Αναφορά 1ης σειράς ασκήσεων:

“Μελέτη μεθόδων ταξινόμησης δεδομένων (Data Classification algorithms)”

Μάθημα: “MYE002 - Μηχανική Μάθηση”

Ακαδημαϊκό έτος: 2021-2022

Πανεπιστήμιο Ιωαννίνων

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Ομάδα 76:

Μπουλώτης Παναγιώτης AM: 4271

Ζέζος Αργύριος AM: 4588

Εισαγωγή

Στόχος της εργασίας είναι η ταξινόμηση των δεδομένων που μας δίνονται από τον διαγωνισμό του Kaggle, δηλαδή να βρούμε την κατηγορία (type) που λείπει από το σύνολο του test . Προκειμένου να γίνει αυτό, πρέπει να έχουμε τα δύο σύνολα δεδομένων, το **"train.csv"** και το **"test.csv"** στο ίδιο path με τον κώδικα που υλοποιήσαμε (σε python) ώστε να μπορεί να τα επεξεργαστεί.

Για τις απαιτήσεις της εργασίας, χρησιμοποιήθηκαν οι παρακάτω βιβλιοθήκες:

```
Import pandas as pd  # for csv reading
import numpy as np   # for csv writing
import statistics as st  # for mean() and stdev()
from math import exp, sqrt, pi  # for the normal pdf
from keras.models import Sequential
from keras.layers import Dense
from sklearn.neighbors import KNeighborsClassifier
from sklearn import svm  # Import svm model
from sklearn.model_selection import GridSearchCV
```

όπου:

- **pandas(pd):** για την επεξεργασία δεδομένων.
- **numpy(np):** για την γραφή δεδομένων σε .csv αρχείο.
- **statistics(st):** για την μέση τιμή και την τυπική απόκλιση.
- **math:** για χρήσιμες μαθηματικές εξισώσεις.
- **KNeighborsClassifier από την sklearn:** για την υλοποίηση την μεθόδου K Nearest Neighbor.
- **Sequential & Dense από το keras του tensorflow:** για την δημιουργία του νευρωνικού δικτύου. Σημείωση: Μπορεί εξαιτίας της βιβλιοθήκης αυτής, η εκτέλεση του προγράμματος να εμφανίσει κάποια warnings που μπορούμε να τα αγνοήσουμε
- **svm, GridSearchCV από την sklearn:** για την υλοποίηση την μεθόδου Support Vector Machine.

Προτείνουμε πως για την εκτέλεση κάθε μεθόδου (δηλαδή knn_execute(), svm_execute(), neural_execute() και naïve_bayes()) να μπουν οι αντίστοιχες των υπόλοιπων σε σχόλια για την πιο εύκολη και γρήγορη εκτέλεση.

Επεξεργασία δεδομένων

Η βιβλιοθήκη pandas είναι ένα πολύ χρήσιμο εργαλείο για την εύκολη επεξεργασία δεδομένων, ειδικά αν είναι της μορφής ".csv" . Πιο συγκεκριμένα, με την συνάρτηση **read_csv()** μπορούμε να αποθηκεύσουμε το σύνολο δεδομένων στην μεταβλητή ***train*** που είναι της μορφής:

	id	bone_length	rotting_flesh	hair_length	has_soul	color	type
0	0	0.354512	0.350839	0.465761	0.781142	clear	Ghoul
1	1	0.575560	0.425868	0.531401	0.439899	green	Goblin
2	2	0.467875	0.354330	0.811616	0.791225	black	Ghoul
3	4	0.776652	0.508723	0.636766	0.884464	black	Ghoul
4	5	0.566117	0.875862	0.418594	0.636438	green	Ghost

Και ύστερα την μετατρέπουμε σε μορφή DataFrame, στην ***train_data***, ώστε να μπορέσουμε να διαχωρίσουμε τα δεδομένα της.

