Δημήτριος Κυριακίδης el17077

Χαράλαμπος Δάλπης el17067

Τεχνητή Νοημοσύνη: Εργαστηριακή Άσκηση 3

Στην άσκηση αυτή θα υλοποιηθούν διάφοροι (σχετικά απλοί) αλγόριθμοι μηχανικής μάθησης για την αυτόματη αναγνώριση μεταξύ 3 μουσικών είδών απο τα δεδομένα που προσφέρει η υπηρεσία Spotify. Συγκεκριμένα, θα δίνονται δύο σύνολα δεδομένων

$$\mathrm{Z}_{train} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$$
$$Z_{test} = \{(x_j, y_j), \ldots, (x_k, y_k)\}$$

όπου κάθε $x_i \in \mathbb{R}^p$ είναι ένα διάνυσμα με τα μουσικά χαρακτηριστικά κάθε κομματιού (όπως dancability, acousticness κ.α.) και y_i είναι το είδος του κομματιού - ένας ακέραιος στο $[0, 2]$. Σε κάθε περίπτωση καλείστε να σχεδιάσετε έναν ταξινομητή, δηλαδή μια απεικόνιση

$$f : \mathbb{R}^p \to [0, 2]$$

1ο Μέρος: Αξιολόγηση

Στο πρώτο μέρος της άσκησης θα υλοποιηθούν συναρτήσεις που θα χρησιμοποιηθούν για την αξιολόγηση των ταξινομητών που θα χρησιμοποιηθούν στα επόμενα μέρη.

Παρακάτω σας δίνεται η κλάση Evaluate, η οποία υπολογίζει διάφορες μετρικές με τη μέθοδο get_metrics, εντοπίζει αντικείμενα που ταξινομήθηκαν λάθος και τα εμφανίζει (μέθοδος get_sample_of_wrong), και υπολογίζει τον πίνακα σύγχυσης (confusion matrix) όπου οπτικοποιούνται ανά κατηγορία οι προβλέψεις του ταξινομητή.

Για το μέρος αυτό καλείστε να υλοποιήσετε στη μέθοδο my_accuracy τη μετρική accuracy, η οποία ορίζεται ως:

$$accuracy = \frac{\#σωστών\_προβλέψεων}{\#δεδομένων}$$

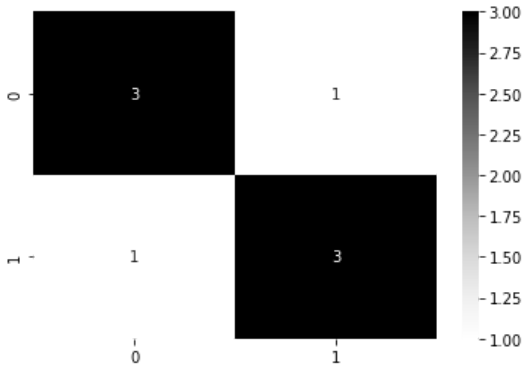
In [ ]:
```
from sklearn.metrics import confusion_matrix, f1_score, precision_score, recall_score
from matplotlib import pyplot as plt
import numpy as np
import random
import seaborn as sns

class Evaluate:
    def __init__(self, y_true, y_pred):
        self.y_true = y_true
        self.y_pred = y_pred

    def my_accuracy(self):
        ##################
        ## Your code below
        correctPredictions = 0
        for i, j in zip(self.y_true, self.y_pred):
            if(i == j): correctPredictions += 1

        acc = correctPredictions / len(self.y_true)
        ## Your code above
        ##################
        return acc

    def get_metrics(self):
        precision = precision_score(self.y_true, self.y_pred, average = "macro")
        recall = recall_score(self.y_true, self.y_pred, average = "macro")
        f1 = f1_score(self.y_true, self.y_pred, average = "macro")
        results = {"precision": precision, "recall": recall, "f1": f1, "accuracy": self.my_accuracy()}
        return results

    def confusion_matrix(self):
        cnfm = confusion_matrix(self.y_true, self.y_pred)
        sns.heatmap(cnfm, annot = True, cmap = "Greys", fmt = 'd')

    def get_evaluation_report(self):
```

```
        metrics = self.get_metrics()
        for m in metrics:
            print(m + ': ' + str(metrics[m]))
        print("Confusion matrix: ")
        self.confusion_matrix()
```

Παράδειγμα χρήσης της κλάσης. Κανονικά στο x θα υπάρχουν τα δεδομένα από το dataset

In [ ]:
```
y_true = [1, 0, 1, 0, 0, 1, 1, 0]
y_pred = [1, 0, 1, 0, 1, 1, 0, 0]

eval = Evaluate(y_true, y_pred)
eval.get_evaluation_report()
```

```
precision: 0.75
recall: 0.75
f1: 0.75
accuracy: 0.75
Confusion matrix:
```



Dataset

Το dataset που σας δίνεται περιέχει πληθώρα μουσικών κομματιών για τα οποία έχουν καταγραφεί διάφορα χαρακτηριστικά τους, όπως επίσης και το μουσικό είδος στο οποίο ανήκουν. Στη συγκεκριμένη άσκηση θα δουλέψουμε με ένα υποσύνολο (αριθμητικών) χαρακτηριστικών, τα οποία συνεισφέρουν στον καθορισμό της μουσικής κατηγορίας κάθε κομματιού.

Τα χαρακτηριστικά τα οποία θα μελετήσουμε στην παρούσα άσκηση είναι τα "acousticness", "danceability", "energy", "instrumentalness", "liveness", "speechiness", ενώ οι κατηγορίες στις οποίες καλούμαστε να ταξινομήσουμε τα μουσικά κομμάτια είναι οι "Electronic", "Rock", και "Rap".

In [ ]:
```
# Σύνδεση του Google Colab με το Google Drive

from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Θα χρησιμοποιήσουμε τα DataFrames της βιβλιοθήκης pandas για να χειριστούμε τα δεδομένα μας. Μπορείτε να βρείτε περισσότερες πληροφορίες για τα pandas DataFrames στο αντίστοιχο documentation.

In [ ]:
```
import pandas as pd
from tqdm.notebook import tqdm
```

In [ ]:
```
# read data in the form of pandas DataFrame
data = pd.read_csv("/content/drive/My Drive/Colab Notebooks/AI/music_df_processed.csv")

# print the first 5 values of the DataFrame using .head() command
data.head()
```

Out[ ]:

	instance_id	artist_name	track_name	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	key	liveness	loudnes
0	32894.0	Röyksopp	Röyksopp's Night Out	27.0	0.00468	0.652	-1.0	0.941	0.79200	A#	0.115	-5.20
1	46652.0	Thievery Corporation	The Shining Path	31.0	0.01270	0.622	218293.0	0.890	0.95000	D	0.124	-7.04
2	30097.0	Dillon Francis	Hurricane	28.0	0.00306	0.620	215613.0	0.755	0.01180	G#	0.534	-4.61
3	62177.0	Dubloadz	Nitro	34.0	0.02540	0.774	166875.0	0.700	0.00253	C#	0.157	-4.49
4	24907.0	What So Not	Divide & Conquer	32.0	0.00465	0.638	222369.0	0.587	0.90900	F#	0.157	-6.26

In [ ]:
```
# What can we see here?
data.describe()
```

Out[ ]:

	instance_id	popularity	acousticness	danceability	duration_ms	energy	instrumentalness	liveness	loudness
count	45020.000000	45020.000000	45020.000000	45020.000000	4.502000e+04	45020.000000	45020.000000	45020.000000	45020.000000
mean	55884.691892	44.263327	0.306596	0.558532	2.211695e+05	0.599553	0.181843	0.193951	-9.137016
std	20728.197040	15.553972	0.341391	0.178858	1.276884e+05	0.264510	0.325847	0.161715	6.156400
min	20002.000000	0.000000	0.000000	0.059600	-1.000000e+00	0.000792	0.000000	0.009670	-47.046000
25%	38000.750000	34.000000	0.020100	0.442000	1.747230e+05	0.432000	0.000000	0.096900	-10.860000
50%	55857.500000	45.000000	0.145000	0.568000	2.194385e+05	0.642000	0.000159	0.126000	-7.284000
75%	73856.500000	56.000000	0.551000	0.687000	2.686400e+05	0.815250	0.154000	0.244000	-5.177000
max	91759.000000	99.000000	0.996000	0.986000	4.497994e+06	0.999000	0.996000	1.000000	3.744000

Επιλογή χαρακτηριστικών x και στόχων y.

Για λόγους απλότητας επιλέγουμε τα χαρακτηριστικά (inputs) και τις κατηγορίες-στόχους (genres). Καλείστε να διαχωρίσετε τα δεδομένα σε train/test set. Ας θεωρήσουμε το διαχωρισμό 30% - test set, 70% - train set.

In [ ]:
```
from pandas.core.common import random_state
from sklearn.model_selection import train_test_split

# χαρακτηριστικά
featureNames = ["acousticness", "danceability", "energy", "instrumentalness", "liveness", "speechiness"]

# κατηγορίες-στόχοι
output = "music_genre"
genres = ["Electronic", "Rock", "Rap"]

# φιλτράρουμε το DataFrame ώστε να διατηρήσουμε μόνο τις 3 κατηγορίες που μας ενδιαφέρουν.
## only keep relevant rows
data = data.loc[data[output].isin(genres), :]

print(data.head())

## only keep relevant columns
features = data.loc[:, data.columns.isin(featureNames)]
labels = data.loc[:, output]

print(features.head())
print(labels.head())
```

```
   instance_id           artist_name  ... valence  music_genre
0      32894.0             Röyksopp  ...   0.759   Electronic
1      46652.0  Thievery Corporation  ...   0.531   Electronic
2      30097.0        Dillon Francis  ...   0.333   Electronic
3      62177.0              Dubloadz  ...   0.270   Electronic
4      24907.0           What So Not  ...   0.323   Electronic

[5 rows x 18 columns]
   acousticness  danceability  energy  instrumentalness  liveness  speechiness
0       0.00468         0.652   0.941           0.79200     0.115       0.0748
1       0.01270         0.622   0.890           0.95000     0.124       0.0300
2       0.00306         0.620   0.755           0.01180     0.534       0.0345
3       0.02540         0.774   0.700           0.00253     0.157       0.2390
4       0.00465         0.638   0.587           0.90900     0.157       0.0413
0    Electronic
1    Electronic
2    Electronic
3    Electronic
4    Electronic
Name: music_genre, dtype: object
```

In [ ]:
```python
# dictionary to map genre to label id
genres_to_id = {genre: i for i, genre in enumerate(genres)}
encodedLabels = np.asarray([genres_to_id[i] for i in labels.values])
# εδώ πρέπει να διαχωρίσετε τα δεδομένα σε train (70% των δεδομένων)/test set (30% των δεδομένων)
# ονομάστε τις μεταβλητές ως εξής:
# τα χαρακτηριστικά του train set: x_train
# τις κατηγορίες-στόχους του train set: y_train
# τα χαρακτηριστικά του test set: x_test
# τις κατηγορίες-στόχους του test set: y_test

##################
## Your code below
x_train, x_test, y_train, y_test = train_test_split(features.values, encodedLabels, test_size = 0.3, random_state = 31)
## Your code above
##################
```

Μορφή των δεδομένων

Βεβαιωθείτε ότι τα δεδομένα σας έχουν τη σωστή μορφή εκτυπώνοντας τον αριθμό γραμμών και στηλών για τα x_test, y_test, x_train, y_train.

In [ ]:
```python
# Shape of x_test, y_test, x_train, y_train

##################
## Your code below
print(f"x_train: {x_train.shape}")
print(f"y_train: {y_train.shape}")
print(f"x_test: {x_test.shape}")
print(f"y_test: {y_test.shape}")
## Your code above
##################
```

```
x_train: (9471, 6)
y_train: (9471,)
x_test: (4060, 6)
y_test: (4060,)
```

Αναφορικά με τις τιμές των χαρακτηριστικών, είναι σημαντικό να γνωρίζουμε το εύρος τους, δηλαδή τη μέγιστη και την ελάχιστη τιμή που λαμβάνει το κάθε χαρακτηριστικό. Εξερευνήστε το εύρος του κάθε χαρακτηριστικού στα train και test set.

In [ ]:
```python
# Range of x_train, x_test columns

##################
## Your code below
print("Train set ranges:")
for i, f in enumerate(featureNames):
    print(f"{f}: [{x_train[:, i].min()}, {x_train[:, i].max()}]")

print("\nTest set ranges:")
for i, f in enumerate(featureNames):
    print(f"{f}: [{x_test[:, i].min()}, {x_test[:, i].max()}]")
## Your code above
##################
```

```
Train set ranges:
acousticness: [1.39e-06, 0.9940000000000001]
danceability: [0.0822, 0.977]
energy: [0.0231, 0.9990000000000001]
instrumentalness: [0.0, 0.986]
liveness: [0.0194, 0.991]
speechiness: [0.0228, 0.922]

Test set ranges:
acousticness: [1.6e-06, 0.9940000000000001]
danceability: [0.0645, 0.973]
energy: [0.00259, 0.998]
instrumentalness: [0.0, 0.972]
liveness: [0.0173, 0.963]
speechiness: [0.0224, 0.863]
```

In [ ]:
```
labels.value_counts()
```

Out[ ]:
```
Rock          4561
Rap           4504
Electronic    4466
Name: music_genre, dtype: int64
```

Από την παραπάνω ανάλυση προκύπτουν κάποια ερωτήματα σημαντικά για τα επόμενα βήματα:

- Έχουν τα χαρακτηριστικά μας περίπου το ίδιο εύρος;
- Σε πολλές εφαρμογές είναι σημαντικό τα χαρακτηριστικά να βρίσκονται στο εύρος [0, 1]. Ισχύει αυτό στην περίπτωσή μας;

Και τα έξι χαρακτηριστικά φαίνεται πως έχουν περίπου το ίδιο εύρος [0, 1]. Επίσης, οι τρεις κλάσεις είναι ομοιόμορφα κατανεμημένες ως προς το πλήθος δειγμάτων.

2ο Μέρος: Υλοποίηση KNN

Στο δεύτερο μέρος της άσκησης θα υλοποιήσετε τον αλγόριθμο KNN για ταξινόμηση. Υπενθυμίζεται από τις διαφάνειες το πλάνο σχεδιασμού για τον ταξινομητή k κοντινότερων γειτόνων:

- Αποθηκεύουμε όλα τα δεδομένα (Z_{train}) στη μνήμη
  - Τα δεδομένα μπορούν αποθηκευτούν σε έναν πίνακα $n \times p$ με χρήση του numpy
- Συγκρίνουμε την είσοδο με τα δεδομένα και βρίσκουμε τα k κοντινότερα ($k < n$) με βάση κάποια απόσταση.
  - Όταν μας δίνεται ένα "φρέσκο" δείγμα ως διάνυσμα από χαρακτηριστικά x_i χρειαζόμαστε μια συνάρτηση που να υπολογίζει την απόσταση $d(x_i, x_j)$, όπου x_j είναι το διάνυσμα που αντιστοιχεί στα χαρακτηριστικά ενός δείγματος από τα δεδομένα εκπαίδευσης. Θα πειραματιστείτε με την ευκλείδια απόσταση και την απόσταση συνημιτόνου. Στη συνέχεια ταξινομούνται τα δεδομένα εκπαίδευσης ως προς την απόστασή τους από το x_i και επιλέγονται τα k κοντινότερα
- Δίνουμε στην έξοδο την κλάση στην οποία ανήκει η πλειοψηφία των k κοντινότερων δεδομένων.

Αφού κατασκευαστεί ο ταξινομητής θα αξιολογήσετε την επίδοσή του στα 100 πρώτα δείγματα του Z_{test} για κάποιες τιμές του k που θα επιλέξετε εσείς, ξεκινώντας από $k = 1$.

Στην πράξη πολύ σπάνια θα χρειαστεί να υλοποιήσετε έναν αλγόριθμο μηχανικής μάθησης από το μηδέν, αφού υπάρχουν έτοιμες υλοποιήσεις, π.χ. σε πακέτα της python, οι οποίες είναι βελτιστοποιημένες και εύχρηστες. Το τελευταίο ζητούμενο στο 2ο μέρος είναι να επαναλάβετε το παραπάνω πείραμα με την έτοιμη υλοποίηση του KNN που παρέχει η βιβλιοθήκη sklearn. Καλείστε να συγκρίνετε τα αποτελέσματα και τους χρόνους εκτέλεσης.

Σας δίνεται η κλάση KNN η οποία αρχικοποιείται με ένα σύνολο από δεδομένα x, ετικέτες y και το k για τον αλγόριθμο. Καλείστε να συμπληρώσετε τον κώδικα που λείπει στις μεθόδους distance, get_knn, και classify.

Η απόσταση συνημιτόνου μεταξύ δύο διανυσμάτων u,v ορίζεται ως:

$$d(u, v) = 1 - \frac{u \cdot v}{||u||_2 ||v||_2}$$

In [ ]:
```
import numpy as np
from collections import Counter

class KNN:
    def __init__(self, x, y, k, distance = "euclidean"):
        self.x = x
        self.y = y
```

```
        self.k = k
        self.distance = distance

    ## Compute the distance between the two vectors (2 rows of the DataFrame)
    # hint: use np.linalg.norm for euclidean
    # hint: use equation given above for cosine
    def get_distance(self, row1, row2):
        if(self.distance == "euclidean"):
            ##################
            ## Your code below
            dist = np.linalg.norm(row1 - row2)
            ## Your code above
            ##################
        elif(self.distance == "cosine"):
            ##################
            ## Your code below
            dist = np.dot(row1, row1) / (np.linalg.norm(row1) * np.linalg.norm(row2))
            ## scipy.spatial.distance.cosine
            ## Your code above
            ##################

        return dist

    ## Given a DataFrame row as a vector, returns indices of k nearest neighbors
    def get_knn(self, row):
        distancesList = list()

