Εκπαίδευση με χρήση **batches** για διαχείριση της υπολογιστικής πολυπλοκότητας.
- Ρύθμιση **υπερπαραμέτρων** (C, γ, βαθμός πολυωνύμου) και χρήση διαφόρων **Kernel functions** (linear, polynomial, rbf) για τον υπολογισμό και τον προσδιορισμό των **support vectors** και του **bias** κατά τη διαδικασία εκπαίδευσης.

Στόχοι της εργασίας είναι :

- Η διερεύνηση της επίδρασης διαφορετικών **πυρήνων** και τιμών **υπερπαραμέτρων** στην απόδοση του SVM.
- Η παρουσίαση χαρακτηριστικών παραδειγμάτων ορθής και εσφαλμένης κατηγοριοποίησης από το μοντέλο.
- Η σύγκριση της απόδοσης του **SVM** με άλλες μεθόδους κατηγοριοποίησης, όπως **Nearest Neighbor** (1-NN, 3-NN), **Nearest Class Centroid** (NCC) και **Multi-Layer Perceptron (MLP)** με χρήση **Hinge Loss**.

2. Περιγραφή αλγορίθμων – Ανάλυση Κώδικα

2.1. Βιβλιοθήκες

Οι παρακάτω βιβλιοθήκες αποτελούν απαραίτητα εργαλεία για την κάλυψη των απαιτήσεων της εργασίας :

- **import numpy as np** → για την αποδοτική διαχείριση αριθμητικών υπολογισμών και πολυδιάστατων πινάκων (x_train, x_test κτλ.)
- **import cvxpy as cp** → βιβλιοθήκη βελτιστοποίησης που χρησιμοποιείται για τη διατύπωση και επίλυση του δυϊκού προβλήματος SVM.
- **import time** → για την μέτρηση του χρόνου εκπαίδευσης
- **import pickle** → για την αποθήκευση και ανάκτηση των δεδομένων σε δυαδική μορφή
- **import requests** → για την αποστολή αιτήματος HTTP, με σκοπό την λήψη του dataset CIFAR-10 από το διαδίκτυο
- **import tarfile** → για την αποσυμπίεση των δεδομένων (εξάγει αρχεία .tar.gz)
- **import os**→ για τον χειρισμό λειτουργιών του συστήματος αρχείων (έλεγχος ύπαρξης αρχείων κτλ.)

2.2. Download και extract του dataset CIFAR-10

Η παρακάτω συνάρτηση εξυπηρετεί τη λήψη από το διαδίκτυο και την αποσυμπίεση του dataset σε περίπτωση που το συμπιεσμένο αρχείο με όνομα *"cifar-10-python.tar.gz"* δεν υπάρχει στον τοπικό φάκελο [*"os.path.exists()"*] στον οποίο εξάγει το περιεχόμενο του σε δυαδική μορφή. Πλέον, τα δεδομένα του CIFAR-10 είναι έτοιμα για περαιτέρω επεξεργασία σε μη συμπιεσμένη μορφή στον φάκελο "cifar-10-batches-py".

```
# Download και extract του dataset CIFAR-10
def download_and_extract_cifar10():
    url = "https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"
    archive_name = "cifar-10-python.tar.gz"
    if not os.path.exists(archive_name):
        print("Downloading CIFAR-10 dataset...")
        with requests.get(url, stream=True) as r:
            with open(archive_name, "wb") as file: # ανοίγει για γράψιμο
                file.write(r.content)
    if not os.path.exists("cifar-10-batches-py"):
        print("Extracting dataset...")
        with tarfile.open(archive_name, "r:gz") as tar:
            tar.extractall()
```

2.3. Φόρτωση και προεπεξεργασία του dataset

Η βοηθητική συνάρτηση *" unpickle(file) "* διαβάζει τα δεδομένα από αρχεία που είναι αποθηκευμένα σε δυαδική μορφή και χρησιμοποιείται για τη φόρτωση (*pickle*) των δεδομένων του CIFAR-10 ύστερα από την λήψη και αποθήκευση τους στο φάκελο *"./cifar-10-batches-py"* με χρήση της συνάρτησης *" download_and_extract_cifar10() "* της παραγράφου *2.2.* Τα δεδομένα εκπαίδευσης περιλαμβάνονται σε πέντε batches (αρχεία data_batch_1 έως data_batch_5), με κάθε **batch** να περιέχει τις εικόνες και τις αντίστοιχες ετικέτες τους, δηλαδή τις κατηγορίες/κλάσεις στις οποίες ανήκουν. Οι εικόνες φορτώνονται και αποθηκεύονται στον πίνακα ***x*** και οι ετικέτες στον πίνακα ***y***.

Έπειτα πραγματοποιείται η **κανονικοποίηση**, που μετατρέπει τις τιμές των εικόνων από το εύρος [0, 255] στο εύρος [0, 1], διαιρώντας με το μέγιστο (255.0). Αυτή η διαδικασία μειώνει τη διακύμανση μεταξύ των εισόδων και καθιστά τα δεδομένα πιο κατάλληλα για την εκπαίδευση του SVM. Αντίστοιχη επεξεργασία υφίστανται και τα δεδομένα ελέγχου που περιέχονται στο αρχείο *"test_batch"*. Η κανονικοποίηση βοηθά στην ομαλότερη εκτέλεση του **kernel** (βλ. *2.5*), καθώς οι υπολογισμοί των εσωτερικών γινομένων ή των εκθετικών όρων που χρησιμοποιούνται, ειδικά στους μη γραμμικούς πυρήνες, παραμένουν εντός αριθμητικά σταθερών ορίων, αποτρέποντας πιθανή αριθμητική αστάθεια (πχ. overflow). Τα κανονικοποιημένα δεδομένα αποθηκεύονται ως **NumPy Arrays** στις μεταβλητές *"x_train"* (δείγματα εκπαίδευσης) και *"x_test"* (δείγματα ελέγχου). Τελικά, η συνάρτηση επιστρέφει τις μεταβλητές *"x_train"* , *"y_train"* (ετικέτες των x_train), *"x_test"*, *"y_test"* (ετικέτες των x_test).

```python
# Φόρτωση των δεδομένων του dataset
def load_cifar10():
    def unpickle(file):
        with open(file, 'rb') as fo:
            data = pickle.load(fo, encoding='bytes')
        return data

    download_and_extract_cifar10()
    data_path = "./cifar-10-batches-py/"
    x, y = [], []

    for i in range(1, 6):
        batch = unpickle(os.path.join(data_path, f"data_batch_{i}"))
        x.append(batch[b'data'])
        y.extend(batch[b'labels'])

    x_train = np.vstack(x).astype(np.float32) / 255.0 # Κανονικοποίηση τιμών
    y_train = np.array(y)

    test_batch = unpickle(os.path.join(data_path, "test_batch"))
    x_test = test_batch[b'data'].astype(np.float32) / 255.0
    y_test = np.array(test_batch[b'labels'])

    return x_train, y_train, x_test, y_test
```

2.4. PCA – Μείωση διάστασης δεδομένων

Η συνάρτηση *"pca(X, n_components=0.9)"* εφαρμόζει την τεχνική **Ανάλυσης Κύριων Συνιστωσών (PCA)** στο dataset και στηρίζεται στα ιδιοδιανύσματα του πίνακα συνδιασποράς που έχουμε να χρησιμοποιήσουμε. Έχει σκοπό την συμπίεση των δεδομένων, διατηρώντας παράλληλα τη μέγιστη δυνατή πληροφορία. Θέλει δηλαδή να βρει έναν υποχώρο διάστασης $d < p$ (p η διάσταση του χώρου δεδομένων) που εξηγεί όσο το δυνατόν περισσότερο τη διασπορά των σημείων. Η συνάρτηση δέχεται ως είσοδο:

- *"X" : τα δεδομένα που πρόκειται να υποβληθούν σε ανάλυση*
- *"n_components": Το ποσοστό της διατηρούμενης πληροφορίας (variance), με προεπιλεγμένη τιμή 0.91 (91% της διακύμανσης)*

```
# PCA implementation
def pca(X, n_components=0.91):
    X_mean = np.mean(X, axis=0)
    X_centered = X - X_mean # κεντράρισμα των τιμών

    # Υπολογισμός πίνακα συνδιασποράς, ιδιοτιμών και ιδιοδιανυσμάτων
    # > rowvar=false: οι στήλες του πίνακα αντιπροσωπεύουν τα χαρακτηριστικά
    cov_matrix = np.cov(X_centered, rowvar=False)
    eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

    # Ταξινόμηση των ιδ/μων για επιλογή των σημαντικότερων ιδ/μών και ιδ/των
    # με [::-1] ώστε η σειρά να είναι φθίνουσα
    sorted_indices = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[sorted_indices]
    eigenvectors = eigenvectors[:, sorted_indices]

    # Υπολογισμός της συνολικής διακύμανσης, της σωρευτικής εξηγούμενης
    # διακύμανσης και των απαιτούμενων κύριων συνιστωσών
    total_variance = np.sum(eigenvalues)
    explained_variance = np.cumsum(eigenvalues) / total_variance
    num_components = np.argmax(explained_variance >= n_components) + 1

    print(f"Επιλέχθηκαν {num_components} συνιστώσες για διατήρηση {n_components*100}% διακύμανσης.")

    # Προβολή στον νέο χώρο χαμηλότερης διάστασης
    selected_eigenvectors = eigenvectors[:, :num_components]
    reduced_X = np.dot(X_centered, selected_eigenvectors)

    return reduced_X, selected_eigenvectors, X_mean
```