Ο διαχωρισμός των δεδομένων σε x_train και y_train είναι απαραίτητος για την εκτέλεση των αλγορίθμων ταξινόμησης. Από τα δεδομένα που μας δίνουμε καταλαβαίνουμε πως οι τιμές του x_train είναι αυτές των στηλών ["bone_length", "rotting_flesh" "hair_length" "has_soul" "color"], ενώ η y_train έχει μόνο τις τιμές της στήλης "type".

Για να ξεχωρίσουμε τα δεδομένα του x_train, χρησιμοποιούμε το παρακάτω:

```
x_train = train_data.iloc[:, 1:6].values  # we don't want the id and type values
```

Δηλαδή στην .iloc δίνουμε ως όρισμα όλες τις γραμμές και τις στήλες 1-6 και με την .values παίρνουμε μόνο τιμές και όχι και την $1^{η}$ γραμμή που έχει την ετικέτα των στηλών.

Επειδή όμως η στήλη color έχει τιμές τύπου string θέλουμε να τις μετατρέψουμε σε τιμές στο διάστημα [0,1]. Επομένως κάνουμε την εξής μετατροπή:

color	clear	black	green	white	blood	blue
αντίστοιχη τιμή	1/6	2/6	0.5	4/6	5/6	1

Ο αντίστοιχος κώδικας:

```
train = pd.read_csv(r'train.csv')
# train data handling, splitting the columns to different lists
train_data = pd.DataFrame(train)
x_train = train_data.iloc[:, 1:6].values  # we don't want the id and type
                                                                    values
y_train = train_data['type'].values.tolist()
train_length = len(y_train)  # 371 rows


# convert the color column to numbers
for i in range(len(x_train)):
    if x_train[i][4] == 'clear':
        x_train[i][4] = 1/6
    elif x_train[i][4] == 'black':
        x_train[i][4] = 2/6
    elif x_train[i][4] == 'green':
        x_train[i][4] = 0.5
    elif x_train[i][4] == 'white':
        x_train[i][4] = 4/6
    elif x_train[i][4] == 'blood':
        x_train[i][4] = 5/6
    elif x_train[i][4] == 'blue':
        x_train[i][4] = 1
```

Με όμοιο τρόπο χειριζόμαστε το test σύνολο, αλλά αυτή τη φορά δεν έχουμε y_test από τα δεδομένα, καθώς αυτό θα είναι το αποτέλεσμα των αλγορίθμων. Για να κάνουμε submit το αποτέλεσμα στην Kaggle, πρέπει να δημιουργήσουμε ένα .csv που θα είναι της μορφής [“id”, “type”] (***results***) επομένως θα κρατήσουμε τα δεδομένα στην ***test_id***.

Ο αντίστοιχος κώδικας:

```
# same for test.csv
test = pd.read_csv(r'test.csv')
test_data = pd.DataFrame(test)
test_id = test_data['id'].values.tolist()
x_test = test_data.iloc[:, 1:6].values
test_length = len(test_id)  # 529 rows

# convert the color column to numbers
for i in range(len(x_test)):
    if x_test[i][4] == 'clear':
        x_test[i][4] = 1/6
    elif x_test[i][4] == 'black':
        x_test[i][4] = 2/6
    elif x_test[i][4] == 'green':
        x_test[i][4] = 0.5
    elif x_test[i][4] == 'white':
        x_test[i][4] = 4/6
    elif x_test[i][4] == 'blood':
        x_test[i][4] = 5/6
    elif x_test[i][4] == 'blue':
        x_test[i][4] = 1
```

1)K Nearest Neighbor με Ευκλείδια απόσταση:

Την μέθοδο αυτή την υλοποιούμε στην αντίστοιχη συνάρτηση **knn(k)** όπου k ο αριθμός γειτόνων. Για την υλοποίηση της χρησιμοποιήσαμε την συνάρτηση **KNeighborsClasssifier**() της βιβλιοθήκης sklearn. Σε αυτή δώσαμε ως ορίσματα, n_neighbors=k, δηλαδή ο αριθμός γειτόνων να είναι ίσος με τον k και στο metric='euclidean' γιατί θέλουμε να χρησιμοποιεί την Ευκλείδια απόσταση. Ως αποτέλεσμα της συνάρτησης δημιουργείται ο **classifier** (ταξινομητής). Αυτός στη συνέχεια δέχεται τα στοιχεία εκπαίδευσης (x_train, y_train) και με το σύνολο στοιχείων ελέγχου (x_test) βρίσκει μια πρόβλεψη της ζητούμενης στήλης (y_pred) με τις συναρτήσεις **fit()** και **predict()** αντίστοιχα.

```
def knn(k):
    # KNearestNeighbours with the use of the euclidean distance
    classifier = KNeighborsClassifier(n_neighbors=k, metric='euclidean')
    classifier.fit(x_train, y_train)  # the classifier uses the train data
                                        needed
    y_pred = classifier.predict(x_test)  # y_prediction is the missing type
        of the test samples
```

Προκειμένου να εκτιμήσουμε την ακρίβεια του αποτελέσματος, ορίζουμε μια μεταβλητή **knn_results**. Αυτή αρχικοποιείται με τις στήλες ετικέτες για τα 'id' και 'type' στοιχεία που χρειάζεται το Kaggle. Στη συνέχεια, γεμίζουμε τον πίνακα αυτό με τις τιμές τους, όπου για τα id χρησιμοποιούμε το test_id (δηλαδή την πρώτη στήλη του test.csv) και για τα type τα στοιχεία του y_pred που παράγει η μέθοδος.

```
# for the Kaggle accuracy
knn_results = [['id', 'type']]  # the required labels
for j in range(len(y_pred)):
    knn_results.append([test_id[j], y_pred[j]])
return knn_results
```

Προκειμένου να εκτελέσουμε τις 4 περιπτώσεις και να δημιουργήσουμε τα κατάλληλα έγραφα .csv για να υπολογίσουμε την ακρίβεια μέσω του διαγωνισμού στο Kaggle, δημιουργήσαμε την συνάρτηση **knn_execute()**.

```
def knn_execute():
    results1 = knn(1)  # k=1
    np.savetxt("knn1.csv", results1, delimiter=",", fmt='% s')

    results2 = knn(3)  # k=3
    np.savetxt("knn3.csv", results2, delimiter=",", fmt='% s')

    results3 = knn(5)  # k=5
    np.savetxt("knn5.csv", results3, delimiter=",", fmt='% s')

    results4 = knn(10)  # k=10
    np.savetxt("knn10.csv", results4, delimiter=",", fmt='% s')
```

2)Νευρωνικό Δίκτυο

Έχουμε δύο διαφορετικές μορφές νευρωνικών δικτύων, το ένα με ένα κρυμμένο επίπεδο και το άλλο με δύο, που υλοποιούνται στα **mlp1()** και **mlp2()** αντίστοιχα.

Η mlp1() λαμβάνει ως όρισμα το k , που είναι ο αριθμός νευρώνων στο μοναδικό κρυμμένο επίπεδο και το epochs, το πλήθος εποχών του αλγορίθμου, το οποίο παίζει σημαντικό ρόλο στην ακρίβεια της μεθόδου.

Για να δημιουργήσουμε το δίκτυο (**network**), αρχικά χρησιμοποιούμε τη συνάρτηση **Sequential()**, ώστε να του αναθέσουμε κάθε επίπεδο στη σειρά (sequence) με τη συνάρτηση **Dense()**. Για το επίπεδο εισόδου προσθέτουμε το όρισμα *input_dim=5* στη δημιουργία του κρυμμένου επίπεδου. Του δίνουμε αυτό τον αριθμό καθώς το x_train περιέχει 5 διαφορετικά χαρακτηριστικά (5 διαφορετικές στήλες).

Το κρυμμένο επίπεδο προστίθεται στο δίκτυο με την χρήση της Dense() και της **add()**, όπου η Dense πέρα από το όρισμα για το επίπεδο εισόδου, λαμβάνει το όρισμα *k* για το πλήθος των νευρώνων του και το *activation='sigmoid'*, καθώς χρησιμοποιούμε την σιγμοειδή συνάρτηση ως συνάρτηση ενεργοποίησης.