        ##################
        ## Your code below - populate the distances list
        # hint: you can use a for loop

        for e in self.x: distancesList.append(self.get_distance(e, row))

        ## Your code above
        ##################

        # Sort distances, and return the indexes of k first elements
        ans_indices = np.argsort(distancesList)[:self.k]
        return ans_indices

    ## Given a DataFrame row as a vector, classify it according to KNN
    # hint: we have a list of k labels and want to return the most common one
    def classify(self, row):
        nn_labels = [self.y[i] for i in self.get_knn(row)]

        ##################
        ## Your code below
        '''
         most_common([n])

         Returns a list of the n most common elements and their counts (in tuple form) from the most common to the least
         If n is omitted or None, most_common() returns all elements in the counter.
         Elements with equal counts are ordered in the order first encountered.
        '''
        prediction = Counter(nn_labels).most_common(1)[0][0]
        ## Your code above
        ##################
        return prediction


knn = KNN(x_train, y_train, k = 5, distance = "euclidean")
```

Τώρα που είναι έτοιμος ο ταξινομητής ας δούμε τι προβλέπει σε μεμονωμένα δείγματα.

Αξιολόγηση του KNN

In [ ]:
```
x_test[0:101]
```

Out[ ]:

	acousticness	danceability	energy	instrumentalness	liveness	speechiness
26818	0.209000	0.878	0.706	0.000571	0.0762	0.2150
4077	0.005910	0.656	0.558	0.634000	0.0422	0.0450
33596	0.122000	0.332	0.339	0.000001	0.0985	0.0355
4268	0.000179	0.686	0.980	0.761000	0.1970	0.1840
23905	0.004130	0.897	0.634	0.000000	0.0686	0.1610
...	...	...	...	...	...	...
23321	0.012400	0.585	0.650	0.000000	0.1120	0.0657
26409	0.495000	0.775	0.573	0.000000	0.1260	0.0585
26841	0.093300	0.813	0.453	0.000000	0.1400	0.3380
24206	0.000941	0.434	0.960	0.028600	0.0437	0.0462
31598	0.172000	0.666	0.713	0.032000	0.1770	0.0384

101 rows × 6 columns

In [ ]:
```
x_test[0].tolist()
```

Out[ ]:
```
[0.209, 0.878, 0.706, 0.000571, 0.0762, 0.215]
```

In [ ]:
```
np.linalg.norm(x_train[0] - x_test[0], ord = 2)
```

Out[ ]:
```
0.4973241422261742
```

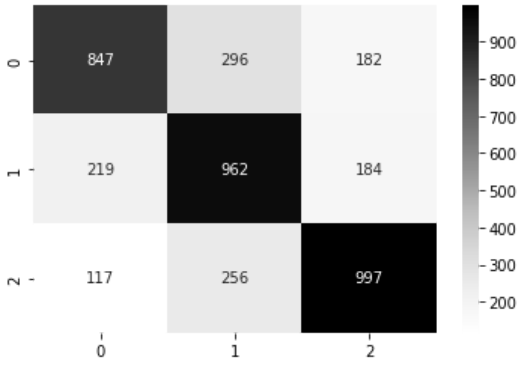
In [ ]:
```
for _, r in x_test.iterrows(): print(r.values)
```

In [ ]:
```
%%time
preds = [knn.classify(r) for r in x_test]
## real = [y_test[i] for i in range(N)]
```

```
CPU times: user 5min 8s, sys: 17.8 s, total: 5min 26s
Wall time: 5min 12s
```

In [ ]:
```
eval = Evaluate(y_test, preds)
eval.get_evaluation_report()
```

```
precision: 0.6942846419191621
recall: 0.690581471352715
f1: 0.6911091224089293
accuracy: 0.6911330049261084
Confusion matrix:
```



Έτοιμος KNN classifier

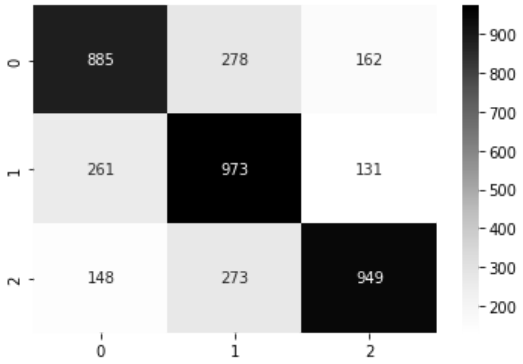
Όπως και με τους περισσότερους αλγορίθμους μηχανικής μάθησης, υπάρχουν έτοιμες βελτιστοποιημένες υλοποιήσεις. Παρακάτω δείχνουμε ένα παράδειγμα χρήσης του ταξινομητή KNN που παρέχει η βιβλιοθήκη sklearn (documentation).

In [ ]:
```
%%time
from sklearn.neighbors import KNeighborsClassifier

knc = KNeighborsClassifier(n_neighbors = 5, p = 2)
knc.fit(x_train, y_train)
y_pred = knc.predict(x_test)
```

```
CPU times: user 357 ms, sys: 0 ns, total: 357 ms
Wall time: 794 ms
```

In [ ]:
```
eval = Evaluate(y_test, y_pred)
eval.get_evaluation_report()
```

```
precision: 0.6954891440468708
recall: 0.6911485903498024
f1: 0.6920220650194852
accuracy: 0.6913793103448276
Confusion matrix:
```



Σύγκριση υλοποιήσεων

Στα παρακάτω κελιά πειραματιστείτε με τις δύο υλοποιήσεις (τη δική σας και την έτοιμη). Βεβαιωθείτε πως προκύπτουν τα ίδια αποτελέσματα για διάφορες τιμές του k (για ευκλείδια απόσταση) και μετρήστε τους χρόνους εκτέλεσης.

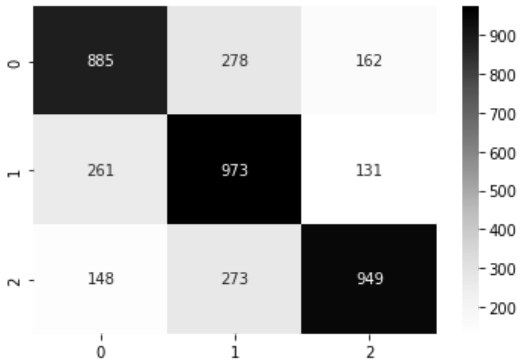
Για τους χρόνους εκτέλεσης για k = 5 τρέχουμε τα 3 παρακάτω κελιά:

In [ ]:
```
%%time
knn = KNN(x_train, y_train, k = 5, distance = "euclidean")
preds = [knn.classify(i) for i in x_test]
```

```
CPU times: user 4min 59s, sys: 19.7 s, total: 5min 18s
Wall time: 4min 59s
```

In [ ]:
```
eval = Evaluate(y_test, y_pred)
eval.get_evaluation_report()
```

```
precision: 0.6954891440468708
recall: 0.6911485903498024
f1: 0.6920220650194852
accuracy: 0.6913793103448276
Confusion matrix:
```

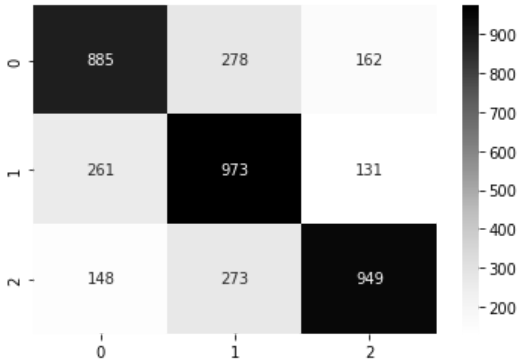


In [ ]:
```
%%time
knn = KNN(x_train, y_train, k = 5, distance = "cosine")
preds = [knn.classify(i) for i in x_test]
```

```
CPU times: user 8min 56s, sys: 11.5 s, total: 9min 8s
Wall time: 8min 59s
```

In [ ]:
```
eval = Evaluate(y_test, y_pred)
eval.get_evaluation_report()
```

```
precision: 0.6954891440468708
recall: 0.6911485903498024
f1: 0.6920220650194852
accuracy: 0.6913793103448276
Confusion matrix:
```



In [ ]:
```
%%time
knc = KNeighborsClassifier(n_neighbors = 5)
knc.fit(x_train, y_train)
y_pred = knc.predict(x_test[:100])
```

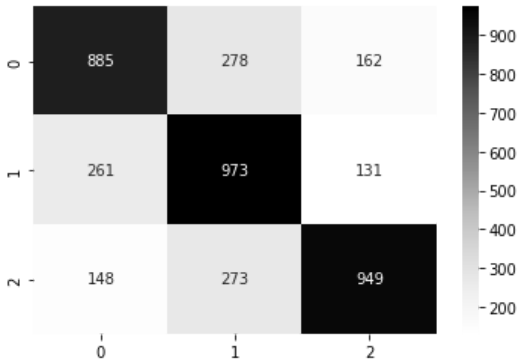
Για τους χρόνους εκτέλεσης για k = 50 τρέχουμε τα 3 παρακάτω κελιά:

In [ ]:
```
%%time
knn = KNN(x_train, y_train, k = 50, distance = "euclidean")
preds = [knn.classify(i) for i in x_test]
```

```
CPU times: user 5min 2s, sys: 19.4 s, total: 5min 22s
Wall time: 5min 1s
```

In [ ]:
```
eval = Evaluate(y_test, y_pred)
eval.get_evaluation_report()
```

```
precision: 0.6954891440468708
recall: 0.6911485903498024
f1: 0.6920220650194852
accuracy: 0.6913793103448276
Confusion matrix:
```

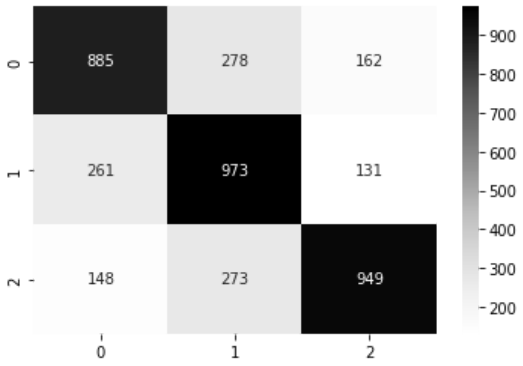


In [ ]:
```
%%time
knn = KNN(x_train, y_train, k = 50, distance = "cosine")
preds = [knn.classify(i) for i in x_test]
```

```
CPU times: user 10min 32s, sys: 10.5 s, total: 10min 43s
Wall time: 11min 10s
```

In [ ]:
```
eval = Evaluate(y_test, y_pred)
eval.get_evaluation_report()
```

```
precision: 0.6954891440468708
recall: 0.6911485903498024
f1: 0.6920220650194852
accuracy: 0.6913793103448276
Confusion matrix:
```

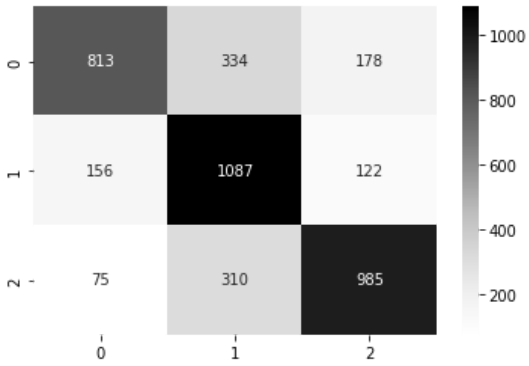


In [ ]:
```
%%time
knc = KNeighborsClassifier(n_neighbors = 50)
knc.fit(x_train, y_train)
y_pred = knc.predict(x_test)
```

```
CPU times: user 537 ms, sys: 8.52 ms, total: 546 ms
Wall time: 566 ms
```

In [ ]:
```
eval = Evaluate(y_test, y_pred)
eval.get_evaluation_report()
```

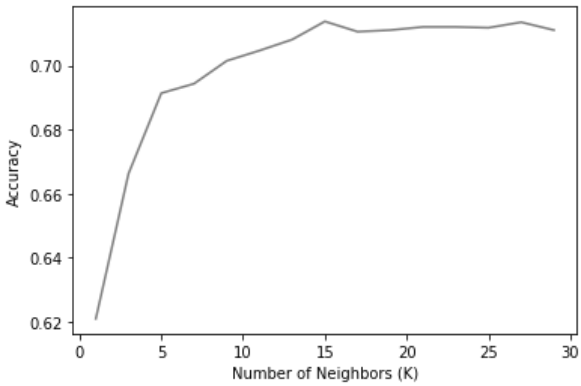
```
precision: 0.7244111045044613
recall: 0.7096333347290514
f1: 0.7101860570123774
accuracy: 0.7105911330049262
Confusion matrix:
```



Βελτιστοποίηση ως προς k

In [ ]:
```
accScores = []

for k in range(1, 30, 2):
    clf = KNeighborsClassifier(n_neighbors = k)
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)
    eval = Evaluate(y_test, y_pred)
    accScores.append(eval.get_metrics().get("accuracy"))
```

In [ ]:
```
import matplotlib.pyplot as plt

plt.plot(list(range(1, 30, 2)), accScores, color = "gray")
plt.xlabel("Number of Neighbors (K)")
plt.ylabel("Accuracy")
plt.show()
```



Σχολιασμός

Για τους ταξινομητές kNN παρατηρούνται τα εξής:

- Γιά όλες τις υλοποιήσεις και τιμές παραμέτρων, η ακρίβεια και οι λοιπές μετρικές κυμαίνονται γύρω στο 70%.
- Από τους πίνακες σύγχυσης φαίνεται πως αρκετά δείγματα της κλάσης 1 ("Rock") ταξινομούνται εσφαλμένα στις δύο άλλες. Αυτό σημαίνει ότι δεν είναι όλα συγκεντρωμένα σε μια συνεκτική περιοχή στον διανυσματικό χώρο.
- Ως προς τις μετρικές ορθότητας, δεν υπάρχει διαφορά στην απόδοση του έτοιμου classifier και του υλοποιημένου από εμάς (γιά ίδιες τιμές k), καθώς πρόκειται για ντετερμινιστική διαδικασία. Όμως, η έτοιμη συνάρτηση είναι σημαντικά ταχύτερη.
- Εξετάζοντας την μεταβολή της απόδοσης συναρτήσει του k εμφανίζεται μικρή μεταβολή, με το μέγιστο στο 15.

3ο Μέρος: Naive Bayes

Στο τρίτο μέρος της άσκησης θα υλοποιήσετε τον αλγόριθμο Naive Bayes. Ας θυμηθούμε από τις διαφάνειες:

Υποθέσεις:

- Τα χαρακτηριστικά είναι boolean αντί για συνεχή, δηλαδή παίρνουν δύο τιμές 0 ή 1. Συνεπώς, χρειάζεται να τροποποιήσουμε τα χαρακτηριστικά του dataset μας.
    - Για το συγκεκριμένο πρόβλημα μπορούμε να 'σπάσουμε' τις τιμές κάθε χαρακτηριστικού σε N διαφορετικά bins. Για παράδειγμα, για ένα χαρακτηριστικό που οι τιμές του κυμαίνονται στο [0, 1], για N=5, θα έχουμε τα ακόλουθα bins: [0, 0.2), [0.2, 0.4), [0.4, 0.6), [0.6, 0.8), [0.8, 1]. (Γι αυτό το λόγο στα προηγούμενα βήματα αναφέραμε ότι είναι σημαντικό να έχουμε τα χαρακτηριστικά μας στο [0, 1]!)
- Η πιθανότητα ένα στοιχείο με χαρακτηριστικά x να ανήκει στην κλάση i δίνεται από τον τύπο:

$$p(i|x) = \frac{p(i) \cdot \prod_{k=1}^{p} p(x^{(k)}|i)}{\sum_{j=1}^{p} p(x^{(k)}|j)}$$

- Για να ταξινομήσουμε ένα διάνυσμα χαρακτηριστικών x σε μια κλάση i επιλέγουμε την κλάση που μεγιστοποιεί την παραπάνω πιθανότητα
    - Μπορούμε για τη σύγκριση να αγνοήσουμε τον παρονομαστή, αφού για όλες τις κλάσεις θα είναι ίδιος

In [ ]:
```
# κάνουμε κάθε μεταβλήτη του συνόλου εκπαίδευσης διακριτή σε σε διαστήματα

def discretize(x, num_of_classes = 5):
    x_r = []
    for row in x:
        discrete = []
        for i, feature in enumerate(row):
            discrete_feature = [0] * num_of_classes
            for j, v in enumerate(np.linspace(0, 1, num_of_classes + 1)):
                if float(feature) < v:
                    break
            discrete_feature[j-1] = 1
            discrete += discrete_feature
        x_r.append(discrete)
    return np.array(x_r)

x_train_r = discretize(x_train)
x_test_r = discretize(x_test)
```

Παρακάτω σας δίνεται η κλάση NaiveBayes που υλοποιεί τον αλγόριθμο. Καλείστε αρχικά να υπολογίσετε την πιθανότητα $p(x^{(k)}|i)$ για διάνυσμα χαρακτηριστκών x και κατηγορία i στη μέθοδο compute_probabilities. Στη συνέχεια θα υπολογίσετε την πιθανότητα $p(i|x)$ στη μέθοδο predict.

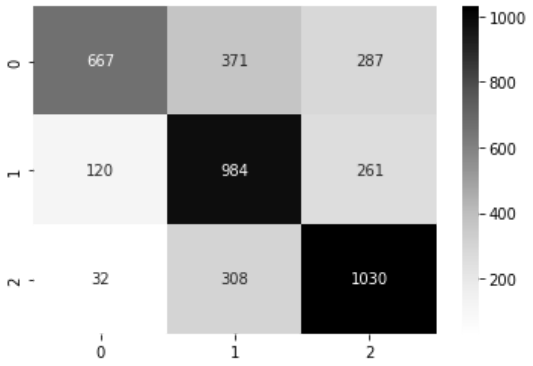
In [ ]:
```
class NaiveBayes:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        ## pC is a vector with the probability of each class
        self.pC = np.zeros((len(genres),))
        ## pxC is an array with all probabilities p(xi|C)
        self.pxC = np.zeros((x.shape[-1], len(genres)))
        ## Compute the probabilities
        self.compute_probabilities()

    def compute_probabilities(self):
        ## Compute p(C) for each class
        for label in self.y: self.pC[label] += 1
        self.pC = self.pC / self.y.shape[0]