Αναλυτικότερα:

I. Για τον υπολογισμό των **ιδιοδιανυσμάτων** (*"eigenvectors"*) του πίνακα συνδιασποράς (*"cov_matrix"*), απαραίτητη διαδικασία αποτελεί το **κεντράρισμα** των τιμών, αφαιρώντας τον **μέσο όρο** από κάθε χαρακτηριστικό (στήλη του X από pixels) του dataset, ώστε η διακύμανση να μετριέται γύρω από το μηδέν. Οι **ιδιοτιμές** (*"eigenvalues"*) αντιπροσωπεύουν την διακύμανση που εξηγεί κάθε ιδιοδιάνυσμα:

```
X_mean = np.mean(X, axis=0)
X_centered = X - X_mean # κεντράρισμα των τιμών

# Υπολογισμός πίνακα συνδιασποράς, ιδιοτιμών και ιδιοδιανυσμάτων
# - rowvar=false: οι στήλες του πίνακα αντιπροσωπεύουν τα χαρακτηριστικά
cov_matrix = np.cov(X_centered, rowvar=False)
eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)
```

II. Οι ιδιοτιμές **ταξινομούνται** σε φθίνουσα σειρά με βάση τους δείκτες τους(“sorted_indices”) και αντίστοιχα με αυτές ταξινομούνται και τα ιδιοδιανύσματα. Αυτό συμβαίνει, επειδή θέλουμε να διατηρήσουμε τις **πρώτες κύριες συνιστώσες**, που εξηγούν τη **μεγαλύτερη διακύμανση** στα δεδομένα και αντιστοιχούν στα **ιδιοδιανύσματα** που σχετίζονται με τις **μεγαλύτερες ιδιοτιμές**:

```
# Ταξινόμηση των ιδ/μων για επιλογή των σημαντικότερων ιδ/μων και ιδ/των
# με [::-1] ώστε η σειρά να είναι φθίνουσα
sorted_indices = np.argsort(eigenvalues)[::-1]
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]
```

III. Υπολογίζεται η **συνολική διακύμανση** των δεδομένων (“*total_variance*”), η **σωρευτική** (cumulative) εξηγούμενη διακύμανση (“*explained_variance*”) για να δούμε πόση διακύμανση εξηγούν οι κύριες συνιστώσες και ο **αριθμός των κύριων συνιστωσών** (“*num_components*”) που χρειάζονται ώστε να διατηρείται το καθορισμένο ποσοστό της συνολικής διακύμανσης:

```
# Υπολογισμός της συνολικής διακύμανσης, της σωρευτικής εξηγούμενης
# διακύμανσης και των απαιτούμενων κύριων συνιστωσών
total_variance = np.sum(eigenvalues)
explained_variance = np.cumsum(eigenvalues) / total_variance
num_components = np.argmax(explained_variance >= n_components) + 1
```

IV. Επιλέγονται τα πρώτα “*num_components*” ιδιοδιανύσματα που αντιστοιχούν στις πρώτες κύριες συνιστώσες, και τα δεδομένα προβάλλονται στον νέο χώρο χαμηλότερης διάστασης. Η συνάρτηση επιστρέφει:

- *“reduced_X”: Τα δεδομένα στον νέο, μειωμένο χώρο διάστασης, μετά την εφαρμογή του PCA*
- *“selected_eigenvectors”: Τα επιλεγμένα ιδιοδιανύσματα (οι κύριες συνιστώσες) που χρησιμοποιήθηκαν για τη μείωση της διάστασης*
- *“X_mean”: Ο αρχικός μέσος όρος των δεδομένων, που χρησιμοποιείται για να επαναφέρει τα δεδομένα στην αρχική τους κλίμακα (αν χρειαστεί)*

```
# Προβολή στον νέο χώρο χαμηλότερης διάστασης
selected_eigenvectors = eigenvectors[:, :num_components]
reduced_X = np.dot(X_centered, selected_eigenvectors)

return reduced_X, selected_eigenvectors, X_mean
```

Οι μεταβλητές *“selected_eigenvectors” και “X_mean”* θα χρειαστούν στην **εφαρμογή PCA στα δείγματα ελέγχου.**

Η μείωση της διάστασης μέσω PCA βελτιώνει την εκπαίδευση του SVM, διότι απομακρύνει θόρυβο και περιττές πληροφορίες και μειώνει την πολυπλοκότητα των υπολογισμών.

2.5. Kernels

Στην παρούσα εργασία καλούμαστε να εξετάσουμε την απόδοση του SVM με χρήση διαφόρων πυρήνων στην εκπαίδευση του, ώστε να πραγματοποιεί κατηγοριοποίηση των εικόνων του dataset CIFAR-10. Οι **πυρήνες** (**kernels**) που χρησιμοποιήθηκαν είναι:

- ***Γραμμικός*** *Πυρήνας (Linear Kernel)*
- ***Πολυωνυμικός*** *Πυρήνας (Polynomial Kernel)*
- *Πυρήνας* ***RBF (****Gaussian Kernel)*

Έχοντας βασιστεί στην ιδέα του **Kernel Trick**, οι kernels είναι συναρτήσεις που μετασχηματίζουν τα δεδομένα από τον αρχικό τους χώρο σε έναν **χώρο χαρακτηριστικών υψηλότερης διάστασης** (Hilbert space), όπου είναι πιθανότερο να είναι γραμμικά διαχωρίσιμα. Αυτός ο μετασχηματισμός δίνει την δυνατότητα να λύνουμε γραμμικά με **SVM**, προβλήματα δεδομένων που είναι **μη γραμμικά διαχωρίσιμα**, όπως του CIFAR-10, καθώς η '***ομοιότητα***' των διανυσμάτων στον χώρο υψηλής διάστασης έχει ένα προς ένα αντιστοιχία με την '*ομοιότητα*' των διανυσμάτων στον αρχικό χώρο, αποφεύγοντας τον ρητό υπολογισμό των συντεταγμένων τους στον νέο χώρο. Αναλυτικότερα:

i. **Linear Kernel**:

Ο γραμμικός πυρήνας ["*linear_kernel(x1, x2)*"] χρησιμοποιεί το **εσωτερικό γινόμενο** των διανυσμάτων χαρακτηριστικών (δείγματα): $K(x_1, x_2) = \langle x_1, x_2 \rangle$. Αυτός ο πυρήνας είναι αποτελεσματικός όταν τα δεδομένα είναι γραμμικά διαχωρίσιμα, κάτι το οποίο δεν ισχύει, οπότε αναμένουμε χαμηλότερη απόδοση συγκριτικά με τους υπόλοιπους πυρήνες.

```
def linear_kernel(x1, x2):
    return np.dot(x1, x2)
```

ii. **Polynomial Kernel**:

Ο πολυωνυμικός πυρήνας ["*polynomial_kernel(x1, x2, degree=…)*"] μπορεί να χειριστεί πιο πολύπλοκα, μη γραμμικά προβλήματα, καθώς προσθέτει πολυωνυμικές σχέσεις μεταξύ των χαρακτηριστικών: $K(x_1, x_2) = (\langle x_1, x_2 \rangle + 1)^d$. Έτσι, εισάγεται η παράμετρος "*degree*" (βαθμός του πολυωνύμου), η οποία δοκιμάστηκε για διάφορες τιμές (π.χ. 2, 3) που αναμένουμε να οδηγήσουν σε καλύτερα αποτελέσματα από τον γραμμικό πυρήνα:

```
def polynomial_kernel(x1, x2, degree=2):
    return (np.dot(x1, x2) + 1) ** degree
```

Ωστόσο, οι πολυωνυμικοί πυρήνες έχουν το πρόβλημα ότι όσο υψηλότερη είναι η δύναμη '*d*', τόσο η συνάρτηση $\langle x_1, x_2 \rangle + 1$:

- Τείνει στο ∞ αν $\langle x_1, x_2 \rangle + 1 > 1$
- Τείνει στο 0 αν $\langle x_1, x_2 \rangle + 1 < 1$

Αν συμβαίνει κάποιο από αυτά αργεί η βελτιστοποίηση και παρουσιάζεται numerical instability.

iii. **RBF (Gaussian) Kernel**:

Ο πυρήνας RBF [“*rbf_kernel(x1, x2, gamma=…)*”] είναι αυτός που χρησιμοποιείται κατά κόρον στη θέση της μη γραμμικότητας για μη γραμμικά προβλήματα με SVM. Υπολογίζει την ‘ομοιότητα’ των δειγμάτων βάσει της ευκλείδειας απόστασής τους και του $\gamma(gamma)$: $K(x_1, x_2) = e^{-\gamma\|x_1 - x_2\|^2}$, όπου “*gamma*”, η υπερπαράμετρος που καθορίζει το πλάτος της συνάρτησης Gauss, για την οποία δοκιμάστηκαν διάφορες τιμές (π.χ. 0.001, 0.01, 0.1):

- Όταν x_1 και x_2 είναι κοντά στον χώρο χαρακτηριστικών (δηλ. η απόσταση $\|x_1 - x_2\|$ είναι μικρή), τότε το $K(x_1, x_2)$ είναι κοντά στο 1
- Όταν η απόσταση $\|x_1 - x_2\|$ είναι μεγάλη, τότε το $K(x_1, x_2)$ τείνει στο 0

```
def rbf_kernel(x1, x2, gamma=0.001):
    return np.exp(-gamma * np.linalg.norm(x1 - x2)**2)
```

Στο σημείο αυτό αξίζει να αναφερθεί ότι οι πυρήνες RBF προτιμώνται, διότι δεν εμφανίζουν το πρόβλημα που αναφέρθηκε για τους πολυωνυμικούς πυρήνες (βλ. *2.5.ii*), ωστόσο αν δεν επιλέξουμε κατάλληλα τον συνδυασμό “*gamma*” και *“C”* μπορούμε να έχουμε φοβερές υπερεκπαιδεύσεις.