Το επίπεδο εξόδου προστίθεται με όμοιο τρόπο, αλλά αυτή τη φορά έχει ως ορίσματα το 3, δηλαδή ο βαθμός των κατηγοριών άρα και των νευρώνων του και ως συνάρτηση ενεργοποίησης, την *softmax*.

Για την σύνθεση του δικτύου, χρησιμοποιούμε την **compile()** με ορίσματα *optimizer = 'sqd'* για να έχουμε την Stochastic Gradient Descent ως μέθοδο βελτιστοποίησης και *loss='sparse_categorical_crossentropy'*. Η χρήση του συγκεκριμένου ορίσματος είναι απαραίτητη για το συγκεκριμένο πρόβλημα ταξινόμησης, καθώς το categorical_crossentropy ανταποκρίνεται σε προβλήματα ταξινόμησης πολλών κλάσεων και το sparse στην αρχή χρησιμοποιείται για την αντιμετώπιση error

```
def mlp1(k, epochs):
    global x_train, y_train, x_test
    # We need to create the mlp network
    network = Sequential()
    # the input variables are 5, the number of columns
    # the hidden layer has K neurons uses sigmoid activation function
    network.add(Dense(k, input_dim=5, activation='sigmoid'))
    network.add(Dense(3, activation='softmax'))  # the output layer has 3
                                                 neurons, as many as the classes
    network.compile(optimizer='sgd', loss='sparse_categorical_crossentropy')
```

Ομοίως με την προηγούμενη μέθοδο, χρησιμοποιούμε την fit() για να τροφοδοτήσουμε στο δίκτυο τα στοιχεία εκπαίδευσης, με επιπλέον όρισμα το *epochs=epochs* για το πλήθος εποχών και την predict() για να βρούμε την τοπική μεταβλητή **temp_y_pred**. Λόγω της χρήσης της συνάρτησης softmax στην έξοδο του δικτύου, ως αποτελέσματα, η μεταβλητή αυτή έχει τις πιθανότητες το κάθε στοιχείο της test να ανήκει σε κάποια από τις 3 κλάσεις. Επομένως στην αντίστοιχη y_pred της μεθόδου αναθέτουμε την σωστή τιμή ανάλογα της στήλης που έχει την μεγαλύτερη τιμή πιθανότητας. Αν δηλαδή στην j-οστή γραμμή η μεγαλύτερη τιμή πιθανότητας ήταν στην $1^{η}$ στήλη, αναθέτεται στην y_pred[j] η 'Ghost', για την $2^{η}$ στήλη, η 'Ghoul' και για την $3^{η}$ η 'Goblin'.

```
network.fit(x=x_train, y=y_train, epochs=epochs)
temp_y_pred = network.predict(x_test)
y_pred = []
for j in range(len(temp_y_pred)):
    if temp_y_pred[j][0] == max(temp_y_pred[j]):
        y_pred.append('Ghost')
    elif temp_y_pred[j][1] == max(temp_y_pred[j]):
        y_pred.append('Ghoul')
    elif temp_y_pred[j][2] == max(temp_y_pred[j]):
        y_pred.append('Goblin')
```

Τελικά, δημιουργούμε το αντίστοιχο **mpl1_results** για τον υπολογισμό της ακρίβειας της μεθόδου.

```
# for the Kaggle accuracy
mlp1_results = [['id', 'type']]
for j in range(len(y_pred)):
    mlp1_results.append([test_id[j], y_pred[j]])
return mlp1_results
```