        ## Compute p(xi|C) for each feature xi and class C
        # hint: you can use one or more for loops
        ####################
        ## Your code below

        for i in range(self.x.shape[0]):
            for j in range(self.x.shape[-1]):
                self.pxC[j][self.y[i]] += self.x[i][j]

        self.pxC /= self.x.shape[0]

        ## Your code above
        ###################

    def predict(self, x):
        ## ~Probability of x belonging to each class
        ## (not actual probability since we ignore denominator)
        pcX = np.ones((len(genres),))
        xsize = self.x.shape[-1]

        for i in range(len(genres)):
            # hint: We have probabilities p({x_j=1}|i) in self.pxC
            # We also need p({x_j=0}|i) for computing p(x|i)
            ##################
            ## Your code below

            for j in range(30):
                if x[j] == 1:
                    pcX[i] *= self.pxC[j][i]
                else:
                    pcX[i] *= 1 - self.pxC[j][i]

            pcX[i] *= self.pC[i]

            ## Your code above
            ###################
        return np.argmax(pcX)
```

Αξιολόγηση του Naive Bayes

In [ ]:
```
nb = NaiveBayes(x_train_r, y_train)
preds = [nb.predict(i) for i in x_test_r]
```

In [ ]:
```
eval = Evaluate(y_test, preds)
eval.get_evaluation_report()
```

```
precision: 0.6862781755195254
recall: 0.6587000549374878
f1: 0.6569714251487847
accuracy: 0.6603448275862069
Confusion matrix:
```

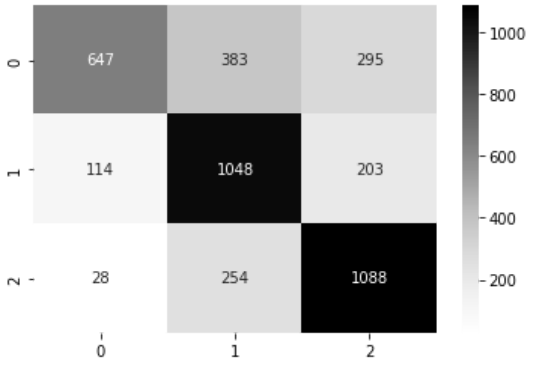


Έτοιμος Naive Bayes

Όπως με τους περισσότερους αλγορίθμους μηχανικής μάθησης, υπάρχουν έτοιμες βελτιστοποιημένες υλοποιήσεις για τον Naive Bayes. Παρακάτω ο Gaussian Naive Bayes από το Sklearn. Σε αντίθεση με τη δική μας υλοποίηση, ο συγκεκριμένος δουλεύει και με συνεχή δεδομένα, αφού πρώτα κάνει την υπόθεση πως κάθε χαρακτηριστικό ακολουθεί κανονική κατανομή (documentation).

In [ ]:
```
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()
y_pred = gnb.fit(x_train, y_train).predict(x_test)

eval = Evaluate(y_test, y_pred)
eval.get_evaluation_report()
```

```
precision: 0.7093287758612828
recall: 0.6834093461665421
f1: 0.6784842548638587
accuracy: 0.6854679802955665
Confusion matrix:
```



Σύγκριση υλοποιήσεων

Όμοια με πριν θα συγκρίνετε τα αποτελέσματα και τους χρόνους εκτέλεσης για τις δύο υλοποιήσεις. Σχολιάστε την επίδοση σε κάθε περίπτωση. Ποιες από τις παραδοχές που κάναμε δεν ισχύουν;

In [ ]:
```
%%time
nb = NaiveBayes(x_train_r,y_train)
preds = [ nb.predict(i) for i in x_test_r]
```

```
CPU times: user 1.06 s, sys: 46.2 ms, total: 1.11 s
Wall time: 1.04 s
```

In [ ]:
```
%%time
gnb = GaussianNB()
y_pred = gnb.fit(x_train, y_train).predict(x_test)
```

```
CPU times: user 4.19 ms, sys: 1.21 ms, total: 5.41 ms
Wall time: 5.38 ms
```

Σχολιασμός

Για τον αλγόριθμο Naive Bayes παρατηρούνται τα εξής:

- Γιά όλες τις υλοποιήσεις και τιμές παραμέτρων, η ακρίβεια και οι λοιπές μετρικές κυμαίνονται γύρω στο 65%.

- Τόσο ο δικός μας ταξινομητής με την κβαντοποίηση σε διαστήματα, όσο και ο έτοιμος, που υποθέτει κανονική κατανομή των δειγμάτων, ταξινομούν πολλά στοιχεία της κλάσης 1 ("Rock") στις άλλες 2.
- Ως προς τις μετρικές ορθότητας, δεν υπάρχει σημαντική διαφορά στην απόδοση του έτοιμου classifier και του υλοποιημένου από εμάς. Όμως, η έτοιμη συνάρτηση είναι σημαντικά ταχύτερη.

4ο Μέρος: Multi-Layer Perceptron

Στο τέταρτο μέρος της άσκησης θα κατασκευάσετε ένα πολυεπίπεδο νευρωνικό δίκτυο. Ο ταξινομητής αυτός θα εκπαιδευτεί στο να ταξινομεί τα δείγματα των μουσικών κομματιών σε μια από τις 3 διαφορετικές κλάσεις που επιλέχθηκαν (Electronic, Rock, Rap). Αρχικά, θα υλοποιήσετε το μοντέλο αυτό χρησιμοποιώντας αποκλειστικά την βιβλιοθήκη numpy ενώ στην συνέχεια θα χρησιμοποιήσετε μια έτοιμη κλάση για την κατασκευή του ίδιου μοντέλου.

Ας θυμηθούμε από τις διαφάνειες:

Κάθε MLP αποτελείται από επίπεδα όπου το κάθε ένα από αυτά χωρίζεται στα παρακάτω μέρη:

$$z(x) = w^{\mathrm{T}}x + b$$

$$f(x) = a(z(x))$$

όπου w, b είναι τα βάρη του επιπέδου. Η έξοδος z(x) είναι η απόκριση κάθε νευρώνα πριν την συνάρτηση ενεργοποίησης ενώ η f(x) μετά. Κάθε επίπεδο συνδέεται με ένα επόμενο του οποίου η είσοδός του αποτελεί την έξοδο (με την συνάρτηση ενεργοποίησης) του προηγούμενου.

Στο μέρος αυτό καλείστε να συμπληρώσετε ορισμένα σημεία κώδικα ώστε να επιτυγχάνεται αυτή η λειτουργικότητα. Στην συνέχεια θα υλοποιήσετε το ίδιο ακριβώς μοντέλο χρησιμοποιώντας όμως μια έτοιμη βιβλιοθήκη και θα συγκρίνετε τα αποτελέσματά τους (χρόνο, σκορ κ.α.).

Σε αυτό το σημείο της άσκησης θα επιλύσετε το παραπάνω πρόβλημα κατασκευάζοντας ένα πολυεπίπεδο νευρωνικό δίκτυο. Αρχικά θα υλοποιήσετε το νευρωνικό χωρίς να χρησιμοποιήσετε κάποια έτοιμη κλάση κάποιας βιβλιοθήκης (όπως scikit-learn, keras), ενώ στην συνέχεια θα κατασκευάσετε το ίδιο σύστημα με την χρήση της βιβλιοθήκης scikit-learn.

Στο παρακάτω κελί κώδικα σάς δίνεται η βασική δομή του επιπέδου ενός πολυεπίπεδου νευρωνικού δικτύου. Η παρακάτω κλάση δεν υλοποιεί κάποιο πραγματικό επίπεδο (όπως Dense) αλλά αυτή χρησιμοποιείται για την παρουσίαση των λειτουργιών κάθε επιπέδου.

Ουσιαστικά κάθε επίπεδο ενός νευρωνικού δικτύου πρέπει να είναι σε θέση:

1. Για μια είσοδο να υπολογίζει την έξοδο κάθε νευρώνα. Αυτό επιτυγχάνεται μέσω της μεθόδου forward η όποια δέχεται ως όρισμα μια είσοδο και επιστρέφει έναν πίνακα με τις εξόδους κάθε νευρώνα του επιπέδου.

2. Να υπολογίζει τις μεταβολές οι όποιες πρέπει να γίνουν στα βάρη κάθε επιπέδου, ανάλογα με το πόσο καλά-κοντινά ήταν τα αποτελέσματα του επιπέδου στα πραγματικά. Η λειτουργία αυτή θα μας βοηθήσει στην ανανέωση των βαρών του δικτύου και συνεπώς στη σωστή εκπαίδευσή του. Η λειτουργικότητα αυτή επιτυγχάνεται μέσω της μεθόδου backward.

Η λειτουργικότητα, συνεπώς, κάθε επιπέδου καθορίζεται από την συνάρτηση που υλοποιείται στην μέθοδο forward. Ένα instance της παρακάτω κλάσης, συνεπώς, επιστρέφει ως έξοδο την είσοδο κάθε νευρώνα (ταυτοτική συνάρτηση) όποτε δεν προσφέρει κάποια υψηλή λειτουργικότητα. Στην παρακάτω κλάση δεν έχετε να προσθέσετε κάτι, απλά να μελετήσετε και να καταλάβετε την δομή που πρέπει να έχει ένα επίπεδο.

In [ ]:
```
class Layer:
    def __init__(self):
        """Here we can initialize layer parameters (if any) and auxiliary stuff."""
        # A dummy layer does nothing
        pass

    def forward(self, input):
        """
        Takes input data of shape [batch, input_units], returns output data [batch, output_units]
        """
        # A dummy layer just returns whatever it gets as input.
        return input

    def backward(self, input, grad_output):
        # The gradient of a dummy layer is precisely grad_output, but we'll write it more explicitly
        num_units = input.shape[1]
```

```
        d_layer_d_input = np.eye(num_units)

        return np.dot(grad_output, d_layer_d_input) # chain rule
```

Στο σημείο αυτό αξίζει να αναφερθεί ότι για την σωστή εκπαίδευση του δικτύου (σε πρακτικό επίπεδο) πρέπει να διαχωριστεί η έξοδος κάθε νευρώνα πριν και μετά την συνάρτηση ενεργοποίησης. Έτσι η παραπάνω μέθοδος forward της κλάσης layer πρέπει να υπολογίζει την έξοδο του επιπέδου χωρίς την συνάρτηση ενεργοποίησης και κάποια άλλη κλάση να υπολογίζει το αποτέλεσμα με αυτή.

Έκτος όμως από την εκπαίδευση του δικτύου, ο διαχωρισμός αυτός μας βοηθά σημαντικά και κατά την φάση σχεδιασμού της αρχιτεκτονικής μιας και μας δίνει την δυνατότητα να αλλάζουμε την συνάρτηση ενεργοποίησης χωρίς κάθε φόρα να πρέπει να αλλάξουμε ολόκληρη την κλάση layer. Για τους παραπάνω λόγους θα χειριζόμαστε την συνάρτηση ενεργοποίησης σαν ένα ξεχωριστό επίπεδο με τις δικές της μεθόδους: forward, backward.

Παρακάτω παρουσιάζεται η κλάση η όποια υλοποιεί την λειτουργικότητα της συνάρτησης ενεργοποίησης ReLU).

Με την ίδια λογική μπορούμε να υλοποιήσουμε οποιαδήποτε άλλη συνάρτηση ενεργοποίησης θέλουμε π.χ. sigmoid, tanh κ.ο.κ. και επιπλέον μπορούμε να τις εναλλάσσουμε μεταξύ επιπέδων χωρίς δυσκολία.

In [ ]:
```
class ReLU(Layer):
    def __init__(self):
        """ReLU layer simply applies elementwise rectified linear unit to all inputs"""
        pass

    def forward(self, input):
        """Apply elementwise ReLU to [batch, input_units] matrix"""
        relu_forward = np.maximum(0, input)
        return relu_forward

    def backward(self, input, grad_output):
        """Compute gradient of loss w.r.t. ReLU input"""
        relu_grad = input > 0
        return grad_output*relu_grad
```

Η κλάση Dense υλοποιεί ένα επίπεδο dense όπου η έξοδος κάθε νευρώνα (χωρίς τη συνάρτηση ενεργοποίησης) υπολογίζεται από την παρακάτω εξίσωση:

$$z(x) = w^{\mathrm{T}}x + b$$

όπου w, b είναι τα βάρη του επιπέδου.

Συνεπώς το δίκτυο είναι απαραίτητο να διατηρεί τους δυο πίνακες με τα βάρη οι οποίοι στην μέθοδο forward θα χρησιμοποιούνται για τον υπολογισμό της εξόδου και θα ανανεώνονται από την μέθοδο backward. Οι πίνακες αυτοί δημιουργούνται κατά την κατασκευή κάθε στιγμιότυπου και αρχικοποιούνται, ο πρώτος τυχαία και ο δεύτερος με μηδενικά. Στο σημείο αυτό καλείστε να συμπληρώσετε την μέθοδο forward με κατάλληλο τρόπο ώστε να επιτυγχάνεται η επιθυμητή λειτουργικότητα.

In [ ]:
```
class Dense(Layer):
    def __init__(self, input_units, output_units, learning_rate = 0.1):
        self.input_units = input_units
        self.output_units = output_units

        self.learning_rate = learning_rate
        self.weights = np.random.normal(loc = 0.0,
                                        scale = np.sqrt(2 / (input_units + output_units)),
                                        size = (input_units, output_units))
        self.biases = np.zeros(output_units)

    def forward(self, inputs):
        """
        Perform an affine transformation:
        f(x) = <W*x> + b

        input shape: [number of inputs, input units]
        output shape: [number of inputs, output units]
        """
        ###################
        ## Your code below
        ## hint: numpy.dot

        output = np.dot(inputs, self.weights) + self.biases

        ## Your code above
        ###################
```

```
        return output

    def backward(self, inputs, grad_output):
        # compute d f / d x = d f / d dense * d dense / d x
        # where d dense/ d x = weights transposed
        grad_input = np.dot(grad_output, self.weights.T)

        # compute gradient w.r.t. weights and biases
        grad_weights = np.dot(inputs.T, grad_output)
        grad_biases = grad_output.mean(axis = 0) * inputs.shape[0]
        assert grad_weights.shape == self.weights.shape and grad_biases.shape == self.biases.shape

        # Here we perform a stochastic gradient descent step.
        self.weights = self.weights - self.learning_rate * grad_weights
        self.biases = self.biases - self.learning_rate * grad_biases
        return grad_input
```

Οι παρακάτω συναρτήσεις χρησιμοποιούνται για να μπορεί το δίκτυο να ελέγχει πόσο κοντά βρίσκονται τα αποτελέσματά του στα πραγματικά (Loss function). Όπως είναι λογικό υπάρχουν διαφορετικές τέτοιες συναρτήσεις ανάλογα το πρόβλημα που καλείται να λύσει το δίκτυο. Η παρακάτω συνάρτηση ονομάζεται softmax και χρησιμοποιείται κατά κύριο λόγο σε προβλήματα ταξινόμησης όπως το συγκεκριμένο. Η softmax δέχεται σαν είσοδο τις ενεργοποιήσεις του τελευταίου επιπέδου και επιστρέφει μια κατανομή πιθανοτήτων για κάθε μια από τις κλάσεις εξόδου (π.χ. κλάση 0 έχει πιθανότητα 0.001, η κλάση 1 έχει 0.9 κ.ο.κ.).

In [ ]:
```
def softmax_crossentropy_with_logits(logits, reference_answers):
    logits_for_answers = logits[np.arange(len(logits)), reference_answers]
    xentropy = - logits_for_answers + np.log(np.sum(np.exp(logits), axis = -1))
    return xentropy

def grad_softmax_crossentropy_with_logits(logits, reference_answers):
    ones_for_answers = np.zeros_like(logits)
    ones_for_answers[np.arange(len(logits)), reference_answers] = 1
    softmax = np.exp(logits) / np.exp(logits).sum(axis = -1, keepdims = True)
    return (- ones_for_answers + softmax) / logits.shape[0]
```

Έχοντας υλοποιήσει τις κλάσεις Dense και ReLU μπορούμε πλέον να κατασκευάσουμε μια κλάση η όποια θα ορίζει ένα πολυεπίπεδο νευρωνικό δίκτυο (MLP). Το δίκτυο αυτό ουσιαστικά αποτελείται από μια ακολουθία Dense επιπέδων όπου το κάθε ένα (εκτός του τελευταίου) ακολουθείται από μια μη-γραμμική συνάρτηση ενεργοποίησης (ReLU). Όμοια με πριν, η κλάση αυτή πρέπει να περιέχει μια μέθοδο forward η όποια θα δέχεται μια είσοδο (εδώ μια εικόνα flatten) και θα επιστρέφει μια έξοδο (εδώ μια κατανομή 3 πιθανοτήτων). Παράλληλα πρέπει να περιέχει και μια μέθοδο fit, η όποια θα εκπαιδεύει το δίκτυο δεδομένου ενός τέτοιου συνόλου (εδώ του x_train). Στο σημείο αυτό χρησιμοποιούνται οι μέθοδοι backward που έχουν οριστεί για κάθε ένα επίπεδο (δεν χρειάζεται να συμπληρώσετε κάτι). Τέλος θα ήταν βοηθητικό να έχουμε και μια μέθοδο η όποια θα μετατρέπει την κατανομή εξόδου στην επιστρεφόμενη κλάση (predict) για κάποιο ή κάποια στιγμιότυπα του συνόλου δεδομένων.

Το δίκτυο όπως αναφέρθηκε και προηγουμένως αποτελείται από έναν αριθμό Dense επιπέδων κάθε ένα από τα όποια ακολουθείται από μια συνάρτηση ReLU. Η κατασκευή των επιπέδων γίνεται κατά την στιγμή δημιουργίας του δικτύου, όπου δίνεται ως είσοδος μια λίστα με το μέγεθος κάθε επιπέδου, μαζί με το μέγεθος εισόδου. Έτσι για παράδειγμα η παρακάτω γραμμή κώδικα:

```
net = MLP([100, 200, 100, 3], 6)
```

κατασκευάζει ένα MLP το όποιο αποτελείται από 4 επίπεδα με μέγεθος 100, 200, 100, 10. Ο αριθμός των επιπέδων καθώς και του μεγέθους καθενός από αυτά είναι ελεύθερος να οριστεί από τον χρήστη.
Στον constructor της κλάσης ουσιαστικά ορίζεται μια λίστα η όποια περιέχει κάθε ένα από τα επίπεδα που πρέπει να οριστούν, π.χ. για το παραπάνω παράδειγμα η μεταβλητή net.network περιέχει τα εξής στιγμιότυπα των κλάσεων:

```
[Dense(100), ReLU(), Dense(200), ReLU(), Dense(100), ReLU(), Dense(10)]
```

Συνεπώς η λειτουργικότητα του δικτύου όπως και πριν πρέπει να οριστεί στην μέθοδο forward. Στο σημείο αυτό καλείστε να συμπληρώσετε την μέθοδο αυτή έτσι ώστε το δίκτυο να λειτουργεί όπως πρέπει, δηλαδή στο παράδειγμά μας η είσοδος να περνά από το επίπεδο Dense(100), μετά από το ReLU(), στην συνέχεια από το Dense(200) κ.ο.κ. μέχρι και το τελευταίο επίπεδο. Ο αλγόριθμος αυτός παρουσιάζεται και σε ψευδοκώδικα στην διαφάνεια 33 του μαθήματος.

In [ ]:
```
class MLP:
    def __init__(self, shapes, input_dim):
        self.shapes = shapes
        self.network = [Dense(input_dim, shapes[0])]
        self.network.append(ReLU())
        for i in range(1, len(self.shapes) - 1):
            self.network.append(Dense(shapes[i-1], shapes[i]))
            self.network.append(ReLU())
```

```
        self.network.append(Dense(shapes[i], shapes[-1]))

    def forward(self, X):
        """
        Αλγόριθμος διαφάνειας 33
        """
        activations = []
        inputs = X
        # Looping through each layer
        for l in self.network:
            ###################
            ## Your code below
            # hint: τροφοδοτούμε την έξοδο κάθε επιπέδου στο επόμενο

            inputs = l.forward(inputs)
            activations.append(inputs)

            ## Your code above
            ###################
        assert len(activations) == len(self.network)
        return activations

    def predict(self,X):
        """
        Προβλέπει την έξοδο του δικτύου για ένα ή περισσότερα στιγμιότυπα εισόδου
        """
        logits = self.forward(X)[-1]
        return logits.argmax(axis = -1)

    def fit(self, X, y):
        # Get the layer activations
        layer_activations = self.forward(X)
        layer_inputs = [X] + layer_activations
        logits = layer_activations[-1]

        # Compute the loss and the initial gradient
        loss = softmax_crossentropy_with_logits(logits, y)
        loss_grad = grad_softmax_crossentropy_with_logits(logits, y)

        # Propagate gradients through the network
        # Reverse propogation as this is backprop
        for layer_index in range(len(self.network))[::-1]:
            layer = self.network[layer_index]
            loss_grad = layer.backward(layer_inputs[layer_index], loss_grad)
        return np.mean(loss)
```

Αξιολόγηση ενός Multi-Layer Perceptron

Αφού έχουμε κατασκευάσει τα παραπάνω είμαστε πλέον σε θέση να εκπαιδεύσουμε το MLP. Αυτό γίνεται καλώντας την μέθοδο fit. Στο παρακάτω κελί κώδικα ορίζεται το MLP του παραπάνω παραδείγματος και εκπαιδεύεται για 25 εποχές. Στο τέλος κάθε εποχής παρουσιάζονται τα αποτελέσματα του μαζί με μια γραφική των train και test accuracy.

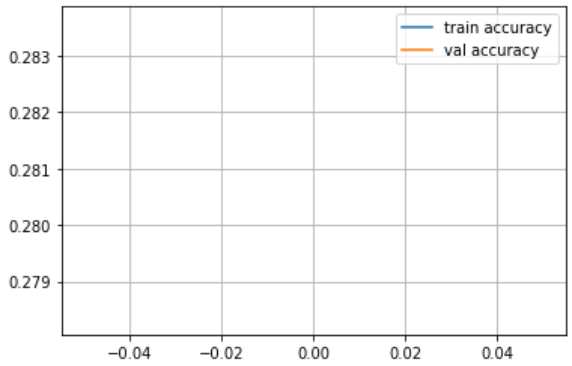
In [ ]:
```
type(x_train)
```

Out[ ]:
```
numpy.ndarray
```

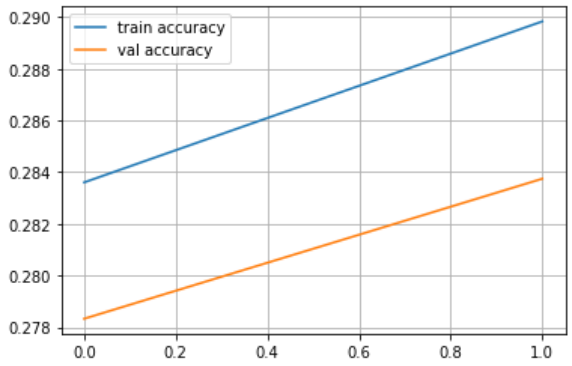
In [ ]:
```
%%time
from IPython.display import clear_output
import numpy as np

network = MLP([10, 15, 20, 3], 6)

train_log = []
val_log = []

for epoch in range(100):
    network.fit(x_train, y_train)
    train_log.append(np.mean(network.predict(x_train) == y_train))
    val_log.append(np.mean(network.predict(x_test) == y_test))
    #clear_output()
    print("Epoch", epoch)
    print("Train accuracy:", train_log[-1])
    print("Val accuracy:", val_log[-1])
    plt.plot(train_log, label = 'train accuracy')
    plt.plot(val_log, label = 'val accuracy')
    plt.legend(loc = 'best')
```

```
    plt.grid()
    plt.show()
```

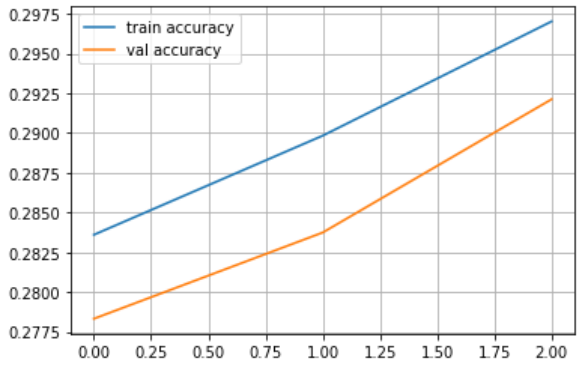
```
Epoch 0
Train accuracy: 0.2836025762855031
Val accuracy: 0.27832512315270935
```



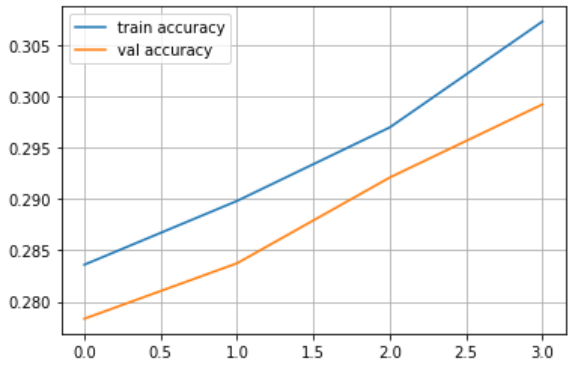

```
Epoch 1
Train accuracy: 0.2898321191004118
Val accuracy: 0.28374384236453204
```



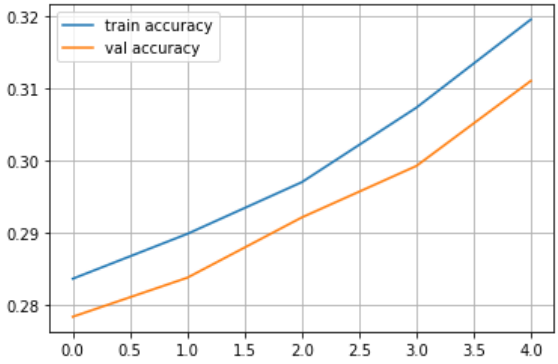

```
Epoch 2
Train accuracy: 0.2970119311582726
Val accuracy: 0.29211822660098524
```



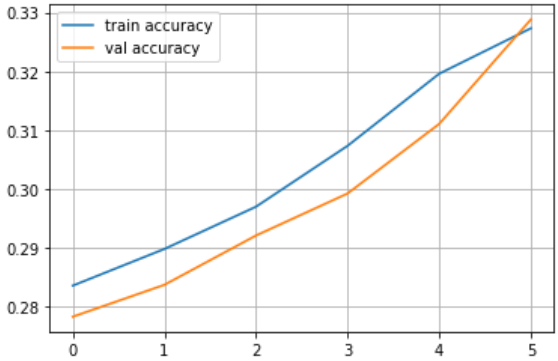

```
Epoch 3
Train accuracy: 0.30735930735930733
Val accuracy: 0.29926108374384236
```



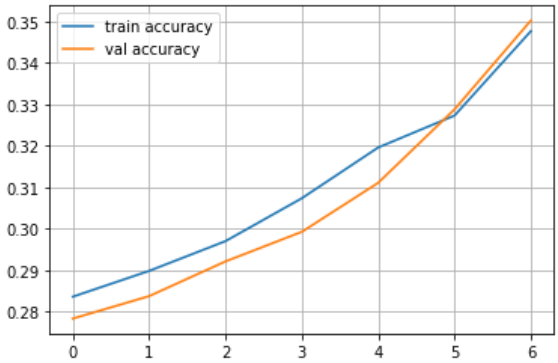