---

*Η **κλάση** “**SVM**” υλοποιεί ένα Support Vector Machine (SVM) το οποίο εκπαιδεύεται για ταξινόμηση δεδομένων. Οι μέθοδοι που παρουσιάζονται στις παραγράφους 2.6 έως 2.8 ανήκουν σε αυτήν την κλάση:*

2.6. Αρχικοποίηση παραμέτρων κλάσης

Η μέθοδος “*__init__(self, kernel, C=…)*” αρχικοποιεί το SVM:

- *“kernel”: Η συνάρτηση πυρήνα που καθορίζει το είδος του μετασχηματισμού (π.χ. γραμμικός, πολυωνυμικός, RBF).*
- *“C”: Υπερπαράμετρος που ελέγχει την ισορροπία μεταξύ της ορθότητας ταξινόμησης (χαμηλό σφάλμα εκπαίδευσης) και της αποφυγής υπερεκπαίδευσης (γενίκευση). Μέσω αυτής ‘ζυγίζουμε’ τα σφάλματα, δηλαδή πόσο μας ενδιαφέρει ή όχι το γεγονός να έχει σφάλματα η μηχανή μας.*

```
def __init__(self, kernel, C=0.1):
    self.kernel = kernel # συνάρτηση Kernel
    self.C = C # υπερ/μετρος που ζυγίζει τα σφάλματα
```

Όσον αφορά τις **τιμές** του *"C"* ισχύουν τα παρακάτω:

- **Μεγάλες** τιμές του *"C"* δίνουν μεγάλη έμφαση στη σωστή ταξινόμηση όλων των δειγμάτων εκπαίδευσης και το SVM προσπαθεί να ελαχιστοποιήσει το πλήθος των λαθών, επιλέγοντας ένα στενότερο περιθώριο, με αποτέλεσμα ίσως ένα πιο πολύπλοκο μοντέλο που είναι ευάλωτο σε **υπερπροσαρμογή** (overfitting).
- **Μικρές** τιμές του *"C"* επιτρέπουν μεγαλύτερη ανοχή σε σφάλματα ταξινόμησης ή σημεία που παραβιάζουν το περιθώριο και το SVM προτιμά ένα ευρύτερο περιθώριο, ακόμα και αν αυτό συνεπάγεται μερικά λάθη στα δεδομένα εκπαίδευσης όντας λιγότερο ευαίσθητο σε outliers, με αποτέλεσμα είναι ένα πιο απλό μοντέλο με μεγαλύτερη πιθανότητα **γενίκευσης** (ίσως και underfitting).

2.7. Εκπαίδευση SVM

Η μέθοδος *"train(self, x_train, y_train)"* χρησιμοποιεί τα δεδομένα εκπαίδευσης και τον πίνακα Kernel για να υπολογίσει τα **βάρη** των **Support Vectors** (**πολλαπλασιαστές Lagrange**) και το **bias** :

- *"x_train"*: τα δείγματα εκπαίδευσης
- *"y_train"*: οι ετικέτες (labels) των δειγμάτων εκπαίδευσης

```
def train(self, x_train, y_train):
    self.kernel_matrix = np.zeros((len(x_train), len(x_train))) # του Hilbert space
    self.x_train = x_train
    self.y_train = y_train

    # Υπολογισμός του πίνακα H (kernel matrix) και προσθήκη στην διαγώνιο μίας πολύ μικρής τιμής
    # (regularization) σκοπεύοντας να μην χάσω ακρίβεια στους υπολογισμούς μου
    for i in range(len(x_train)):
        for j in range(len(x_train)):
            self.kernel_matrix[i, j] = y_train[i] * y_train[j] * self.kernel(x_train[i], x_train[j])
    self.kernel_matrix += np.eye(self.kernel_matrix.shape[0]) * 1e-10

    # Επίλυση του προβλήματος βελτιστοποίησης
    c = np.ones(len(x_train))
    self.a = cp.Variable(len(x_train))
    objective = cp.Minimize(0.5 * cp.quad_form(self.a, self.kernel_matrix) - c.T @ self.a)
    constraints = [self.a >= 0, self.a <= self.C] # περιορισμοί
    problem = cp.Problem(objective, constraints)
    problem.solve() # επιστρέφει το δ/μα των πολ/στων Lagrange
    self.a = self.a.value

    epsilon = 1e-6 # ανοχή για τον εντοπισμό των SVs στο περιθώριο
    svi = np.where((self.a > epsilon) & (self.a < self.C - epsilon))[0]
    print("Σύνολο των support vectors:", len(svi))

    # Υπολογισμός του bias
    if len(svi) > 0:
        self.b = 0;
        biases = []
        for sv in svi:
            b_sv = y_train[sv]
            for i in range(len(x_train)):
                b_sv -= self.a[i] * y_train[i] * self.kernel(x_train[sv], x_train[i])
            biases.append(b_sv)
        self.b = np.mean(biases)
        print("Bias (b):", self.b)
    else:
        print("Δεν βρέθηκαν support vectors στο περιθώριο!")
```

Αναλυτικότερα:

I. Ο "*kernel_matrix*" (H) αποτελεί πίνακα $N \times N$ διαστάσεων, όπου N ο αριθμός των δειγμάτων εκπαίδευσης (*"x_train"*), στον οποίο αποθηκεύεται στοιχείο προς στοιχείο για κάθε ζεύγος δειγμάτων x_i, x_j το γινόμενο $y_i \cdot y_j \cdot k(x_i, x_j)$, όπου y_i, y_j οι ετικέτες των αντίστοιχων δειγμάτων και $k(x_i, x_j)$ η τιμή της συνάρτησης Kernel (βλ. *2.5, σελ.6*) που υπολογίζει την ομοιότητα μεταξύ των x_i *και* x_j . Ο πίνακας H ξεφεύγει σε πλήθος και κατά συνέπεια προκύπτουν προβλήματα όσον αφορά την δέσμευση μνήμης.

Έπειτα από τον υπολογισμό του πίνακα H, προστίθεται στη διαγώνιο μία πολύ μικρή τιμή (regularization), ώστε να αυξηθεί η ευστάθεια επίλυσης του προβλήματος βελτιστοποίησης.

(*Μπορεί πολλά από τα δείγματα να είναι γραμμικώς εξαρτημένα και άρα ο* H *να μην αντιστρέφεται ή να έχει πολύ 'κακή κατάσταση'*)

```
# Υπολογισμός του πίνακα H (kernel matrix) και προσθήκη στην διαγώνιο μίας πολύ μικρής τιμής
# (regularization) σκοπεύοντας να μην χάσω ακρίβεια στους υπολογισμούς μου
for i in range(len(x_train)):
    for j in range(len(x_train)):
        self.kernel_matrix[i, j] = y_train[i] * y_train[j] * self.kernel(x_train[i], x_train[j])
self.kernel_matrix += np.eye(self.kernel_matrix.shape[0]) * 1e-10
```

II. Η **αντικειμενική συνάρτηση** που έχουμε να **ελαχιστοποιήσουμε** στο **dual space** του SVM (πρόβλημα βελτιστοποίησης) είναι η : $\frac{1}{2}\boldsymbol{\alpha^T H \alpha} - \boldsymbol{c^T a}$, όπου "$c$" ένα διάνυσμα που έχει παντού μέσα μονάδες και "a" ένα διάνυσμα που αντιστοιχεί στους πολλαπλασιαστές Lagrange και είναι αυτό που επιστρέφει το πρόβλημα βελτιστοποίησης, για την αριθμητική επίλυση του οποίου χρησιμοποιείται η βιβλιοθήκη "*cvxpy*" (convex optimization). Οι **περιορισμοί** για το "a" είναι: $0 \leq a_i \leq C$, όπου "C" η υπερπαράμετρος που αναφέρθηκε στην παράγραφο *2.6, σελ.7.*