Με όμοιο τρόπο δημιουργούμε το νευρωνικό δίκτυο με δύο κρυμμένα επίπεδα, το **mlp2()** στο οποίο απλώς χρησιμοποιούμε τα ορίσματα k1 και k2 ως ορίσματα στις Dense() συναρτήσεις που τα δημιουργούν.

```
def mlp2(k1, k2, epochs):
    global x_train, y_train, x_test
    # We need to create the mlp network
    network = Sequential()
    network.add(Dense(k1, input_dim=5, activation='sigmoid'))  # the 1st
                                                hidden layer with K1 neurons
    network.add(Dense(k2, activation='sigmoid'))  # the 2nd hidden layer with
                                                    K2 neurons
    network.add(Dense(3, activation='softmax'))
    network.compile(optimizer='sgd', loss='sparse_categorical_crossentropy')

    network.fit(x=x_train, y=y_train, epochs=epochs)
    temp_y_pred = network.predict(x_test)
    y_pred = []
    for j in range(len(temp_y_pred)):
        if temp_y_pred[j][0] == max(temp_y_pred[j]):
            y_pred.append('Ghost')
        elif temp_y_pred[j][1] == max(temp_y_pred[j]):
            y_pred.append('Ghoul')
        elif temp_y_pred[j][2] == max(temp_y_pred[j]):
            y_pred.append('Goblin')

    # for the Kaggle accuracy
    mlp2_results = [['id', 'type']]
    for j in range(len(y_pred)):
        mlp2_results.append([test_id[j], y_pred[j]])
    return mlp2_results
```

Για την εκτέλεση των πειραμάτων δημιουργήθηκε η συνάρτηση **neural_execute()**. Πριν την εκτέλεση τους τροποποιούμε την y_train ώστε να έχεις τιμές 0 για Ghost, 1 για Ghoul και 2 για Goblin. Αυτό, όπως και η μετατροπή των x_train και x_test σε λίστες, γίνεται για λόγους συμβατότητας. Λόγω της στοχαστικής φύσης της μεθόδου, παρατηρήσαμε μια αρκετά διαφορετική τιμή στην ακρίβεια της σε κάθε εκτέλεση, οπότε εκτελέσαμε το κάθε πείραμα 3 φορές και υπολογίσαμε τον μέσο όρο.

```
# for compatibility reasons assign an int value to the string type classes
for j in range(len(y_train)):
    if y_train[j] == 'Ghost':
        y_train[j] = 0
    elif y_train[j] == 'Ghoul':
        y_train[j] = 1
    else:
        y_train[j] = 2

x_train = x_train.tolist()
x_test = x_test.tolist()
```

```
def neural_execute():
    # MLP with 1 Hidden Layer
    results1 = mlp1(50, 350)  # k=50
    np.savetxt("mlp1_50.csv", results1, delimiter=",", fmt='% s')
    results2 = mlp1(100, 350)  # k=100
    np.savetxt("mlp1_100.csv", results2, delimiter=",", fmt='% s')
    results3 = mlp1(200, 400)  # k=200
    np.savetxt("mlp1_200.csv", results3, delimiter=",", fmt='% s')

    # MLP with 2 Hidden Layers
    results = mlp2(50, 25, 700)  # k1=50,k2=25
    np.savetxt("mlp2_50,25.csv", results, delimiter=",", fmt='% s')
    results = mlp2(100, 50, 700)  # k1=100,k2=50
    np.savetxt("mlp2_100,50.csv", results, delimiter=",", fmt='% s')
    results = mlp2(200, 100, 700)  # k1=200,k2=100
    np.savetxt("mlp2_200,100.csv", results, delimiter=",", fmt='% s')
```

Ύστερα από διάφορες δοκιμές στην τιμή της εποχής, αποφασίσαμε τα παρακάτω αποτελέσματα ως καλύτερα.

Για ένα κρυμμένο επίπεδο:

Αριθμός νευρώνων	πλήθος εποχών	Ακρίβεια 1η εκτέλεσης	Ακρίβεια 2η εκτέλεσης	Ακρίβεια 3η εκτέλεσης	**Μ.Ο.**
k=50	350	0,70888	0,67485	0,68431	**0,689347**
k=100	350	0,724	0,69754	0,6862	**0,70258**
k=200	400	0,71644	0,73156	0,68052	**0,709507**

Για δύο κρυμμένα επίπεδα:

Αριθμός νευρώνων	πλήθος εποχών	Ακρίβεια 1η εκτέλεση	Ακρίβεια 2η εκτέλεσης	Ακρίβεια 3η εκτέλεσης	**Μ.Ο.**
k1=50 k2=25	700	0,60491	0,43478	0,44234	**0,49401**
k1=100 k2=50	700	0,67296	0,6257	0,64461	**0,647757**
k1=200 k2=100	700	0,68052	0,71644	0,54064	**0,645867**

3) Support Vector Machines (SVM):

Αρχικά περιλαμβάνουμε τις απαραίτητες βιβλιοθήκες όπως αυτή του προτύπου, μετρικών και αυτή για αυτοματοποιημένη εύρεση των κατάλληλων παραμέτρων του προτύπου.

Έπειτα δημιουργούμε τον εκτιμητή

Στη συνέχεια ορίζουμε τα διαστήματα στα οποία θα γίνει η αναζήτηση των βέλτιστων παραμέτρων.

Παράμετροι

C : Παράμετρος που συμβάλλει στη μεταβολή του περιθωρίου. Με μεγάλο περιθώριο παρατηρείται μεγαλύτερη ακρίβεια αλλά ίσως προκληθεί υπερεκπαίδευση. Αντιθέτως για μικρό περιθώριο.

Gamma : Παράμετρος που αφορά την καμπυλότητα του ορίου αποφάσεως.

Kernel : Παράμετρος που καθορίζει τη συνάρτηση πυρήνα (Γραμμική ή Gaussian)

Επιλέγουμε αυτόματα τις βέλτιστες τιμές των παραμέτρων μέσω της συνάρτησης **GridSearch()**, η οποία δέχεται ορίσματα:

- τον εκτιμητή (**ml**)
- το διάστημα παραμέτρων (**param_grid**)
- Επιλογή ή μη επανεκπαίδευσης του εκτιμητή με τις βέλτιστες παραμέτρους (**refit = True**)
- Παράμετρος φλυαρίας εξόδου (ελάχιστη δυνατή στην περίπτωσή μας – **verbose =1**)
- Ρύθμιση της στρατηγικής διασταύρωσης, (**cv = 15** – fold στην περίπτωσή μας)

```
def svm_execute(kernel):
    # Create a svm Classifier and hyper parameter tuning
    ml = svm.SVC()

    # defining parameter range
    param_grid = {'C': [1, 10, 100, 1000, 10000],
                  'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
                  'kernel': [kernel]}  # ['linear', 'rbf']}  linear or
gaussian

    grid = GridSearchCV(ml, param_grid, refit=True, verbose=1, cv=15)
```

Εκπαιδεύουμε το μοντέλο με τις βέλτιστες παραμέτρους μεσω της συνάρτησης fit με τα δεδομένα εκπαίδευσης.

Τυπώνουμε τις βέλτιστες παραμέτρους και την ακρίβειά τους και εκτελούμε τυπώνοντας την πρόβλεψη. Δημιουργούμε πίνακα αποτελεσμάτων συμβατό με την ιστοσελίδα kaggle και τον αποθηκεύουμε σε αρχείο κειμένου.

```
# fitting the model for grid search
grid_search = grid.fit(x_train, y_train)

print(grid_search.best_params_)

accuracy = grid_search.best_score_ * 100
print("Accuracy for our training dataset with tuning is :
{:.2f}%".format(accuracy))

y_pred = grid.predict(x_test)

results = [['id', 'type']]
for j in range(len(y_pred)):
    results.append([test_id[j], y_pred[j]])

return results
```

Η ακρίβεια του προτύπου μας όπως διαπιστώθηκε από την ιστοσελίδα kaggle είναι **0.74480** για την **rbf** και **0.72967** για την **linear**

4)Naïve Bayes:

Η μέθοδος αυτή βασίζεται πρωτίστως στην δεσμευμένη πιθανότητα με βάση τον κανόνα του Bayes.