```
Epoch 4
Train accuracy: 0.3196072220462464
Val accuracy: 0.31108374384236454
```



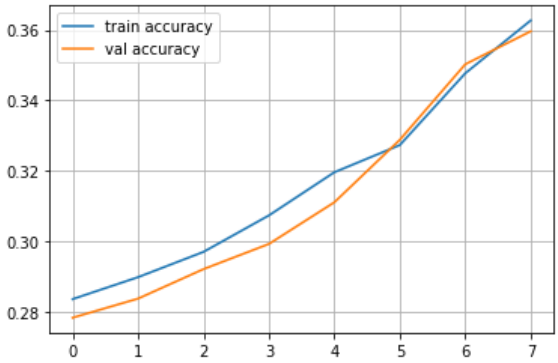
Epoch 5
Train accuracy: 0.3273149614613029
Val accuracy: 0.3288177339901478



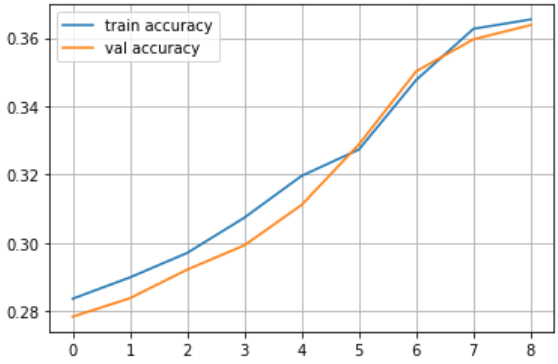
Epoch 6
Train accuracy: 0.347692957449055
Val accuracy: 0.3502463054187192



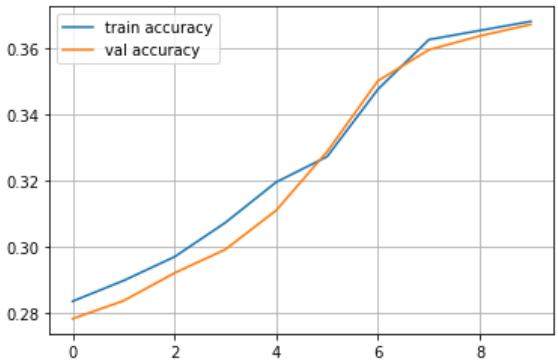
Epoch 7
Train accuracy: 0.3626860943934115
Val accuracy: 0.35960591133004927



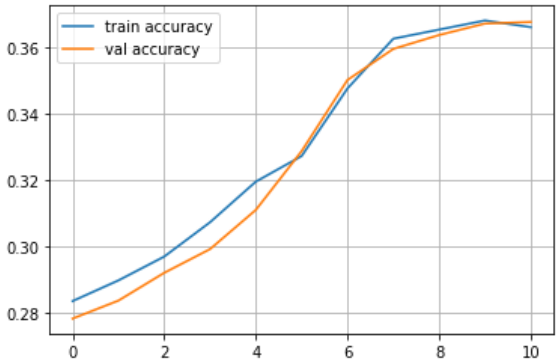
Epoch 8
Train accuracy: 0.36543131665082884
Val accuracy: 0.36379310344827587



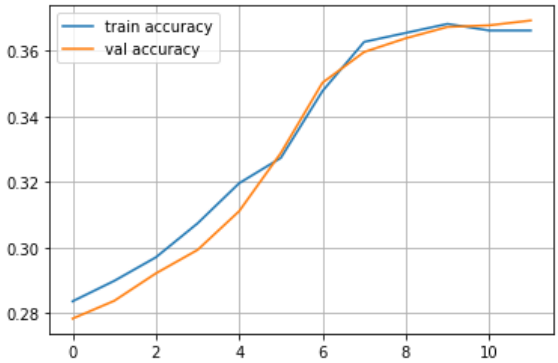
Epoch 9
Train accuracy: 0.3681765389082462
Val accuracy: 0.36724137931034484



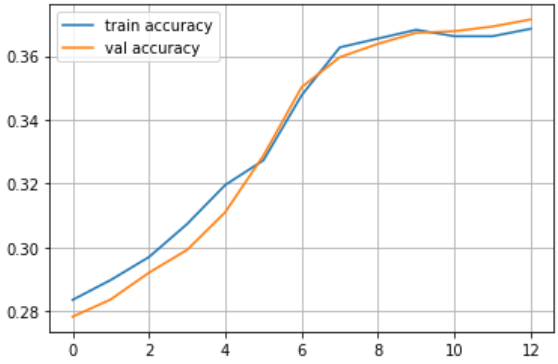
Epoch 10
Train accuracy: 0.36617041495090275
Val accuracy: 0.36773399014778324



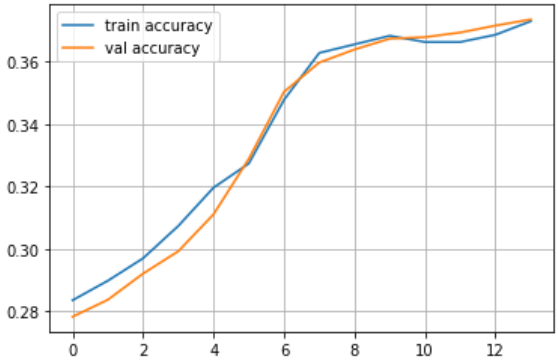
Epoch 11
Train accuracy: 0.36617041495090275
Val accuracy: 0.3692118226600985



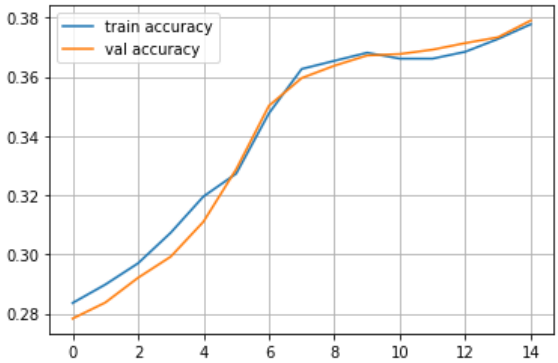
Epoch 12
Train accuracy: 0.36849329532256364
Val accuracy: 0.37142857142857144



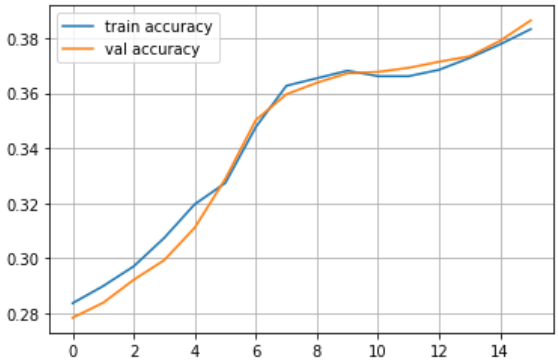
Epoch 13
Train accuracy: 0.37282229965156793
Val accuracy: 0.3733990147783251



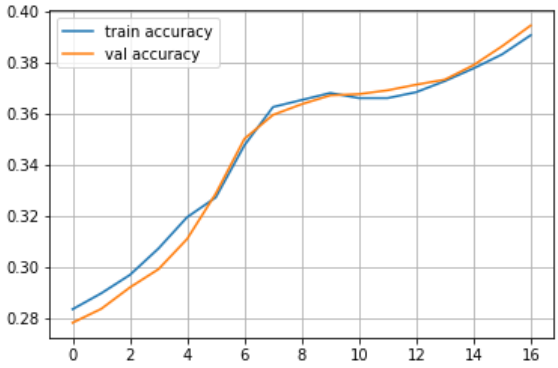
Epoch 14
Train accuracy: 0.37778481680920706
Val accuracy: 0.379064039408867



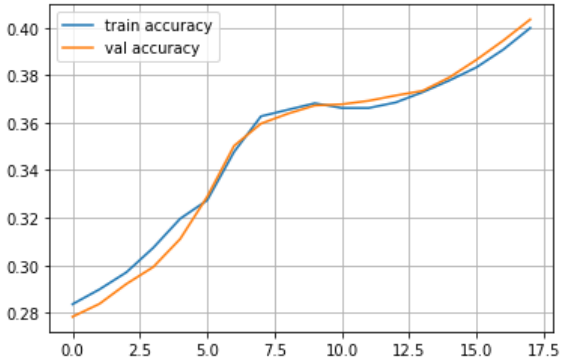
Epoch 15
Train accuracy: 0.3832752613240418
Val accuracy: 0.38645320197044336



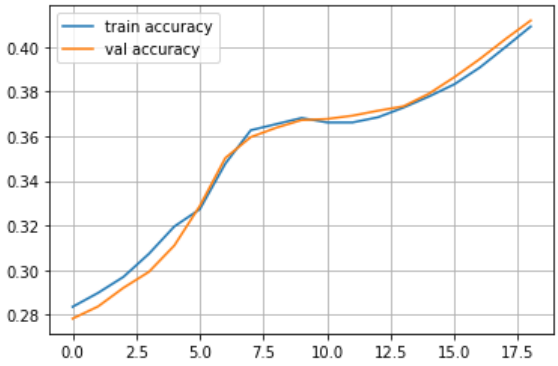
Epoch 16
Train accuracy: 0.39077182979622005
Val accuracy: 0.39458128078817734



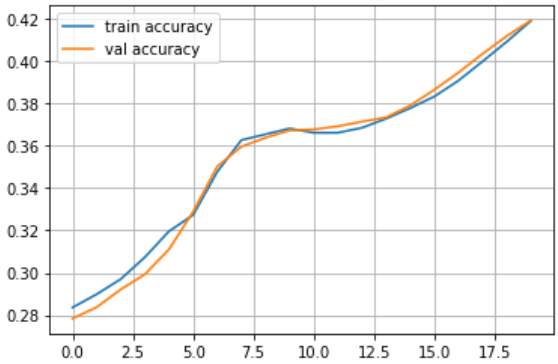
Epoch 17
Train accuracy: 0.3998521803399852
Val accuracy: 0.40344827586206894



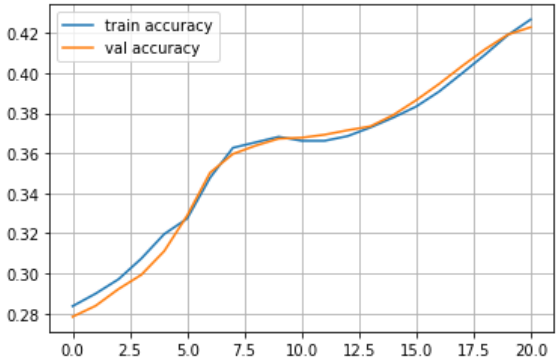
Epoch 18
Train accuracy: 0.4091437018266287
Val accuracy: 0.41182266009852214



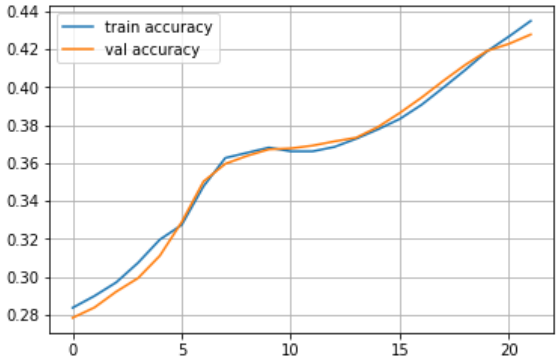
Epoch 19
Train accuracy: 0.41896315067046774
Val accuracy: 0.41921182266009854



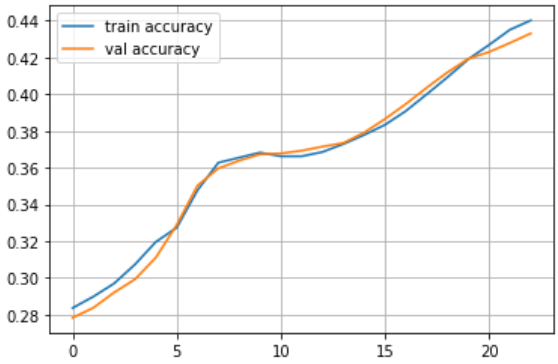
Epoch 20
Train accuracy: 0.42677647555696335
Val accuracy: 0.4229064039408867



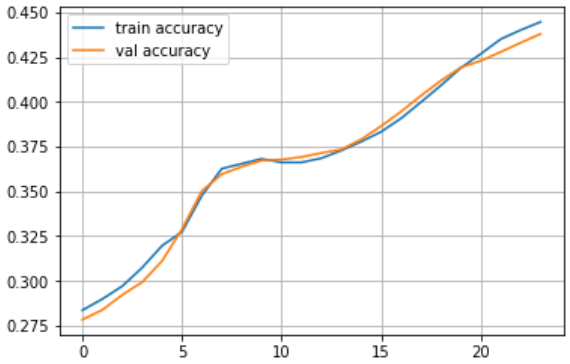
Epoch 21
Train accuracy: 0.4350121423292155
Val accuracy: 0.4278325123152709



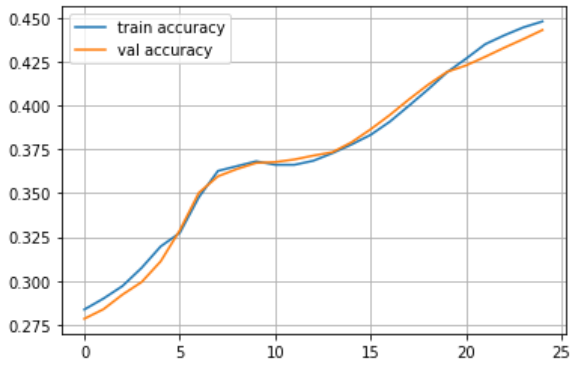
Epoch 22
Train accuracy: 0.44008024495829373
Val accuracy: 0.4330049261083744



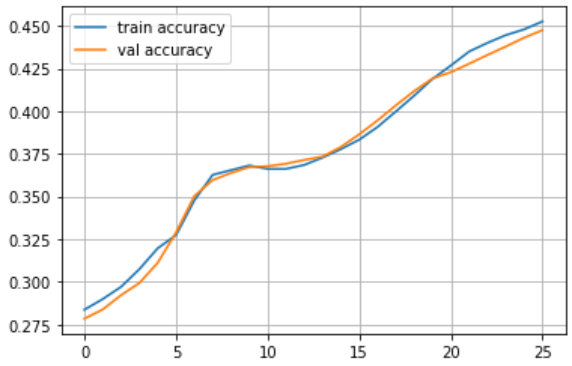
Epoch 23
Train accuracy: 0.44462042023017634
Val accuracy: 0.4379310344827586



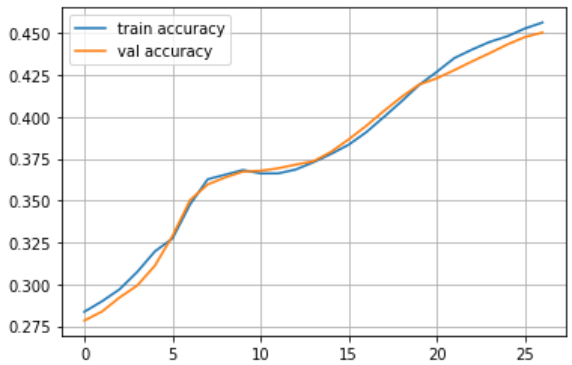
Epoch 24
Train accuracy: 0.44799915531622847
Val accuracy: 0.44310344827586207



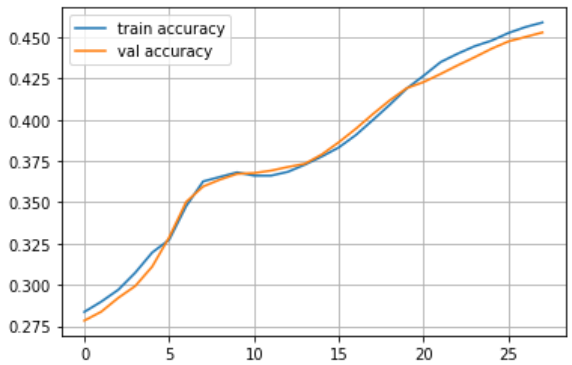
Epoch 25
Train accuracy: 0.4526449160595502
Val accuracy: 0.4475369458128079



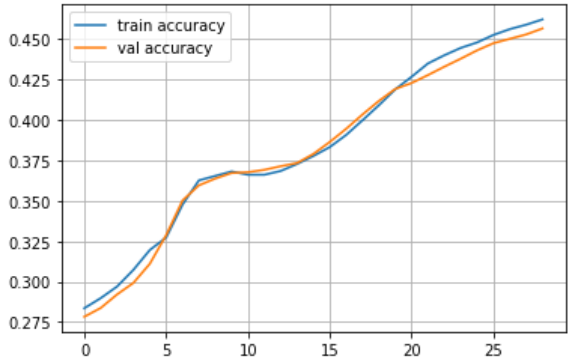
Epoch 26
Train accuracy: 0.45623482208848065
Val accuracy: 0.45024630541871924



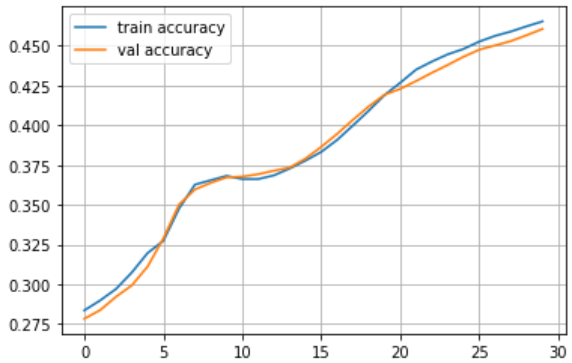
Epoch 27
Train accuracy: 0.458980044345898
Val accuracy: 0.4529556650246305



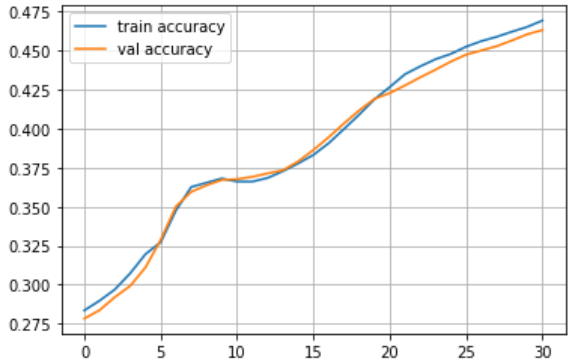
Epoch 28
Train accuracy: 0.462253193960511
Val accuracy: 0.4566502463054187



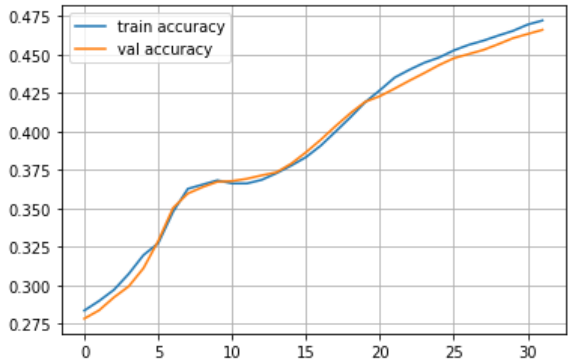
Epoch 29
Train accuracy: 0.4653151726322458
Val accuracy: 0.4605911330049261



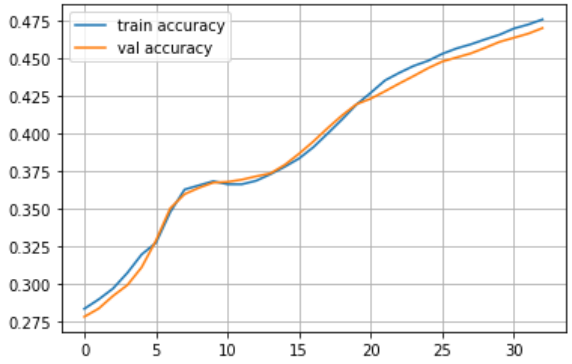
Epoch 30
Train accuracy: 0.46943300601837185
Val accuracy: 0.46330049261083744



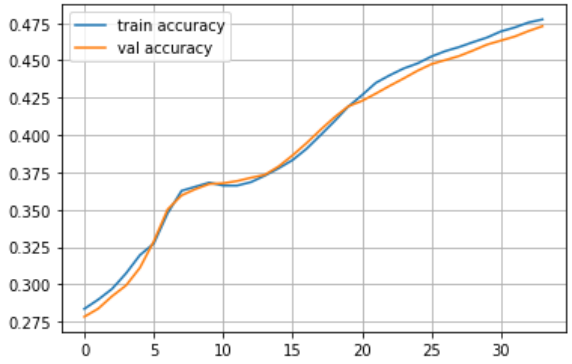
Epoch 31
Train accuracy: 0.47207264280435013
Val accuracy: 0.4660098522167488



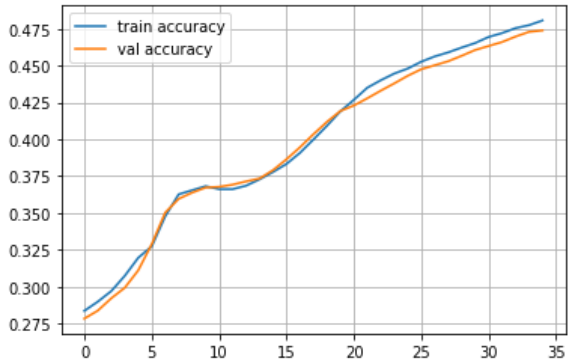
Epoch 32
Train accuracy: 0.47545137789040226
Val accuracy: 0.46970443349753693



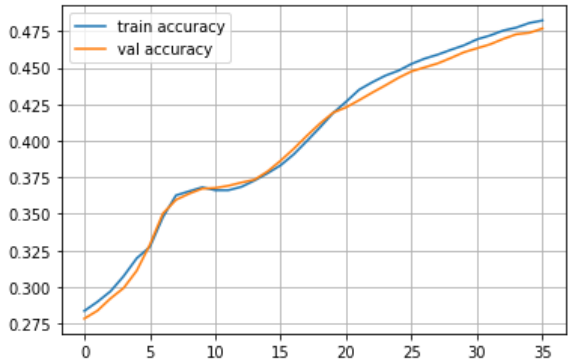
Epoch 33
Train accuracy: 0.47745750184774577
Val accuracy: 0.4729064039408867



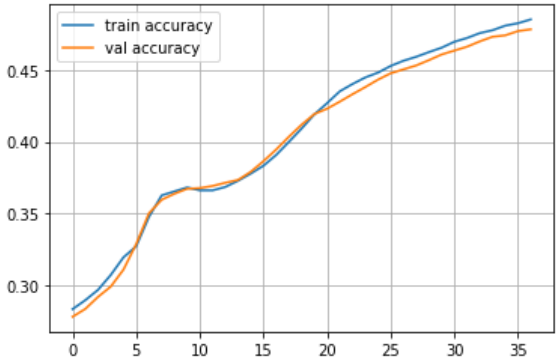
Epoch 34
Train accuracy: 0.48062506599091964
Val accuracy: 0.47389162561576353