Ο **αριθμός** των **support vectors** (SVs) αντιστοιχεί στα μη μηδενικά "a", ωστόσο ορίζουμε την σταθερά "*epsilon*", διότι ποτέ δεν υπολογίζεται κάτι ακριβώς μηδέν), αλλά έχει να κάνει με την ακρίβεια της μηχανής που χρησιμοποιείται.

```
# Επίλυση του προβλήματος βελτιστοποίησης
c = np.ones(len(x_train))
self.a = cp.Variable(len(x_train))
objective = cp.Minimize(0.5 * cp.quad_form(self.a, self.kernel_matrix) - c.T @ self.a)
constraints = [self.a >= 0, self.a <= self.C] # περιορισμοί
problem = cp.Problem(objective, constraints)
problem.solve() # επιστρέφει το δ/μα των πολ/στων Lagrange
self.a = self.a.value

epsilon = 1e-6 # ανοχή για τον εντοπισμό των SVs
svi = np.where((self.a > epsilon) & (self.a < self.C - epsilon))[0]
print("Σύνολο των support vectors:", len(svi))
```

III. Για κάθε Support Vector (SV) που αντιστοιχεί σε μη μηδενικό πολλαπλασιαστή Lagrange υπολογίζουμε το **bias** του "*b_sv*": $\boldsymbol{b_{sv} = y_{sv} - \sum_i a_i \cdot y_i \cdot k(x_{sv}, x_i)}$, όπου y_{sv} η ετικέτα του SV, a_i οι πολλαπλασιαστές Lagrange των άλλων δειγμάτων και $k(x_{sv}, x_i)$ η τιμή του Kernel μεταξύ του *sv* και όλων των άλλων δειγμάτων. Για καλύτερο numerical stability και μεγαλύτερη ακρίβεια παίρνουμε τον μέσο όρο όλων των biases που υπολογίσαμε από τα SVs, καθώς βρίσκονται στα όρια του περιθωρίου (βλ. *2.8, σελ.11*).

```
# Υπολογισμός του bias
if len(svi) > 0:
    self.b = 0;
    biases = []
    for sv in svi:
        b_sv = y_train[sv]
        for i in range(len(x_train)):
            b_sv -= self.a[i] * y_train[i] * self.kernel(x_train[sv], x_train[i])
        biases.append(b_sv)
    self.b = np.mean(biases)
    print("Bias (b):", self.b)
else:
    print("Δεν βρέθηκαν support vectors στο περιθώριο!")
```

2.8. Προβλέψεις

Η μέθοδος "*predict(self, x)*" είναι υπεύθυνη για τις προβλέψεις, δηλαδή την κλάση/κατηγορία στην οποία θεωρεί ότι ανήκει κάθε δείγμα:

- "*x*": Το σύνολο δεδομένων που θέλουμε να ταξινομήσουμε. Είναι ένας πίνακας διαστάσεων $M \times d$, όπου M ο αριθμός των δειγμάτων προς ταξινόμηση και d η διάσταση του χώρου των χαρακτηριστικών

```
# Προβλέψεις
def predict(self, x):
    ker_matrix = np.zeros((len(x), len(self.x_train)))
    for i in range(len(x)):
        for j in range(len(self.x_train)):
            ker_matrix[i, j] = self.y_train[j] * self.kernel(x[i], self.x_train[j])

    predictions = np.dot(ker_matrix, self.a) + self.b
    return predictions
```

Ο "*kerl_matrix*" αποτελεί πίνακα $M \times N$ διαστάσεων, όπου N ο αριθμός των δειγμάτων εκπαίδευσης ("*x_train*"), στον οποίο αποθηκεύεται στοιχείο προς στοιχείο για κάθε ζεύγος δειγμάτων $x_i, x_{train,j}$ το γινόμενο $y_{train,j} \cdot k(x_i, x_{train,j})$, όπου $y_{train,j}$ η ετικέτα του δείγματος j και $k(x_i, x_{train,j})$ τιμή της συνάρτησης Kernel (βλ. *2.5, σελ.6*) που υπολογίζει την ομοιότητα μεταξύ των x_i *και* $x_{train,j}$.

Το αποτέλεσμα $f(x_i)$ (score) για κάθε δείγμα στο σύνολο "x", δηλώνει την **απόσταση** του δείγματος x_i από το διαχωριστικό υπερεπίπεδο $(f(x) = 0)$ στον Hilbert space και υπολογίζεται ως: $\mathbf{f(x_i) = \sum_{j=1}^{N} a_j \cdot y_j \cdot k(x_i, x_{train,j}) + b}$, όπου y_j η ετικέτα του δείγματος j, $k(x_i, x_{train,j})$ η τιμή του Kernel μεταξύ του δείγματος x_i και του $x_{train,j}$, , a_j οι πολλαπλασιαστές Lagrange και b το bias (προστίθεται για την μετατόπιση της απόφασης του μοντέλου) που υπολογίστηκαν κατά την εκπαίδευση. Θεωρητικά ισχύει ότι:

- Αν $f(x) > 0$: Το σημείο είναι στην πλευρά της κλάσης $+1$
- Αν $f(x) < 0$: Το σημείο είναι στην πλευρά της κλάσης -1
- Για τα SVs $|f(x)| = 1$, δηλαδή βρίσκονται στα όρια του περιθωρίου

Η μέθοδος επιστρέφει το:

- *"predictions": Πίνακας μήκους M όπου κάθε στοιχείο αντιστοιχεί στο $f(x)$ για ένα δείγμα του συνόλου δεδομένων "x".*

2.9. Παραδείγματα ορθής και εσφαλμένης κατηγοριοποίησης

Η συνάρτηση "*print_predictions_summary(test_labels, final_predictions, num_examples)*" εμφανίζει παραδείγματα σωστής και εσφαλμένης κατηγοριοποίησης, βοηθώντας στην κατανόηση των επιδόσεών του.

- ***"test_labels": Πίνακας με τις πραγματικές ετικέτες των δειγμάτων του test set.***
- *"final_predictions": Πίνακας με τις ετικέτες που προβλέφθηκαν από το μοντέλο.*
- *"num_examples": Προαιρετικό όρισμα που καθορίζει πόσα παραδείγματα από τις σωστές και εσφαλμένες προβλέψεις θα εμφανιστούν.*

Μέσω της "*vp.where*" εντοπίζονται τα "correct_*indices*" για τις σωστές προβλέψεις και τα "correct_*indices*" για τις εσφαλμένες προβλέψεις:

```
# Συνάρτηση που παρουσιάζει παραδείγματα ορθής και εσφαλμένης κατηγοριοποίησης
def print_predictions_summary(test_labels, final_predictions, num_examples=5):
    correct_indices = np.where(final_predictions == test_labels)[0]
    incorrect_indices = np.where(final_predictions != test_labels)[0]

    print("\nΠαραδείγματα ορθής κατηγοριοποίησης:")
    for i in range(min(num_examples, len(correct_indices))):
        index = correct_indices[i]
        print(f"Index: {index}, Πραγματική ετικέτα: {test_labels[index]}, Πρόβλεψη ετικέτας: {final_predictions[index]}")

    print("\nΠαραδείγματα εσφαλμένης κατηγοριοποίησης:")
    for i in range(min(num_examples, len(incorrect_indices))):
        index = incorrect_indices[i]
        print(f"Index: {index}, Πραγματική ετικέτα: {test_labels[index]}, Πρόβλεψη ετικέτας: {final_predictions[index]}")
```

- Παρακάτω παρουσιάζεται ένα παράδειγμα ορθής και εσφαλμένης κατηγοριοποίησης των δειγμάτων του test set με δείκτες 0, 1, 2, 3:

```
Παραδείγματα ορθής κατηγοριοποίησης:
Index: 0, Πραγματική ετικέτα: 1, Πρόβλεψη ετικέτας: 1
Index: 1, Πραγματική ετικέτα: 2, Πρόβλεψη ετικέτας: 2
```

```
Παραδείγματα εσφαλμένης κατηγοριοποίησης:
Index: 2, Πραγματική ετικέτα: 5, Πρόβλεψη ετικέτας: 3
Index: 3, Πραγματική ετικέτα: 0, Πρόβλεψη ετικέτας: 1
```

- Τα παραπάνω επιβεβαιώνονται με την εμφάνιση των αποτελεσμάτων για τις προβλέψεις κάθε κλάσεις (βλ. *2.8, σελ.10*), των οποίων ο υπολογισμός πραγματοποιείται στην συνάρτηση "*main()*" (βλ. *2.11, σελ.14*):

```
Προβλέψεις για την κλάση 0:
 [ -2.65887485  -2.13085867  -1.26257378  -1.36294843
```

```
Προβλέψεις για την κλάση 1:
 [ 7.87716070e-01 -1.67329503e+00 -3.29509202e+00  5.01065221e-02
```

```
Προβλέψεις για την κλάση 2:
 [-4.24269880e+00  4.23438445e-01 -3.03626909e+00 -1.56143158e+00
```

```
Προβλέψεις για την κλάση 3:
 [-8.41229400e-01 -2.16939752e+00 -8.58060961e-01 -2.34437689e-01
```

```
Προβλέψεις για την κλάση 4:
 [-3.88030826e+00 -2.05084368e-01 -2.24265266e+00 -6.80397087e-01
```

```
Προβλέψεις για την κλάση 5:
 [ 4.55766880e-01 -1.93733421e+00 -1.10027172e+00 -1.85571150e+00
```

```
Προβλέψεις για την κλάση 6:
 [-2.5801178  -2.42523704 -2.24656849 -0.59177914
```

```
Προβλέψεις για την κλάση 7:
 [-1.97794748 -1.67892278 -4.36705931 -1.38414525
```

```
Προβλέψεις για την κλάση 8:
 [-2.21987091e+00 -1.20965861e+00 -5.77093880e+00 -1.69915430e+00
```

```
Προβλέψεις για την κλάση 9:
 [ -1.5463349   -1.55543897  -3.53074913  -2.08796275
```

2.10. Υπολογισμός ακρίβειας

Η συνάρτηση *"compute_accuracy"* υπολογίζει την ακρίβεια (accuracy) του μοντέλου, δηλαδή το ποσοστό των προβλέψεων που ήταν σωστές, την οποία επιστρέφει ως αριθμό κινητής υποδιαστολής (float).