Προκειμένου να την υλοποιήσουμε, πρέπει να υπολογίσουμε τις πιθανότητες αυτές με τα παρακάτω βήματα:

α)Ταξινόμηση του συνόλου εκπαίδευσης

Ανάλογα την τιμή του y_train, το οποίο πλέον έχει τιμές 0,1,2 για τιμές Ghost, Ghoul και Goblin αντίστοιχα, δημιουργούμε λίστες που έχουν τα στοιχεία του x_train ταξινομημένα ανά κλάση. Αυτό θα κάνει την διαχείριση της μεθόδου πιο βολική.

```
def naive_bayes():
    global train_stats, sorted_x_train
    # a) sort the x_train list depending on the class
    sorted0 = []
    sorted1 = []
    sorted2 = []
    for n in range(len(x_train)):
        if y_train[n] == 0:
            sorted0.append(x_train[n])
        elif y_train[n] == 1:
            sorted1.append(x_train[n])
        else:
            sorted2.append(x_train[n])

    sorted_x_train = sorted0 + sorted1 + sorted2
```

β)Αποθήκευση στατιστικών κάθε κλάσης

Αποθηκεύουμε την μέση τιμή και την τυπική απόκλιση των στοιχείων κάθε κλάσης στην λίστα **train_stats**, καθώς θα τα χρειαστούμε για την κανονική κατανομή.

```
# b) save the stats for each class
train_stats.append([(st.mean(column), st.stdev(column)) for column in
zip(*sorted0)])
train_stats.append([(st.mean(column), st.stdev(column)) for column in
zip(*sorted1)])
train_stats.append([(st.mean(column), st.stdev(column)) for column in
zip(*sorted2)])
```

γ)Υπολογισμός πιθανοτήτων

Υπολογίζουμε την πιθανότητα κάποιο στοιχείο να είναι στην κάθε κλάση ως αποτέλεσμα της διαίρεσης του πλήθους στοιχείων στη συγκεκριμένη κλάση με το συνολικό πλήθος στοιχείων (**train_length**).

```
# c) calculating probabilities
prob_class.append(len(sorted0)/train_length)  # probability a row being in
                                                                    class 0
prob_class.append(len(sorted1)/train_length)  # same for class 1
prob_class.append(len(sorted2)/train_length)  # same for class 2
```

Με παρόμοιο τρόπο βρίσκουμε την πιθανότητα κάθε στοιχείο της στήλης ‘color’ να είναι στοιχείο κάποιας συγκεκριμένης κλάσης:

```
# color probabilities for each class
# for class 0
for j in range(len(sorted0)):
    if sorted0[j][4] == 1/6:
        color_prob[0][0] += (1/len(sorted0))
    elif sorted0[j][4] == 2/6:
        color_prob[0][1] += (1/len(sorted0))
    elif sorted0[j][4] == 0.5:
        color_prob[0][2] += (1/len(sorted0))
    elif sorted0[j][4] == 4/6:
        color_prob[0][3] += (1/len(sorted0))
    elif sorted0[j][4] == 5/6:
        color_prob[0][4] += (1/len(sorted0))
    else:
        color_prob[0][5] += (1/len(sorted0))
```

```
# for class 1
for j in range(len(sorted1)):
    if sorted1[j][4] == 1 / 6:
        color_prob[1][0] += (1 / len(sorted1))
    elif sorted1[j][4] == 2 / 6:
        color_prob[1][1] += (1 / len(sorted1))
    elif sorted1[j][4] == 0.5:
        color_prob[1][2] += (1 / len(sorted1))
    elif sorted1[j][4] == 4 / 6:
        color_prob[1][3] += (1 / len(sorted1))
    elif sorted1[j][4] == 5 / 6:
        color_prob[1][4] += (1 / len(sorted1))
    else:
        color_prob[1][5] += (1 / len(sorted1))

# for class 2
for j in range(len(sorted2)):
    if sorted2[j][4] == 1 / 6:
        color_prob[2][0] += (1 / len(sorted2))
    elif sorted2[j][4] == 2 / 6:
        color_prob[2][1] += (1 / len(sorted2))
    elif sorted2[j][4] == 0.5:
        color_prob[2][2] += (1 / len(sorted2))
    elif sorted2[j][4] == 4 / 6:
        color_prob[2][3] += (1 / len(sorted2))
    elif sorted2[j][4] == 5 / 6:
        color_prob[2][4] += (1 / len(sorted2))
    else:
        color_prob[2][5] += (1 / len(sorted2))
```