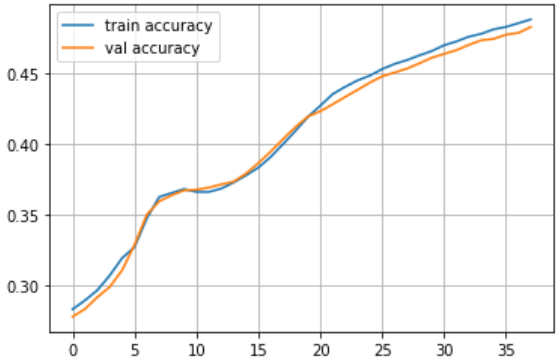
Epoch 35
Train accuracy: 0.4823144335339457
Val accuracy: 0.4768472906403941



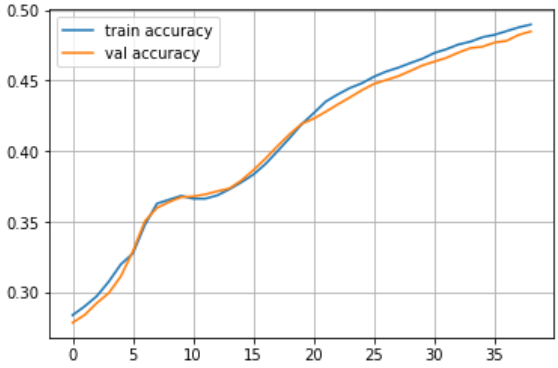
Epoch 36
Train accuracy: 0.484954070319924
Val accuracy: 0.47807881773399014



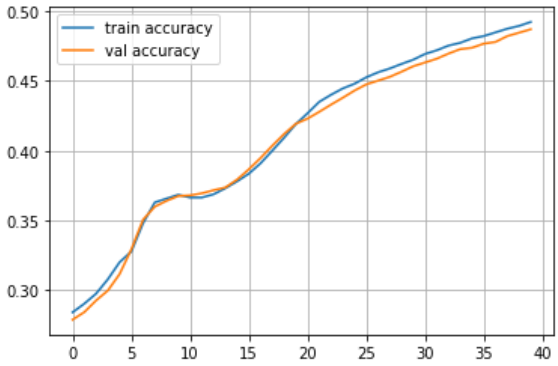
Epoch 37
Train accuracy: 0.4875937071059022
Val accuracy: 0.48226600985221674



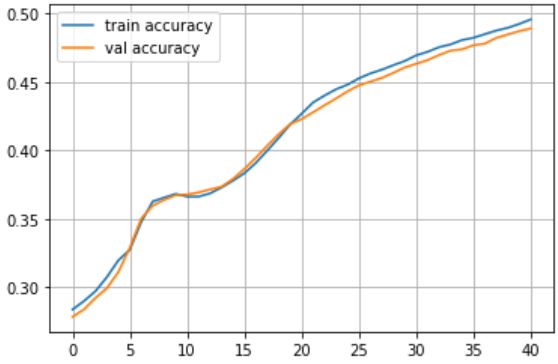
Epoch 38
Train accuracy: 0.48959983106324567
Val accuracy: 0.48472906403940885



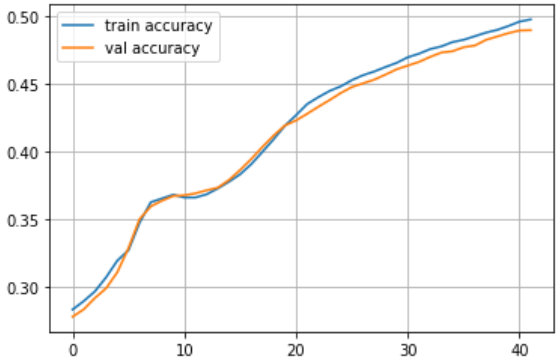
Epoch 39
Train accuracy: 0.4924506387921022
Val accuracy: 0.48719211822660097



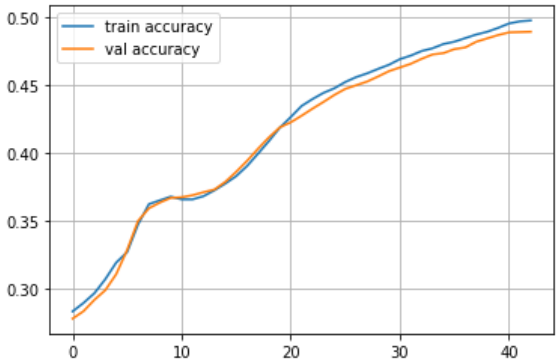
Epoch 40
Train accuracy: 0.4957237884067152
Val accuracy: 0.4891625615763547



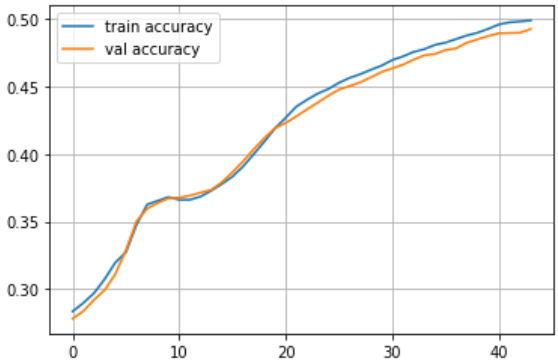
Epoch 41
Train accuracy: 0.4973075704783022
Val accuracy: 0.4894088669950739



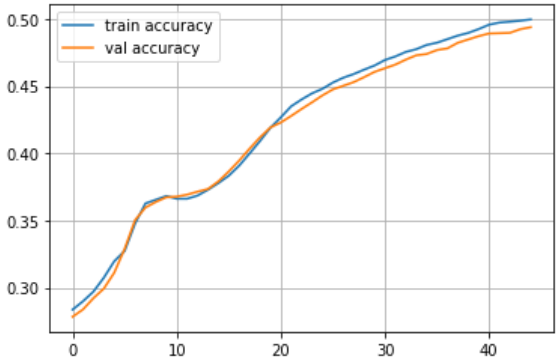
Epoch 42
Train accuracy: 0.497941083306937
Val accuracy: 0.4896551724137931



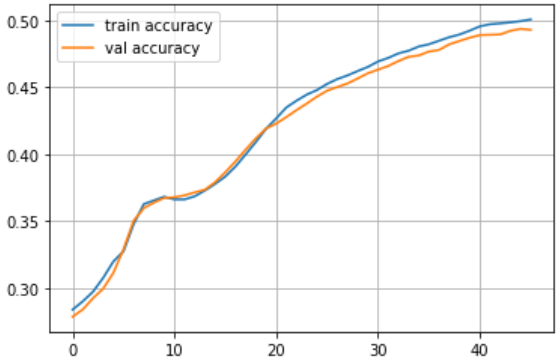
Epoch 43
Train accuracy: 0.4986801816070109
Val accuracy: 0.4923645320197044



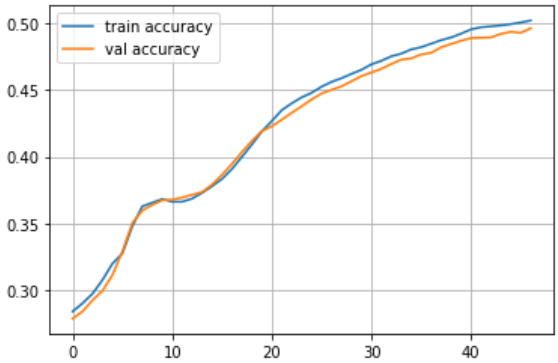
Epoch 44
Train accuracy: 0.49963045084996305
Val accuracy: 0.4938423645320197



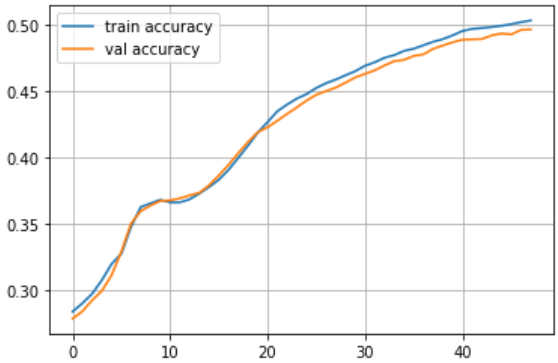
Epoch 45
Train accuracy: 0.5008974765072326
Val accuracy: 0.49310344827586206



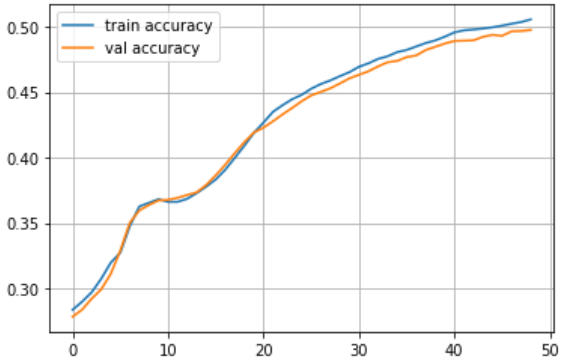
Epoch 46
Train accuracy: 0.5023756731073804
Val accuracy: 0.496551724137931



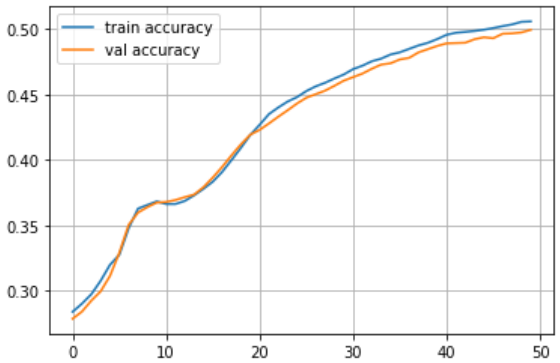
Epoch 47
Train accuracy: 0.50364269876465
Val accuracy: 0.49679802955665026



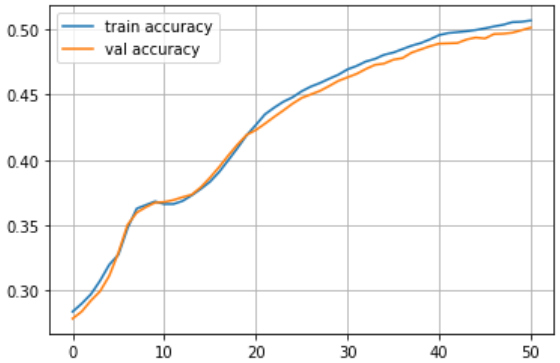
Epoch 48
Train accuracy: 0.5056488227219934
Val accuracy: 0.4975369458128079



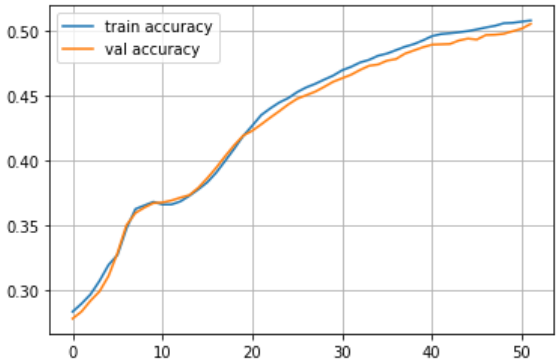
Epoch 49
Train accuracy: 0.5059655791363108
Val accuracy: 0.4995073891625616



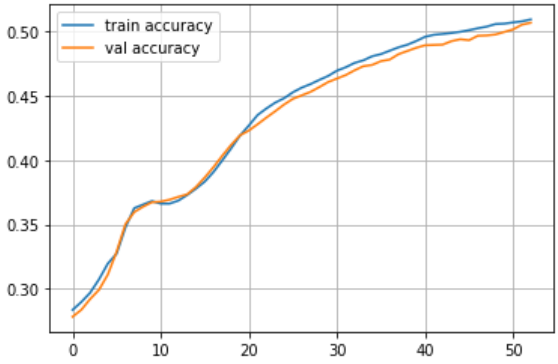
Epoch 50
Train accuracy: 0.5069158483792631
Val accuracy: 0.5014778325123153



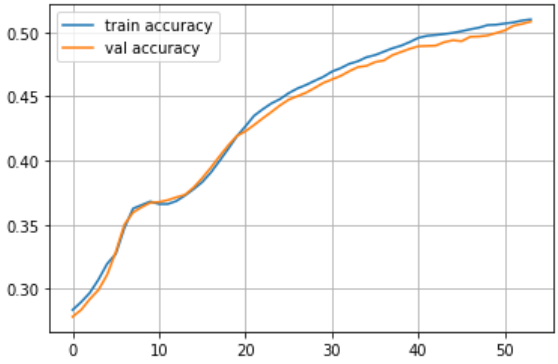
Epoch 51
Train accuracy: 0.5077605321507761
Val accuracy: 0.5051724137931034



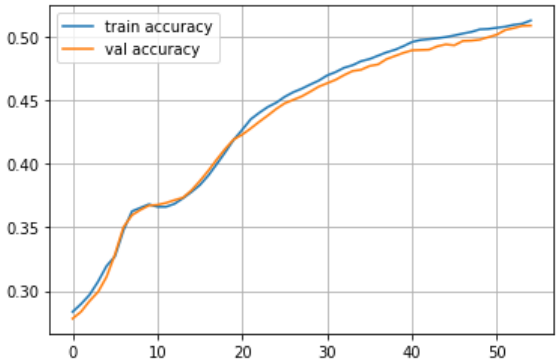
Epoch 52
Train accuracy: 0.5092387287509239
Val accuracy: 0.5066502463054188



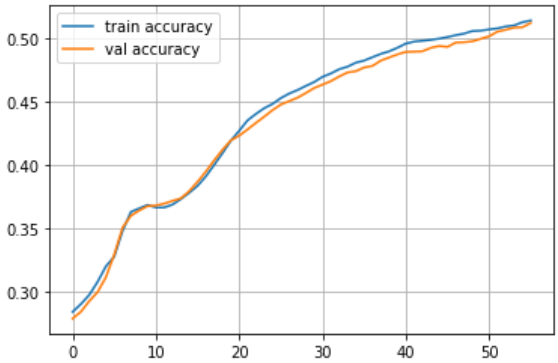
Epoch 53
Train accuracy: 0.5100834125224369
Val accuracy: 0.5083743842364532



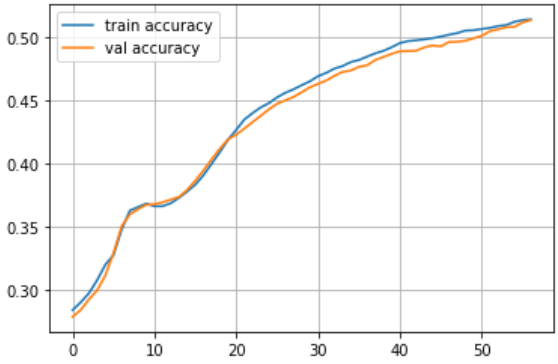
Epoch 54
Train accuracy: 0.512617463836976
Val accuracy: 0.5086206896551724



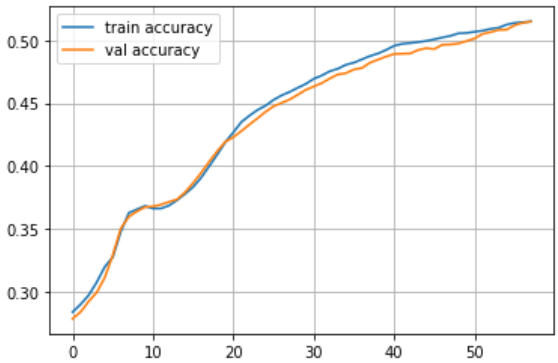
Epoch 55
Train accuracy: 0.5139900749656847
Val accuracy: 0.5120689655172413



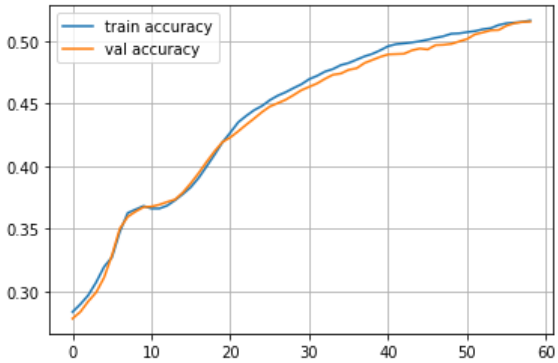
Epoch 56
Train accuracy: 0.5145180023228804
Val accuracy: 0.5140394088669951



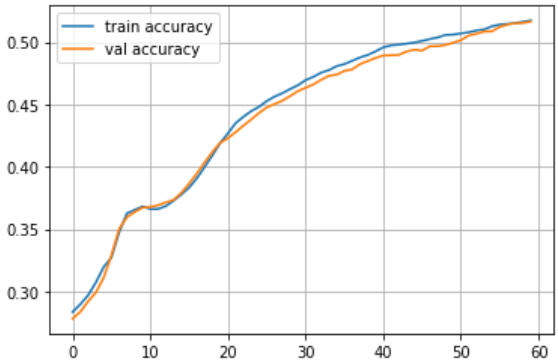
Epoch 57
Train accuracy: 0.5151515151515151
Val accuracy: 0.515024630541872



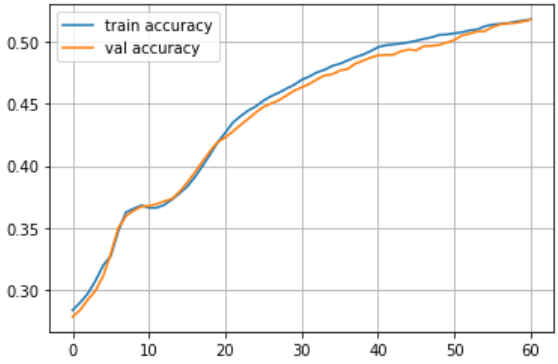
Epoch 58
Train accuracy: 0.5162073698659064
Val accuracy: 0.5152709359605911



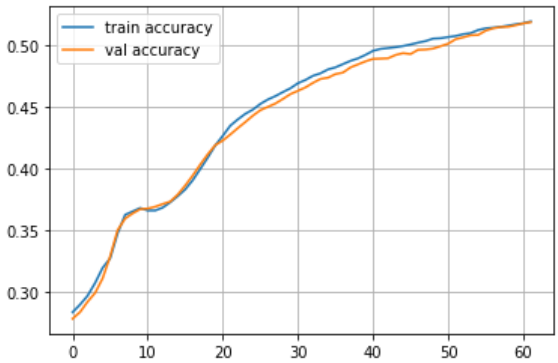
Epoch 59
Train accuracy: 0.5172632245802977
Val accuracy: 0.5165024630541872



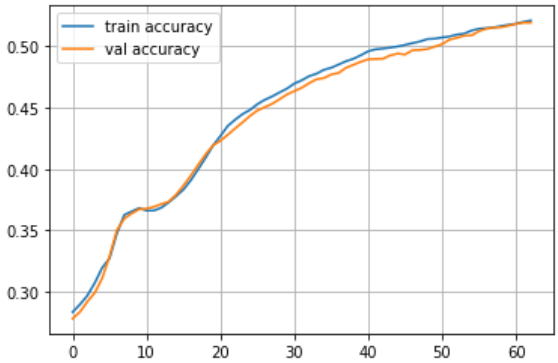
Epoch 60
Train accuracy: 0.5181079083518108
Val accuracy: 0.5179802955665025



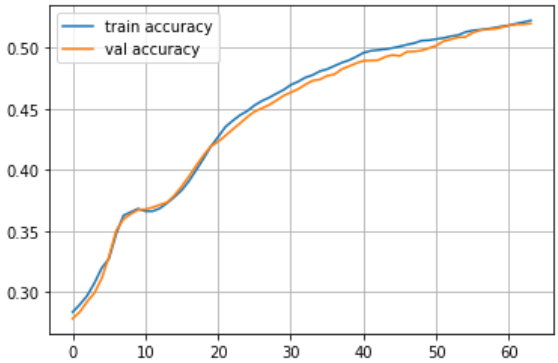
Epoch 61
Train accuracy: 0.5195861049519586
Val accuracy: 0.5189655172413793



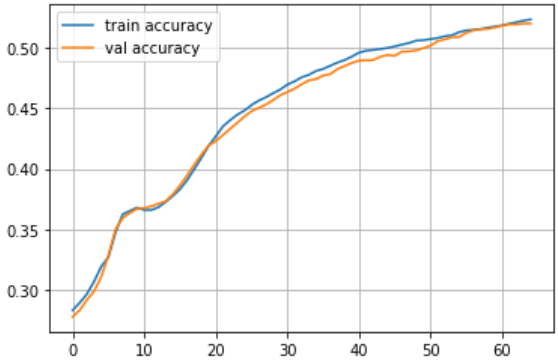
Epoch 62
Train accuracy: 0.5207475451377891
Val accuracy: 0.5189655172413793



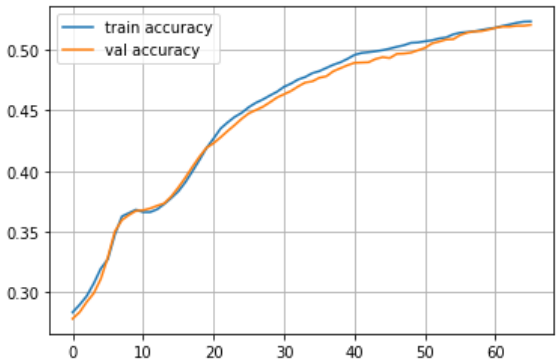
Epoch 63
Train accuracy: 0.5221201562664978
Val accuracy: 0.5197044334975369



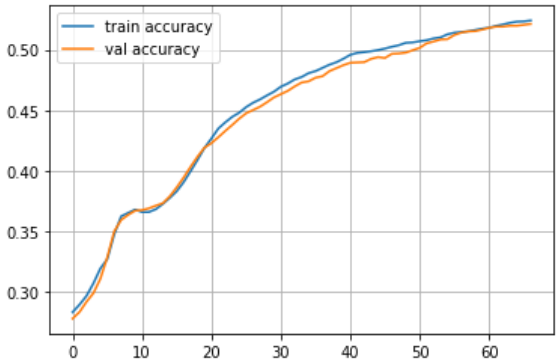
Epoch 64
Train accuracy: 0.5230704255094499
Val accuracy: 0.5197044334975369



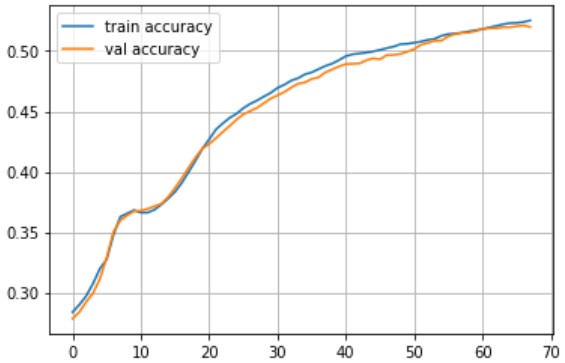
Epoch 65
Train accuracy: 0.5232815964523282
Val accuracy: 0.5204433497536946



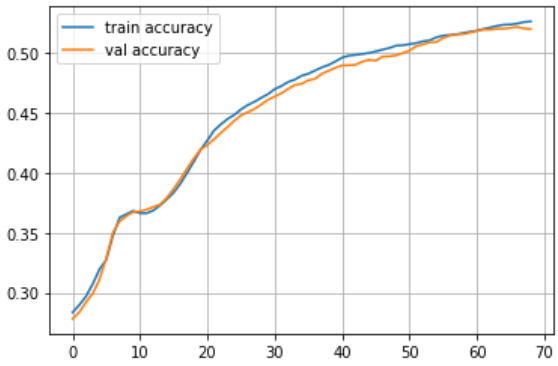