Χρησιμοποιείται η "*np.mean*" για τον υπολογισμό του μέσου όρου.

- ***"true_labels": Πίνακας με τις πραγματικές ετικέτες των δειγμάτων του test set.***

- *"final_predictions": Πίνακας με τις προβλέψεις του μοντέλου για τα αντίστοιχα δείγματα.*

```
# Συνάρτηση για τον υπολογισμό της ακρίβειας
def compute_accuracy(true_labels, predictions):
    return np.mean(true_labels == predictions) * 100
```

2.11. Κύριο πρόγραμμα

Η συνάρτηση "*main()*" αποτελεί το κύριο πρόγραμμα για την εκπαίδευση και αξιολόγηση ενός **SVM** πάνω στο dataset **CIFAR-10**. Αναλυτικότερα:

I. Φορτώνονται τα δεδομένα εκπαίδευσης ("*train_data")* και ελέγχου ("*train_data")* μαζί με τις αντίστοιχες ετικέτες ("*train_labels", "test_labels")* από τη συνάρτηση "*load_cifar10()"* (βλ. *2.3, σελ.3*).

Εφαρμόζεται **PCA** στα δεδομένα εκπαίδευσης (βλ. *2.4, σελ.4*) και χρησιμοποιούνται τα επιλεγμένα ιδιοδιανύσματα ("*selected_eigenvectors*") και ο μέσος όρος ("*train_data_mean*") των δεδομένων εκπαίδευσης για την εφαρμογή PCA στα δεδομένα ελέγχου.

```
train_data, train_labels, test_data, test_labels = load_cifar10_data()
train_size = train_data.shape[0]
test_size = test_data.shape[0]
print(f"Μέγεθος συνόλου δειγμάτων εκπαίδευσης: {train_size}")
print(f"Μέγεθος συνόλου δειγμάτων ελέγχου: {test_size}")

print("Εφαρμογή PCA στα δεδομένα εκπαίδευσης...")
train_data_pca, selected_eigenvectors, train_data_mean = pca(train_data, 0.91)

# Μετασχηματισμός του test set με την ίδια PCA
test_data_centered = test_data - train_data_mean
test_data_pca = np.dot(test_data_centered, selected_eigenvectors)
```

II. Στο σύνολο των δειγμάτων εκπαίδευσης εφαρμόστηκε **batch training** με "batch_size" να είναι το μέγεθος του κάθε batch, το οποίο περιορίστηκε στον αριθμό 1000, λόγω **περιορισμένης μνήμης** και **υπολογιστικών πόρων,** και *"batches_number"* ο συνολικός αριθμός τους. Έτσι, αντί να επεξεργαζόμαστε όλα τα δεδομένα μαζί, χωρίζουμε το σύνολο εκπαίδευσης σε μικρότερα υποσύνολα με σκοπό το μέγεθος των υπολογισμών να είναι διαχειρίσιμο.

Χωρίς batch training και για αριθμό δειγμάτων εκπαίδευσης περίπου 30000 και πάνω εμφανίζεται **σφάλμα** εξαιτίας χρήσης όλης της διαθέσιμης μνήμης **RAM**, το οποίο προκαλείται κατά τη εκπαίδευση του SVM (βλ. *2.7, σελ.9*), κυρίως στη δημιουργία του **πίνακα Kernel** και κατ' επέκταση στην επίλυση του προβλήματος βελτιστοποίησης.

**Θα μπορούσαν να εφαρμοστούν τεχνικές για καλύτερα αποτελέσματα, όπως έλεγχος ισοκατανομής των δειγμάτων από κάθε κλάση σε κάθε batch, ώστε να υπάρχει ισορροπία ή εκπαίδευση στα 'πιο δύσκολα' δείγματα , δηλαδή αυτά που είναι πιο κοντά στο περιθώριο*

Αρχικοποιούνται οι μεταβλητές:

- "*predictions_for_each_label_test*": οι προβλέψεις στο test set, για κάθε κλάση(label)
- "*batch_predictions_test*": οι προβλέψεις στο test set, για κάθε batch των κλάσεων
- "*predictions_for_each_label_train*": οι προβλέψεις στο training set, για κάθε κλάση
- "*batch_predictions_training*": οι προβλέψεις για κάθε δείγμα του training set που αποθηκεύονται κατά την διάρκεια του batch training

```
# Batches
batch_size = 1000
batches_number = train_size // batch_size

# Αρχικοποίηση μεταβλητών
predictions_for_each_label_test = np.zeros((10, test_size))
batch_predictions_test = np.zeros((batches_number, test_size))
predictions_for_each_label_train = np.zeros((10, batches_number * batch_size))
batch_predictions_train = np.zeros(batches_number * batch_size)
```

III. Εκτελείται ένας διπλός βρόχος (*for-loop)* με σκοπό τον υπολογισμό των προβλέψεων και την εκπαίδευση ενός μοντέλου SVM για κάθε batch στην κάθε κλάση ($label \in \{0, 1, \dots, 9\}$). Για το SVM ορίζεται το είδος πυρήνα (kernel) που θα χρησιμοποιηθεί (βλ. *2.5*) και ορίζεται μία τιμή για την υπερπαράμετρο "*C*" (βλ. *2.6, σελ.7*).

Η **ταξινόμηση** που χρησιμοποιείται είναι η '**ONE-vs-ALL'** και βασίζεται στην ιδέα δημιουργίας ενός δυαδικού ταξινομητή SVM για κάθε κλάση, ο οποίος αποφασίζει αν το δείγμα ανήκει στην κλάση +1 (αυτή που μας ενδιαφέρει) ή στην κλάση -1 (όλες οι υπόλοιπες). Μέσω της "*np.where*" τα δείγματα που ανήκουν στην κλάση που εξετάζεται λαμβάνουν την ετικέτα +1, ενώ τα δείγματα των υπολοίπων κλάσεων λαμβάνουν την ετικέτα -1.

Όταν ένα νέο δείγμα x πρέπει να ταξινομηθεί, περνάει και από τα 10 SVMs. Κάθε SVM, μέσω της μεθόδου "svm.*predict(…)*" (βλ. *2.8, σελ.10*), επιστρέφει το score $f_j(x)$, με $j = 1, .., 10$, το οποίο αντιπροσωπεύει την απόσταση του δείγματος από υπερεπίπεδο του ταξινομητή της κλάσης j. Με χρήση της "*np.mean*" συνδυάζονται οι

προβλέψεις για κάθε batch, ώστε να δημιουργηθούν οι τελικές προβλέψεις για κάθε κλάση.

```
start_time = time.time() # για την μέτρηση του χρόνου εκπαίδευσης
for label in range(10):
    print(f"Επεξεργασία της κλάσης {label}...")
    for batch in range(batches_number):
        print(f"Εκπαίδευση στο batch {batch + 1}/{batches_number}...")
        # Επιλογή batch δεδομένων
        batch_start = batch * batch_size
        batch_end = (batch + 1) * batch_size
        x_train_batch = train_data_pca[batch_start:batch_end]
        y_train_batch = np.where(train_labels[batch_start:batch_end] == label, 1, -1)

        # Εκπαίδευση SVM στο τρέχον batch
        svm = SVM(kernel=polynomial_kernel, C=0.1)
        svm.train(x_train_batch, y_train_batch)

        # Προβλέψεις για testing και training batch
        batch_predictions_test[batch, :] = svm.predict(test_data_pca)
        batch_predictions_train[(batch * batch_size):((batch + 1) * batch_size)] = svm.predict(x_train_batch)

    # Συνδυασμός προβλέψεων από όλα τα batches
    predictions_for_each_label_test[label, :] = np.mean(batch_predictions_test, axis=0)
    predictions_for_each_label_train[label, :] = batch_predictions_train
    print(f"Προβλέψεις για την κλάση {label}:\n", predictions_for_each_label_test[label])
```