Epoch 66
Train accuracy: 0.5240206947524021
Val accuracy: 0.5211822660098522



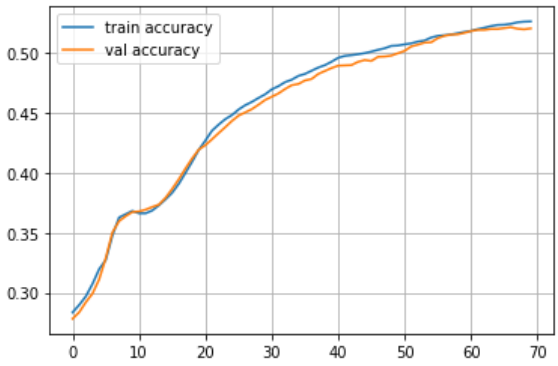
Epoch 67
Train accuracy: 0.5252877204096716
Val accuracy: 0.5199507389162562



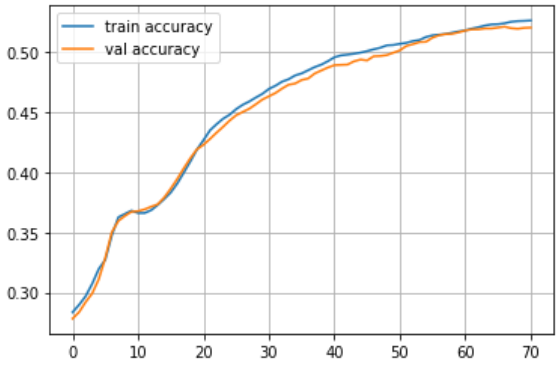
Epoch 68
Train accuracy: 0.5258156477668673
Val accuracy: 0.5194581280788177



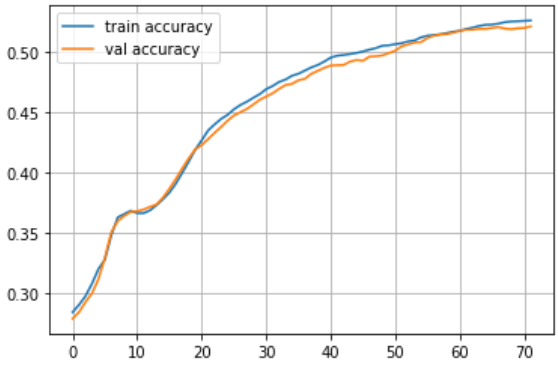
Epoch 69
Train accuracy: 0.5260268187097455
Val accuracy: 0.5201970443349754



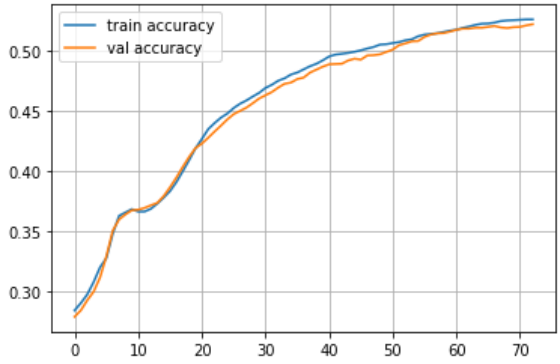
Epoch 70
Train accuracy: 0.5263435751240629
Val accuracy: 0.5204433497536946



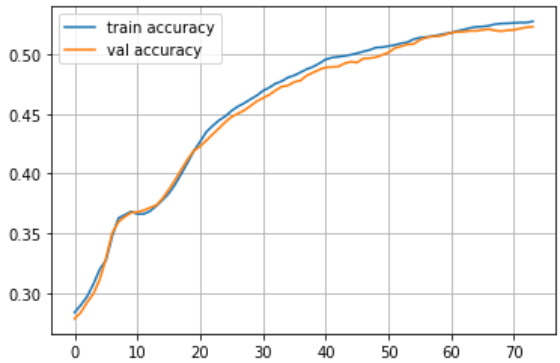
Epoch 71
Train accuracy: 0.5266603315383803
Val accuracy: 0.5216748768472906



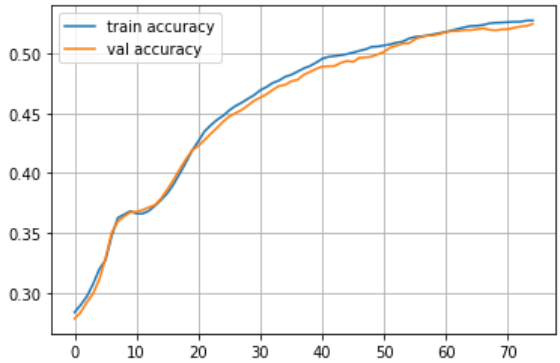
Epoch 72
Train accuracy: 0.5266603315383803
Val accuracy: 0.5226600985221674



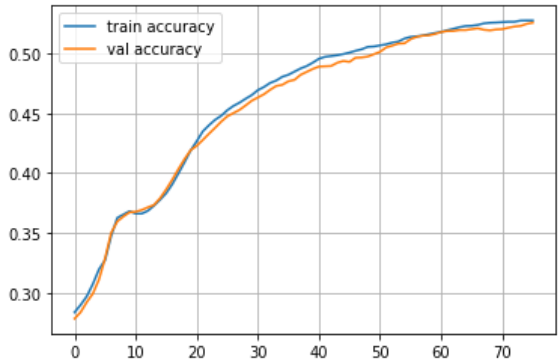
Epoch 73
Train accuracy: 0.5276106007813325
Val accuracy: 0.5231527093596059



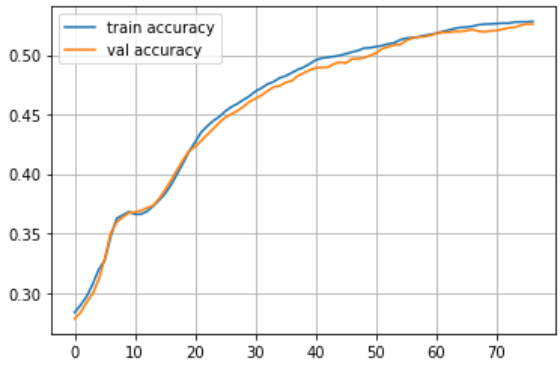
Epoch 74
Train accuracy: 0.5277161862527716
Val accuracy: 0.5248768472906404



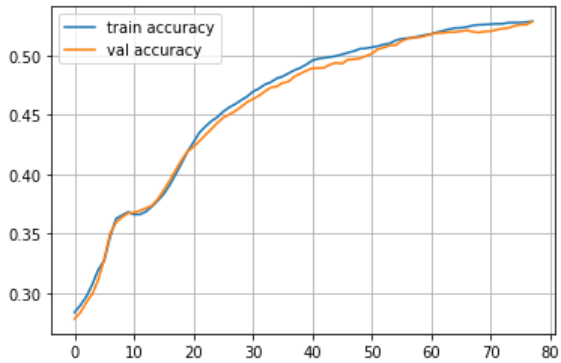
Epoch 75
Train accuracy: 0.5277161862527716
Val accuracy: 0.5258620689655172



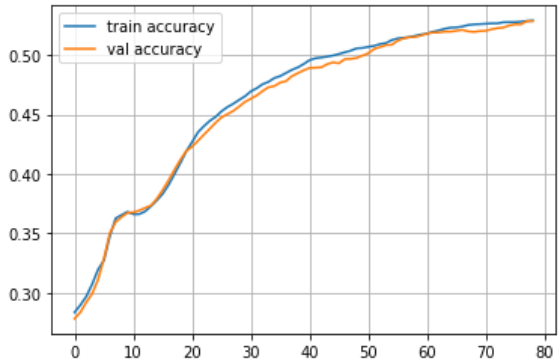
Epoch 76
Train accuracy: 0.5281385281385281
Val accuracy: 0.5258620689655172



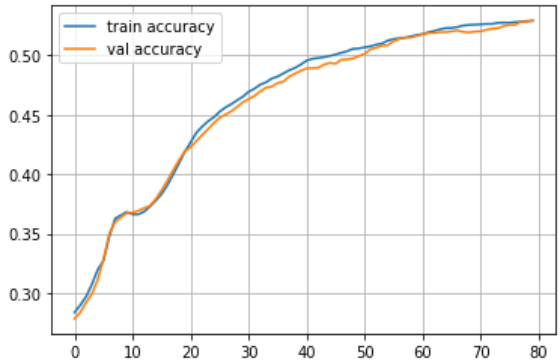
Epoch 77
Train accuracy: 0.5286664554957238
Val accuracy: 0.5285714285714286



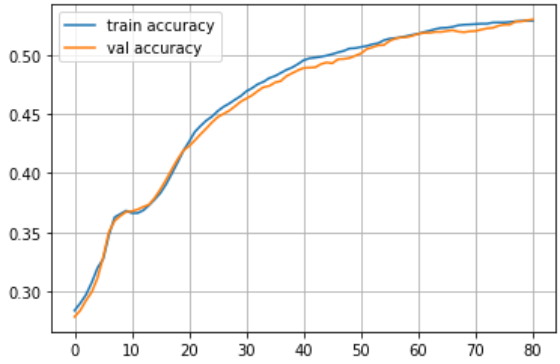
Epoch 78
Train accuracy: 0.5290887973814803
Val accuracy: 0.5285714285714286



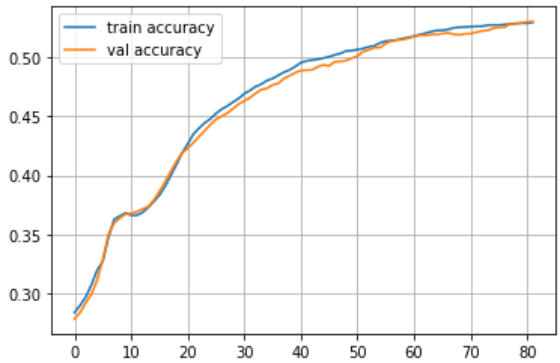
Epoch 79
Train accuracy: 0.5292999683243585
Val accuracy: 0.5295566502463054



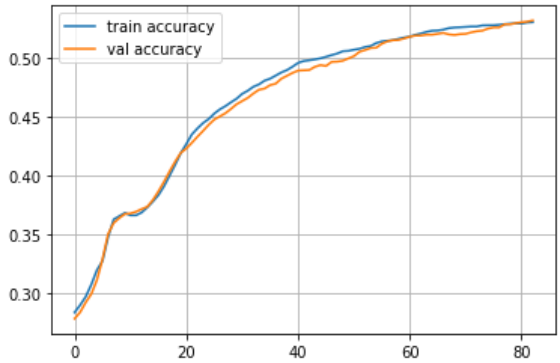
Epoch 80
Train accuracy: 0.5291943828529194
Val accuracy: 0.530295566502463



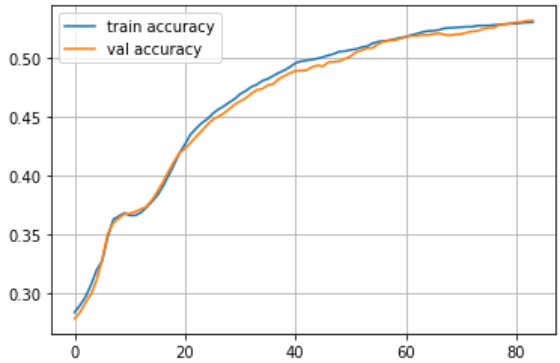
Epoch 81
Train accuracy: 0.5300390666244325
Val accuracy: 0.5307881773399015



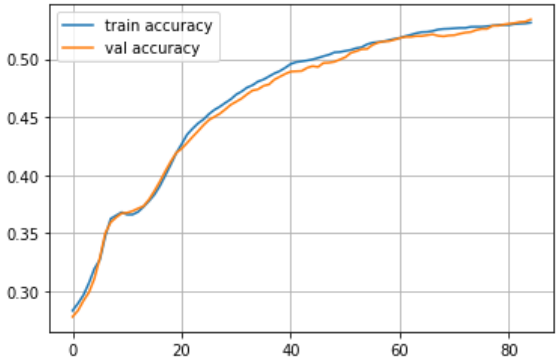
Epoch 82
Train accuracy: 0.5303558230387498
Val accuracy: 0.5317733990147783



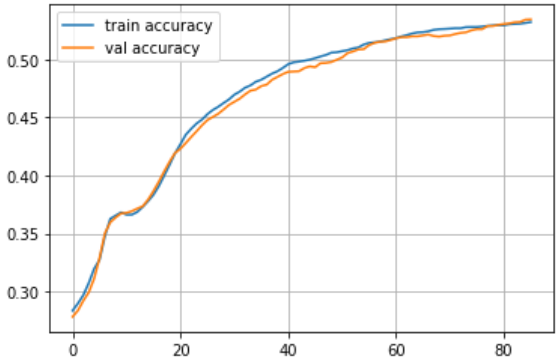
Epoch 83
Train accuracy: 0.5305669939816281
Val accuracy: 0.5320197044334976



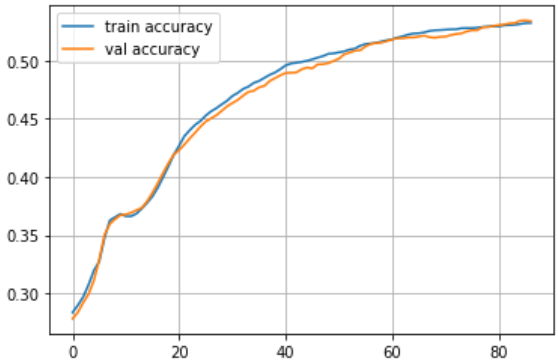
Epoch 84
Train accuracy: 0.5312005068102629
Val accuracy: 0.5339901477832513



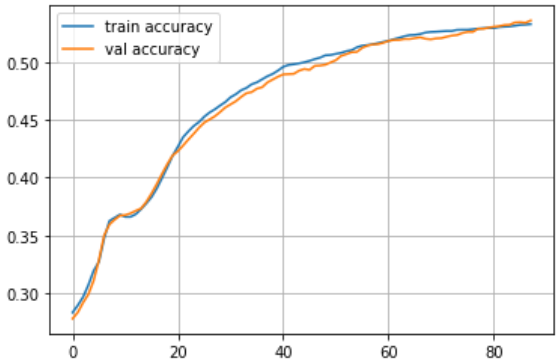
Epoch 85
Train accuracy: 0.5319396051103368
Val accuracy: 0.5342364532019704



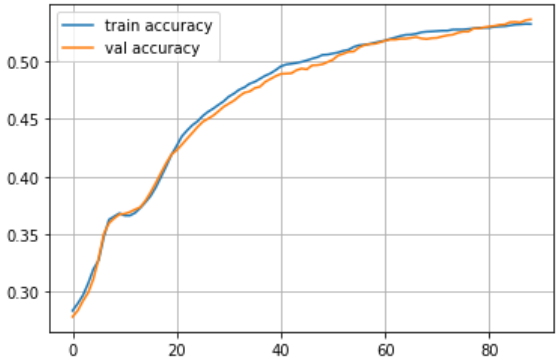
Epoch 86
Train accuracy: 0.532150776053215
Val accuracy: 0.533743842364532



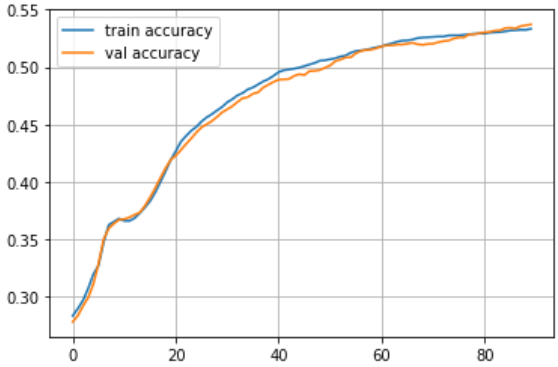
Epoch 87
Train accuracy: 0.5325731179389716
Val accuracy: 0.5357142857142857



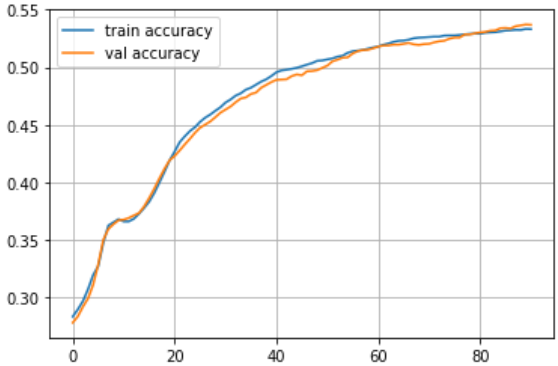
Epoch 88
Train accuracy: 0.5324675324675324
Val accuracy: 0.5364532019704433



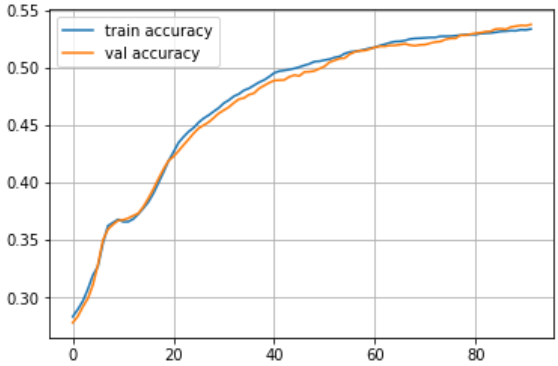
Epoch 89
Train accuracy: 0.5334178017104846
Val accuracy: 0.537192118226601



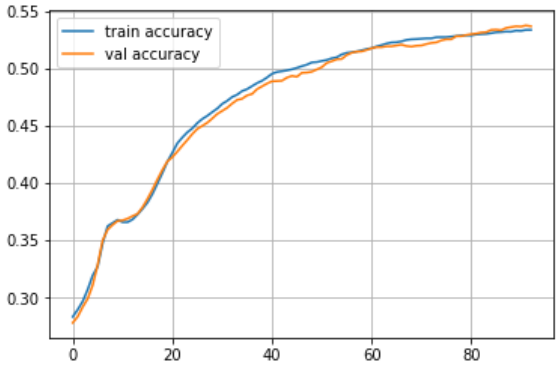
Epoch 90
Train accuracy: 0.5332066307676063
Val accuracy: 0.5369458128078818



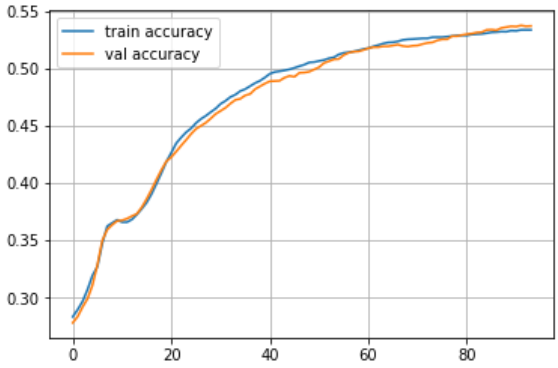
Epoch 91
Train accuracy: 0.5340513145391195
Val accuracy: 0.5379310344827586



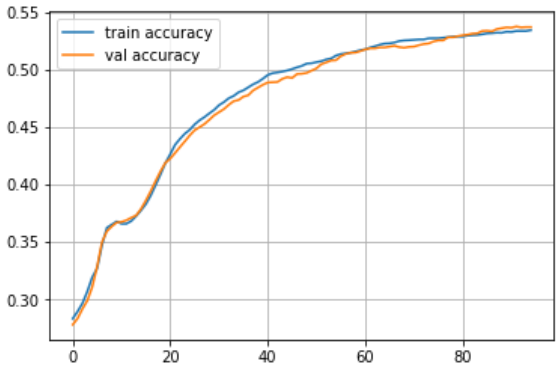
Epoch 92
Train accuracy: 0.5340513145391195
Val accuracy: 0.537192118226601



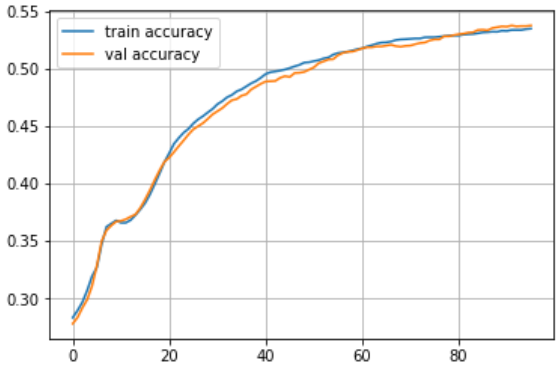
Epoch 93
Train accuracy: 0.5340513145391195
Val accuracy: 0.5374384236453202



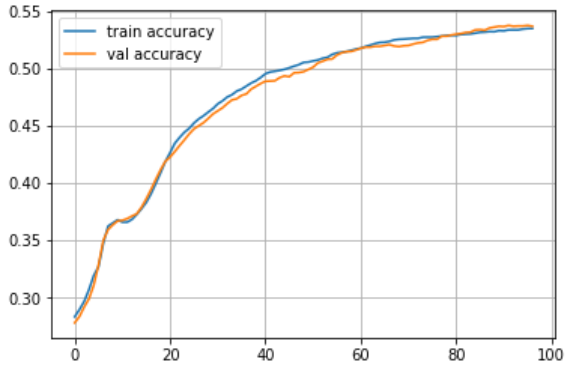
Epoch 94
Train accuracy: 0.5347904128391934
Val accuracy: 0.5374384236453202



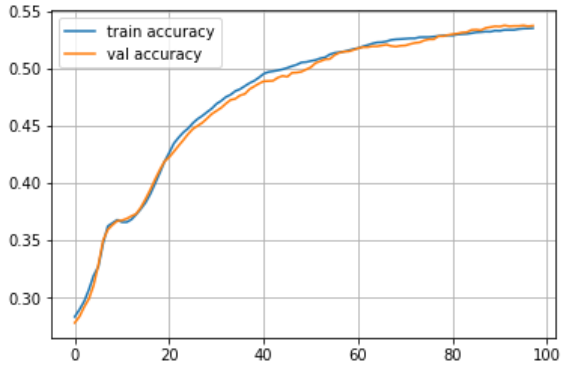
Epoch 95
Train accuracy: 0.5352127547249499
Val accuracy: 0.5379310344827586



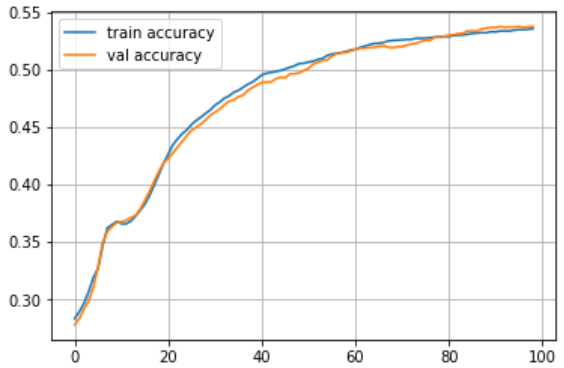
Epoch 96
Train accuracy: 0.535318340196389
Val accuracy: 0.537192118226601



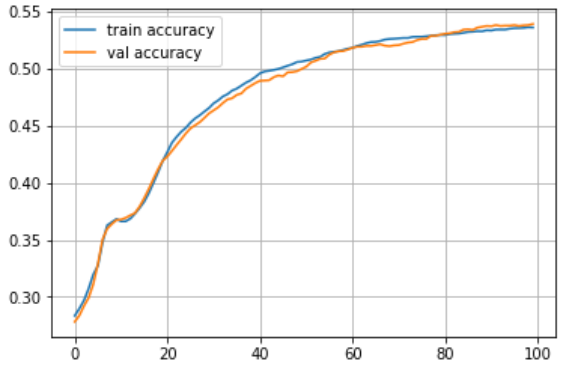