IV. Τελικά, το δείγμα κατατάσσεται στην κλάση με το υψηλότερο $f_j(x)$ με την "*np.argmax*" να επιστρέφει τον δείκτη του στοιχείο με την μεγαλύτερη τιμή. Έτσι, δημιουργούνται οι **τελικές προβλέψεις** και υπολογίζεται η **ακρίβεια** (ποσοστό επιτυχίας), τόσο για το test set ("*final_predictions_test*" και "*testing_accuracy*" αντίστοιχα), όσο και για το training set ("*final_predictions_train*" και "*testing_accuracy*" αντίστοιχα). Αυτά εκτυπώνονται στην οθόνη, όπως και ο **χρόνος εκπαίδευσης**, ενώ καλείται και η συνάρτηση "*print_prediction_summary*" (βλ. 2.9, σελ.11) για την εκτύπωση παραδειγμάτων κατηγοριοποίησης.

```
# Τελικές προβλέψεις και ακρίβεια
final_predictions_test = np.argmax(predictions_for_each_label, axis=0)
final_predictions_train = np.argmax(predictions_for_each_label_train, axis=0)
testing_accuracy = compute_accuracy(test_labels, final_predictions_test)
training_accuracy = compute_accuracy(train_labels, final_predictions_train)

print("Τελικές προβλέψεις:", final_predictions_test)
print(f"Training accuracy: {training_accuracy:.2f}%")
print(f"Testing Accuracy: {testing_accuracy:.2f}%")
print(f"Χρόνος εκπαίδευσης: {time.time() - start_time:.2f} seconds")

# Εκτύπωση παραδειγμάτων κατηγοριοποίησης
print_predictions_summary(test_labels, final_predictions_test)
```

3. Πειραματισμοί και αποτελέσματα

Στη συνέχεια, πραγματοποιήθηκαν πειράματα για διαφορετικά **είδη πυρήνων** (Kernels) και διάφορες τιμές των υπερπαραμέτρων "***C***", "***degree***", "***gamma***"(βλ. *2.6, σελ.7 και 2.5, σελ.8-9 αντίστοιχα*). Για την **PCA** (βλ. *2.4, σελ.4*) επιλέχθηκε να διατηρηθεί 91% της διασποράς, οπότε η νέα διάσταση του χώρου έχει 114 συνιστώσες, ενώ αρχικά είχε 3072 ($32 \times 32 \times 3$).

3.1. Batch training

Για την εκπαίδευση του SVM χρησιμοποιείται batch training, λεπτομέρειες για το οποίο αναφέρονται στην παράγραφο *2.11, σελ.14.* Αν και ιδανικό θα ήταν να υπήρχε ένα batch με όλο το dataset για καλύτερα αποτελέσματα, αυτό είναι αδύνατο λόγω περιορισμένων υπολογιστικών πόρων και μνήμης RAM. Γενικά, όσο μεγαλύτερο το **batch size** τόσο καλύτερη η επίδοση του SVM, κάτι το οποίο διαπιστώθηκε στην πράξη χρησιμοποιώντας ως **παράδειγμα** τον **γραμμικό πυρήνα** (βλ.*2.5, σελ.6*), διότι για το ίδιο batch size είχε τον ταχύτερο χρόνο εκπαίδευσης συγκριτικά με τους άλλους δύο kernels (πολυωνυμικό και RBF) για την εκπαίδευση ολόκληρου του dataset. Για παράδειγμα με batch size [1000]:

Kernels	**Χρόνος εκπαίδευσης**
Linear	14782.94 sec
Polynomial ($2^{ου}$ βαθμού)	20045.15 sec
RBF (γ=0.001)	44401.17 sec

Έτσι, επιλέχθηκε ο Linear για διευκόλυνση, ώστε να εκτελεστούν γρηγορότερα τα πειράματα και τα αποτελέσματα έχουν ως εξής:

Batch size	**Χρόνος εκπαίδευσης**	**Testing Accuracy**
500	12914.11 sec	35.89%
1000	14782.94 sec	39.40%
2000	41333.97 sec	41.16%

Επομένως, επιλέχθηκε να χρησιμοποιηθεί το **batch size [1000]**, συγκρίνοντας τον χρόνο εκπαίδευσης με την ακρίβεια στο test set που προσφέρουν τα διαφορετικά μεγέθη των batches, ως αυτό που προσφέρει το πιο ικανοποιητικό αποτέλεσμα.

*Επιπρόσθετα, αξίζει να αναφερθεί ότι σημαντικό **μειονέκτημα** στον τρόπο με τον οποίο εφαρμόστηκε το batch training, είναι ότι δεν κατανέμονται εξίσου τα δείγματα από κάθε κλάση σε κάθε batch, με αποτέλεσμα την πιθανότητα κάποιο SVM να είναι προκατειλημμένο προς την κλάση για την οποία έχει τα περισσότερα δείγματα και κατά συνέπεια να υπάρχουν ενδεχομένως ανακρίβειες στην απόδοση (ποσοστό επιτυχίας) του SVM.*

3.2. Kernels

Τα είδη πυρήνων που χρησιμοποιήθηκαν είναι:

- ***Γραμμικός*** *Πυρήνας (Linear Kernel)*
- ***Πολυωνυμικός*** *Πυρήνας (Polynomial Kernel)*
- *Πυρήνας* ***RBF (****Gaussian Kernel)*

Αρχικά, δοκιμάστηκε ένα πείραμα με **ολόκληρο** το dataset για **batch size [1000]**, και τιμή για την υπερπαράμετρο "$\boldsymbol{C}" = \mathbf{0.1}$ (βλ. *2.6, σελ.7*):

Kernels	**Χρόνος εκπαίδευσης**	**Training Accuracy**	**Testing Accuracy**	**Support Vectors**
Linear	14782.94 sec	46.86%	39.40%	~350-450
Polynomial ($2^{ου}$ βαθμού)	20045.15 sec	87.20%	47.07%	~300-400
RBF (γ=0.01)	54401.17 sec	92.99%	41.45%	~50-150

Τα αποτελέσματα υποδεικνύουν σημαντικές διαφορές μεταξύ των kernels ως προς την ακρίβεια και το χρόνο εκπαίδευσης:

i. **Linear Kernel**: Το χαμηλό ποσοστό ακρίβειας στο training set (46.86%) υποδηλώνει ότι το μοντέλο δυσκολεύεται να προσαρμοστεί στα δεδομένα με τη γραμμική προσέγγιση. Η μικρή διαφορά μεταξύ training και testing accuracy (39.40%) υποδεικνύει περιορισμένη ικανότητα υπερεκπαίδευσης (overfitting). Το μοντέλο είναι μάλλον υποπροσαρμοσμένο (underfitting), εξαιτίας της περιορισμένης πολυπλοκότητας του linear kernel. Ο χρόνος εκπαίδευσης (14782.94 sec) είναι ο μικρότερος μεταξύ των kernels, γεγονός που καθιστά τον linear kernel γρήγορη αλλά μη αποδοτική επιλογή, καθώς το dataset CIFAR-10 παρουσιάζει υψηλή πολυπλοκότητα.

ii. **Polynomial Kernel** (**$2^{ου}$ βαθμού**): Το πολύ υψηλό ποσοστό ακρίβειας στο training set (91.20%) υποδεικνύει ότι το μοντέλο έχει μεγάλη ικανότητα να προσαρμόζεται στα δεδομένα εκπαίδευσης. Παρότι το testing accuracy (47.07%) είναι καλύτερο από τον linear kernel, παρατηρείται σημαντική διαφορά σε σχέση με την ακρίβεια στο training set, υποδεικνύοντας overfitting. Ο χρόνος εκπαίδευσης (20045.15 sec) είναι μεγαλύτερος από τον linear kernel, κάτι αναμενόμενο λόγω της πολυπλοκότητας του polynomial kernel $2^{ου}$ βαθμού. Επομένως, Ο polynomial kernel 2ου βαθμού είναι πιο κατάλληλος για την ταξινόμηση του CIFAR-10 από τον linear kernel, αλλά υπάρχει περιθώριο βελτίωσης του testing accuracy για τη γενίκευση.

iii. **RBF Kernel**: Το υψηλό ποσοστό στο training accuracy (93.99%) δείχνει ότι το μοντέλο προσαρμόζεται καλά στα δεδομένα εκπαίδευσης, όπως και ο polynomial

kernel. Ωστόσο, η ακρίβεια στα δεδομένα ελέγχου (41.45%) είναι χαμηλότερη από του polynomial kernel 2ου βαθμού, ενώ ο χρόνος εκπαίδευσης είναι πολύ μεγαλύτερος (44401.17 sec). Αυτό υποδεικνύει ότι ο RBF kernel μπορεί να υποφέρει από υπερβολική πολυπλοκότητα και υποβαθμισμένη γενίκευση**.** Παρότι ο RBF kernel είναι πολύ ισχυρός για δεδομένα με υψηλή πολυπλοκότητα, η χρήση του με τις συγκεκριμένες υπερπαραμέτρους "*gamma*" [0.001] και "*C*" [0.1], δεν τον καθιστά την πιο κατάλληλη επιλογή. Αργότερα, θα δούμε ότι αυτό δεν ισχύει.

Συνοψίζοντας:

- Όσον αφορά την **απόδοση** των **Kernels**, ο linear έχει χαμηλή ακρίβεια και ενδείκνυται μόνο αν η ταχύτητα είναι προτεραιότητα. Ο polynomial έχει καλή απόδοση στο training set, αλλά με τάση για overfitting. Ο RBF έχει την βέλτιστη γενίκευση, αλλά με σημαντικό υπολογιστικό κόστος.
- Όσον αφορά την **υπεραπαράμετρο** "***C***" η τιμή [0.1] είναι χαμηλή. Θεωρητικά, το μοντέλο είναι πιο ανεκτικό στα σφάλματα, οπότε βελτιώνεται η γενίκευση, ενώ με υψηλότερες τιμές του "*C*", το μοντέλο τείνει να προσαρμόζεται καλύτερα.
- Όσον αφορά τον **χρόνο εκπαίδευσης,** παρατηρείται ότι αυξάνεται δραματικά με για πιο σύνθετους kernels.

3.3. Υπερπαράμετροι

Παρακάτω πραγματοποιήθηκαν ορισμένοι πειραματισμοί, στον βαθμό που επέτρεπαν οι διαθέσιμοι υπολογιστικοί πόροι και η μνήμη, προκείμενου να αξιολογηθεί η συμπεριφορά του SVM για διάφορες τιμές των **υπερπαραμέτρων** "***degree***" (βλ. *2.5, σελ.6*), "***gamma***" (βλ. *2.5, σελ.7*) και "***C***" (βλ. *2.6, σελ.8*).

I. Βαθμός του πολυωνύμου ("***degree***") για τον **Polynomial kernel**:
Όπως αναφέρθηκε και στη *σελίδα 6* της παραγράφου *2.5* οι πολυωνυμικοί πυρήνες έχουν το πρόβλημα ότι όσο υψηλότερος είναι ο βαθμός του πολυωνύμου $[(\langle x_1, x_2\rangle + 1)^{degree}]$, τόσο η συνάρτηση $\langle x_1, x_2\rangle + 1$:
    - Τείνει στο ∞ αν $\langle x_1, x_2\rangle + 1 > 1$
    - Τείνει στο 0 αν $\langle x_1, x_2\rangle + 1 < 1$

Αν συμβαίνει κάποιο από αυτά αργεί η βελτιστοποίηση και παρουσιάζεται αριθμητική αστάθεια. Αυτό διαπιστώθηκε με τον ακολουθούμενο πειραματισμό για τον **polynomial kernel**, όπου με **batch size [1000]** και $\mathbf{C} = \mathbf{0}.\mathbf{1}$ είχαμε ότι:

Degree	Χρόνος εκπαίδευσης	Training Accuracy	Testing Accuracy
2	20045.15 sec	87.20%	47.07%
3	25766.79 sec	91.80%	43.12%
4	32300.18 sec	93.87%	42.78%

Συνοψίζοντας, συμπεραίνουμε ότι:

- Οι πολυωνυμικοί πυρήνες υψηλότερου βαθμού (3 και 4) αν και βελτιώνουν το training accuracy, λόγω της αυξημένης πολυπλοκότητας τους έχουν αυξημένο χρόνο υπολογισμού (υπολογιστική επιβάρυνση και αριθμητική αστάθεια). Παραάλληλα, υποβαθμίζουν το testing accuracy (μειωμένη απόδοση) και κατά συνέπεια οδγούμαστε σε overfitting και αδυναμία γενίκευσης.

II. Εύρος ("***gamma***") της Gaussian συνάρτησης για τον RBF Kernel:

Για διευκόλυνση του ελέγχου της συμπεριφοράς του SVM για διάφορες τιμές της υπερπαραμέτρου "*gamma*", επιλέχθηκαν να χρησιμοποιηθούν 10000 δείγματα για το training set και 2000 δείγματα για το test set από το dataset CIFAR-10, λόγω περιορισμένων υπολογιστικών πόρων. Έτσι, για τον **RBF kernel** με $\boldsymbol{C = 10}$ προέκυψαν τα παρακάτω:

Gamma	Χρόνος εκπαίδευσης	Training Accuracy	Testing Accuracy	Support Vectors
0.001	3134.79 sec	83.41%	43.40%	300-350
0.01	3499.20 sec	99.99%	44.70%	600-700
0.1	3607.59 sec	100%	36.20%	1000
1	3979.44 sec	100%	11.61%	1000

Αναλυτικότερα:

- Το μικρότερο "**gamma"** [**0.001**] δημιουργεί έναν πιο ομαλό και 'ευρύ' decision boundary (επιφάνεια απόφασης) που μπορεί να γενικεύσει καλύτερα σε σύγκριση με τις υπόλοιπες τιμές της υπερπαραμέτρου,με το testing accuracy του να μην είναι το καλύτερο δυνατό.
- Η αύξηση του "**gamma"** σε [**0.01**] προσφέρει καλύτερη προσαρμογή στα δεδομένα εκπαίδευσης (σχεδόν τέλεια), αλλά σε συνδυασμό με το μικρό testing accuracy υποδεικνύει overfitting.
- Η ακόμη υψηλότερη τιμή του "**gamma"** [**0.1**] οδηγεί σε τέλεια προσαρμογή στα δεδομένα εκπαίδευσης, αλλά με το testing accuracy να παρουσιάζει μεγάλη μείωση σε σχέση με τα προηγούμενα. Επομένως, υπάρχει σοβαρό overfitting, με χαμηλή απόδοση σε νέα δεδομένα.
- Το πιο υψηλό "**gamma"** [**1**] κάνει το μοντέλο σχεδόν να απομνημονεύει τα δεδομένα εκπαίδευσης. Αυτό έχει ως αποτέλεσμα τη δραματική μείωση του testing accuracy που σε συνδυασμό με την τέλεια προσαρμογή στα δεδομένα

οδηγεί σε ακραίο overfitting και με την ταξινόμηση νέων δεδομένων να φτάνει σε βαθμό να είναι τυχαία (10%).

- Όσον αφορά το πλήθος των **support vectors,** όσο λιγότερα είναι φαίνεται να ευνοούν την γενίκευση. Η αύξηση τους υποδεικνύει ότι το μοντέλο γίνεται πιο περίπλοκο και προκαλεί overfitting, με το πλήθος τους να φτάνει το μέγιστο όριο του μοντέλου (1000 λόγω batch size) από την τιμή [0.1] για το "*gamma*" και μετά, υποδεικνύοντας ότι έχει υπερπροσαρμοστεί πλήρως στα δεδομένα.
- Όσον αφορά τον χρόνο εκπαίδευσης, συνδέεται άμεσα με το πλήθος των support vectors, καθώς όσο περισσότερα είναι αυτά, τόσο πιο υπολογιστικά απαιτητική είναι η διαδικασία και αυτό είναι φανερό στον παραπάνω πίνακα.

**Δοκιμάστηκε και η τιμή [0.0001] για το "gamma", αλλά εμφανίστηκε σφάλμα λόγω της 'κακής κατάστασης' του πίνακα πυρήνα "kernel matrix", κατά τον έλεγχο, μέσω της βιβλιοθήκης "cvxpy", για το αν είναι θετικά ημιορισμένος.*

III. Υπερπαράμετρος "***C***" για τον έλεγχο των σφαλμάτων του SVM

Για διευκόλυνση του ελέγχου της συμπεριφοράς του SVM για διάφορες τιμές της υπερπαραμέτρου "*C*", επιλέχθηκαν να χρησιμοποιηθούν 10000 δείγματα για το training set και 2000 δείγματα για το test set από το dataset CIFAR-10, λόγω περιορισμένων υπολογιστικών πόρων. Έτσι, πραγματοποιήθηκαν όρισμένα πειράματα για τα τρία είδη πυρήνων:

a. **Linear Kernel**

C	**Χρόνος εκπαίδευσης**	**Training Accuracy**	**Testing Accuracy**
0.001	1125.16 sec	40.44%	35.95%
0.01	1112.39 sec	49.03%	38.95%
0.1	1165.13 sec	50.89%	39.00%
1	1336.36 sec	49.83%	37.15%

Αναλυτικότερα:

- Ο χρόνος εκπαίδευσης αυξάνεται ελαφρώς για μεγαλύτερες τιμές του C. Ο linear kernel είναι αριθμητικά λιγότερο απαιτητικός από τους άλλους kernels, καθώς η βελτιστοποίηση αφορά μια γραμμική συνάρτηση. Η μικρή αύξηση για μεγαλύτερο C μπορεί να οφείλεται σε πιο αυστηρές απαιτήσεις για τέλειο διαχωρισμό.