```
Epoch 97
Train accuracy: 0.5355295111392673
Val accuracy: 0.5376847290640394
```




```
Epoch 98
Train accuracy: 0.5360574384964629
Val accuracy: 0.5379310344827586
```



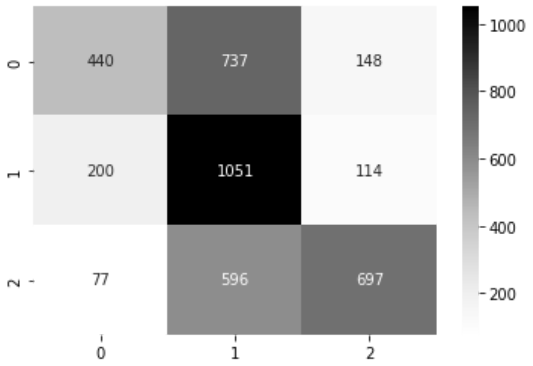

```
Epoch 99
Train accuracy: 0.5358462675535847
Val accuracy: 0.5389162561576355
```




```
CPU times: user 30 s, sys: 13.3 s, total: 43.2 s
Wall time: 30.9 s
```

In [ ]:
```
%%time
y_pred = network.predict(x_test)

evl = Evaluate(y_test, y_pred)
evl.get_evaluation_report()
```

```
precision: 0.5937741715871172
recall: 0.5369326552496915
f1: 0.5300576812388891
accuracy: 0.5389162561576355
Confusion matrix:
CPU times: user 156 ms, sys: 119 ms, total: 276 ms
Wall time: 147 ms
```



Έτοιμο Multi-Layer Perceptron

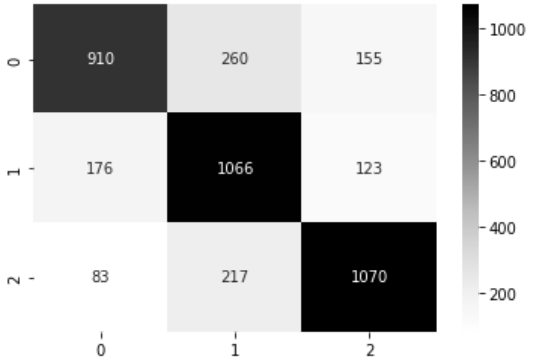
Όπως και με τις τεχνικές των παραπάνω ερωτημάτων έτσι και εδώ υπάρχει έτοιμη η παραπάνω κλάση σε διάφορες βιβλιοθήκες. Έτσι στο δεύτερο μέρος του ερωτήματος αυτού θα κατασκευάσετε το ίδιο MLP χρησιμοποιώντας όμως την έτοιμη κλάση MLPClassifier της βιβλιοθήκης scikit-learn. Παρακάτω παρουσιάζεται ένα παράδειγμα χρήσης της κλάσης αυτής.

In [ ]:
```
%%time
from sklearn.neural_network import MLPClassifier

epochs = 100
mlp = MLPClassifier(hidden_layer_sizes = (10, 15, 20), max_iter = epochs)

mlp.fit(x_train,y_train)
```

```
CPU times: user 4.29 s, sys: 4.18 ms, total: 4.29 s
Wall time: 4.3 s
/usr/local/lib/python3.7/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:696: ConvergenceWarning: Stochas
tic Optimizer: Maximum iterations (100) reached and the optimization hasn't converged yet.
  ConvergenceWarning,
```

In [ ]:
```
%%time
y_pred = mlp.predict(x_test)
evl = Evaluate(y_test, y_pred)
evl.get_evaluation_report()
```

```
precision: 0.7543578723309011
recall: 0.7495889105309296
f1: 0.7500816565393489
accuracy: 0.7502463054187192
Confusion matrix:
CPU times: user 156 ms, sys: 131 ms, total: 286 ms
Wall time: 153 ms
```



Σύγκριση υλοποιήσεων

Η διαφορά των αποτελεσμάτων προκύπτει από το ότι στην έτοιμη κλάση έχουν γίνει αρκετές βελτιστοποιήσεις στην λειτουργία, στον τρόπο εκπαίδευσης κ.λ.π., με αποτέλεσμα να προκύπτουν καλύτερα αποτέλεσματα. Παρ' όλα αυτά στην δική μας κλάση

έχουμε καλύτερο έλεγχο και έχουμε την δυνατότητα να σχεδιάσουμε πιο σύνθετες αρχιτεκτονικές, καθώς να αλλάξουμε τις τιμές παραμέτρων, που στην έτοιμη κλάση μπορεί να μην μας δίνεται η δυνατότητα.

Η έτοιμη υλοποίηση, παρ' ότι δεν συγκλίνει ύστερα από 100 epochs, παρουσιάζει ήδη πολύ καλύτερα αποτελέσματα, με ίδιες διαστάσεις επιπέδων φυσικά, ιδίως για την κλάση 1. Σχετικά με την δική μας υλοποίηση, παρατηρούμε ότι η ακρίβεια μεγαλώνει σταδιακά σε κάθε epoch, επομένως αύξηση αυτών ίσως να επέφερε βελτίωση. Άξιο παρατήρησης είναι ότι πάλι τα περισσότερα σφάλματα ταξινόμησης γίνονται σε δείγματα των κλάσεων 0 και 2 που σημαίνονται εσφαλμένα ως μέλη της 1. Ο χρόνος εκπαίδευσης παρουσιάζει πάλι αισθητή διαφορά, αν και όχι τόσο μεγάλη όσο στην περίπτωση του kNN, με την υλοποίηση του sklearn να υπερτερεί.

Αξιολόγηση- Συμπεράσματα

Τέλος, στο σημείο αυτό καλείστε να αξιολογήσετε τις διάφορες τεχνικές ταξινόμησης (KNN, Naive Bayes, MLP), τα αποτελέσματά τους, τους χρόνους εκτέλεσης, και να παραθέσετε παρατηρήσεις καθώς και οτιδήποτε σας φάνηκε ενδιαφέρον ή ιδιαίτερο.

Συμπεραίνουμε ότι, για το συγκεκριμένο πρόβλημα ταξινόμησης, η καλύτερη επιλογή είναι **ο ταξινομητής MLP του sklearn**, με ακρίβεια και f1 περίπου 75%, το υψηλότερο που παρατηρήσαμε. Ως προς τους χρόνους εκπαίδευσης και εξαγωγής προβλέψεων, ο ταχύτερος είναι ο Naive Bayes, με αρκετά χαμηλότερη, ωστόσο, ακρίβεια.

Classifier	Accuracy	F1 Score	Train time [s]
kNN - custom	0.69	0.69	312
kNN - sklearn	0.69	0.69	0.794
NB - custom	0.66	0.66	1.04
GNB - sklearn	0.69	0.68	0.00538
MLP - custom	0.54	0.53	30
MLP - sklearn	0.75	0.75	4.3