- Για μικρές τιμές όπως $C = 0.001$ το μοντέλο είναι λιγότερο ευαίσθητο σε λάθη, προτιμώντας έναν ευρύτερο margin και αυτό περιορίζει την ικανότητά του να προσαρμόζεται στα δεδομένα.
- Με αύξηση των τιμών του C (0.01 ή 0.1) το μοντέλο βελτιώνει την προσαρμογή του, αυξάνοντας το training accuracy.
- Με μεγάλες τιμές όπως $C = 1$, το μοντέλο μπορεί να αρχίσει να προσαρμόζεται υπερβολικά σε θορυβώδη δεδομένα, με αποτέλεσμα μικρές διακυμάνσεις.

Συνοψίζοντας, συμπεραίνουμε ότι:

- Ο linear kernel έχει περιορισμένη ικανότητα να διαχωρίζει δεδομένα με πολύπλοκα χαρακτηριστικά, όπως αυτά του CIFAR-10. Αν και παρατηρείται μικρή βελτίωση με την αύξηση του C, η γενική απόδοση (testing accuracy) παραμένει χαμηλή, δείχνοντας ότι ο kernel δεν μπορεί να συλλάβει τη μη γραμμική φύση του dataset.

b. **Polynomial Kernel** (2ου βαθμού)

C	Χρόνος εκπαίδευσης	Training Accuracy	Testing Accuracy
0.001	1835.89 sec	96.12%	40.60%
0.01	2046.53 sec	99.73%	44.30%
0.1	1764.89 sec	100%	43.55%
1	1700.81 sec	100%	42.65%

Από τα παραπάνω, συμπεραίνουμε ότι:

- Για $C \geq 0.01$, το training accuracy φτάνει στο 100%, υποδεικνύοντας ότι το μοντέλο προσαρμόζεται τέλεια στα δεδομένα εκπαίδευσης, με $C = 0.001$, το μοντέλο να μην φτάνει την απόλυτη ακρίβεια (96.12%).
- Το testing accuracy είναι σχετικά χαμηλό σε όλες τις περιπτώσεις, καθιστώντας σχετικά προβληματική την γενίκευση. Επιπρόσθετα, σε συνδυασμό με το πολύ υψηλό training accuracy, φαίνεται ότι το μοντέλο οδηγείται σε overfitting.
- Ο χρόνος εκπαίδευσης αν και δεν εμφανίζει σημαντικές διακυμάνσεις, μειώνεται για μεγαλύτερες τιμές του "*C*", λόγω ταχύτερης σύγκλισης του αλγορίθμου, καθώς μειώνεται το πλήθος των υποψήφιων support vectors (το margin είναι μικρότερο).

c. **RBF Kernel** ($\gamma = 0.01$)

C	Χρόνος εκπαίδευσης	Training Accuracy	Testing Accuracy
0.1	2259.06 sec	70.18%	35.65%
1	2046.53 sec	97.67%	41.75%
10	3134.79 sec	99.08%	45.40%
100	3875.59 sec	100%	46.55%
1000	4083.69 sec	100%	45.95%

Από τα παραπάνω, συμπεραίνουμε ότι:

- Ο χρόνος εκπαίδευσης αυξάνεται με την αύξηση του C, καθώς μεγαλύτερο C απαιτεί αυστηρότερη βελτιστοποίηση, δίνοντας μεγαλύτερη έμφαση στα σφάλματα ταξινόμησης.
- Το training accuracy αυξάνεται από 70.18% (C=0.1) σε 100% (C=1000), υποδεικνύοντας ότι το μοντέλο προσαρμόζεται πλήρως στα δεδομένα εκπαίδευσης για υψηλές τιμές C.
- Το testing accuracy βελτιώνεται αρχικά, φτάνοντας το μέγιστο (46.55%) για $C = 100$, αλλά στη συνέχεια μειώνεται ελαφρώς ($C = 1000$), υποδηλώνοντας πιθανό overfitting λόγω της μεγάλης του διαφοράς με το training accuracy.

➢ Με βάση τα αποτελέσματα του ***3.3.II*** και ***3.3.c*** για τον **RBF Kernel**, ο καλύτερος συνδυασμός των υπερπαραμέτρων φαίνεται να είναι με "***C***"=**10** και "***gamma***"=**0.01**. Με αυτές τις υπερπαραμέτρους εκτελέσαμε το πείραμα για ολόκληρο το dataset CIFAR-10 και καταλήξαμε στο εξής:

Χρόνος εκπαίδευσης	Training Accuracy	Testing Accuracy
53722.41 sec	98.10%	52.09%

4. Σύγκριση του SVM με άλλους κατηγοριοποιητές

Σε αυτήν την ενότητα, θα συγκριθεί η απόδοση του **SVM** της παρούσας εργασίας σε σχέση με την κατηγοριοποίηση 1 και 3 πλησιέστερου γείτονα (**Nearest Neighbor**) και πλησιέστερου κέντρου κλάσης (**Nearest Class Centroid**) καθώς επίσης και με ένα **MLP** με ένα κρυφό επίπεδο που θα χρησιμοποιεί **Hinge loss** για την βελτιστοποίηση. Για την αξιολόγηση των

κατηγοριοποιητών πέρα του SVM, θα χρησιμοποιηθούν τα αποτελέσματα από την ενδιάμεση εργασία και από την 1η υποχρεωτική εργασία. Αναλυτικότερα:

- 1-NN, 3-NN, NCC από την ενδιάμεση εργασία με την προσωπική υλοποίηση
- MLP από την 1η υποχρεωτική εργασία με μερικές τροποποιήσεις στο "*backward pass"* και στο "*train",* ώστε να αντικατασταθεί η Cross Entropy Loss από την Hinge Loss (+ απουσία SoftMax στο Outer Layer). Τα χαρακτηριστικά του δικτύου ήταν:
    - Χρήση της συνάρτησης ενεργοποίησης ReLU
    - Δυναμική προσαρμογή του learning rate με αρχικό learning rate = 0.01, lr_decay =0.7 και wait = 1
    - Μοναδικό κρυφό επίπεδο με 512 νευρώνες
    - 250 εποχές
- **RBF Kernel** με "***C***" = **10** και "***gamma***" = **0.01**, όντας ο καλύτερος συνδυασμός υπερπαραμέτρων και kernel

Κατηγοριοποιητής	**Χρόνος Εκπαίδευσης**	**Training Accuracy**	**Testing Accuracy**
1-NN	~9566 sec	100%	35%
3-NN	~9695 sec	~95%	33%
NCC	~4.8 sec	~50%	28%
MLP (Hinge Loss)	5911.75 sec	74.45%	54.36%
SVM (RBF Kernel)	53722.41 sec	98.10%	52.09%

Από τα παραπάνω συμπεραίνουμε ότι:

- Το **SVM** έχει τον μεγαλύτερο **χρόνο εκπαίδευσης**, κυρίως λόγω του μεγάλου αριθμού δεδομένων και της υπολογιστικής πολυπλοκότητας του RBF kernel. Ο **MLP** είναι ταχύτερος από το SVM, αλλά απαιτεί περισσότερο χρόνο από τους Nearest Neighbor αλγόριθμους, λόγω της εκπαίδευσης του νευρωνικού δικτύου. Ο **NCC** είναι εξαιρετικά γρήγορος, καθώς απαιτεί μόνο τον υπολογισμό των κεντροειδών των κλάσεων.
- Ο **1-NN** έχει **training accuracy 100%**, καθώς αποθηκεύει τις πληροφορίες των δεδομένων εκπαίδευσης και αναγνωρίζει τέλεια τα ίδια δείγματα. Ο **3-NN** και το **SVM** προσαρμόζονται επίσης σχεδόν τέλεια στα δείγματα. Το **MLP** έχει καλή επίδοση στην εκπαίδευση, χωρίς να φτάνει αυτήν των προηγούμενων λόγω των χαρακτηριστικών του δικτύου (πχ. Δυναμικό Learning Rate). Ο **NCC** έχει το χαμηλότερο training accuracy, λόγω της απλοποιημένης προσέγγισής του.
- Το **MLP** έχει το καλύτερο **testing accuracy** (**54.36%**), ξεπερνώντας ελαφρώς το SVM και όλους τους άλλους αλγόριθμους, κάτι που υποδεικνύει την ισχυρή γενίκευση του μοντέλου. Οι **1-NN** και **3-NN** αποδίδουν πολύ χαμηλά στο test set (~35% και ~33% αντίστοιχα), παρόλο που απομνημονεύουν τα δεδομένα εκπαίδευσης, λόγω της ευαισθησίας τους στο θόρυβο. Ο **NCC** να έχει το χαμηλότερο testing accuracy (28%), γεγονός που δείχνει την αδυναμία του να αποτυπώσει πολύπλοκα σύνορα απόφασης.

*Τελικά, το **MLP με Hinge Loss** υπερτερεί σε γενίκευση, ενώ παράλληλα έχει αποδεκτό χρόνο εκπαίδευσης. Ο **SVM** αποδίδει επίσης καλά, αλλά με τεράστιο υπολογιστικό κόστος.Οι **NN** έχουν χαμηλή γενίκευση, καθιστώντας τους λιγότερο κατάλληλους για σύνθετα δεδομένα με υψηλή διαστατικότητ, όπως το CIFAR-10.*