Για την υπόλοιπη υλοποίηση της μεθόδου, δημιουργήθηκε η **normal_pdf()** ώστε να υπολογίζουμε τις πιθανότητες κανονικής κατανομής

```
def normal_pdf(x, mean, stdev):
    exponent = exp(-((x-mean)**2 / (2 * stdev**2)))
    return (1 / (sqrt(2 * pi) * stdev)) * exponent
```

Η οποία χρησιμοποιείται στην συνάρτηση **calculate_prob()** που είναι υπεύθυνη για τον υπολογισμό της πιθανότητας της γραμμής που δέχεται ως όρισμα να βρίσκεται σε κάθε κλάση μέσω της τοπικής λίστα που επιστρέφουμε την **probabilities**.

Αρχικά δηλαδή για κάθε κλάση αναθέτουμε την πιθανότητα ένα στοιχείο να είναι στοιχείο της και στη συνέχεια για κάθε χαρακτηριστικό της γραμμής πολλαπλασιάζουμε την πιθανότητα του στην τωρινή.

```
# Calculates the probabilities for the Naive Bayes method
def calculate_prob(row):
    global train_stats, prob_class
    probabilities = [0, 0, 0]
    for class_value in range(3):
        probabilities[class_value] = float(prob_class[class_value])  # class
                                                                      probability
        for column in range(len(row)):
            # measurements needed for normal_pdf
            x_value = row[column]
            mean = train_stats[class_value][column][0]
            stdev = train_stats[class_value][column][1]

            if column != 4:
                # using normal distribution
                probabilities[class_value] *= normal_pdf(x_value, mean,
stdev)
            else:
                probabilities[class_value] *= calculate_color_prob(x_value,
class_value)
    return probabilities
```

Σημείωση: για την περίπτωση του χαρακτηριστικού color, καλούμε την συνάρτηση **calculate_color_prob()** η οποία επιστρέφει την αντίστοιχη δεσμευμένη πιθανότητας της τιμής αυτής (x_value) στην συγκεκριμένη κλάση (class_value).

```
def calculate_color_prob(x_value, class_value):
    if x_value == 1/6:
        return color_prob[class_value][0]
    elif x_value == 2/6:
        return color_prob[class_value][1]
    elif x_value == 0.5:
        return color_prob[class_value][2]
    elif x_value == 4/6:
        return color_prob[class_value][3]
    elif x_value == 5/6:
        return color_prob[class_value][4]
    else:
        return color_prob[class_value][5]
```

Για να προβλέψουμε τις κλάσεις του συνόλου εκπαίδευσης, χρησιμοποιούμε την συνάρτηση **nb_predict()** δίνοντας ως όρισμα γραμμή του συνόλου.

```
# The winning class is predicted as the missing class
def nb_predict(row):
    probabilities = calculate_prob(row)
    best_prob = max(probabilities)
    if probabilities[0] == best_prob:
        return 'Ghost'
    elif probabilities[1] == best_prob:
        return 'Ghoul'
    else:
        return 'Goblin'
```

Τελικά, υπολογίζουμε με τον παραπάνω τρόπο την y_pred και αποθηκεύουμε τα αποτελέσματα στο “naïve_bayes.csv” αρχείο για μέτρηση ακρίβειας στο Kaggle, το οποίο για την Naïve Bayes μέθοδο παρουσιάζει ακρίβεια **0.72967**

```
# predictions using NB
y_pred = []
for row in x_test:
    y_pred.append(nb_predict(row))

results = [['id', 'type']]
for j in range(len(y_pred)):
    results.append([test_id[j], y_pred[j]])

np.savetxt("naive_bayes.csv", results, delimiter=",", fmt='% s')
```

Μετά την υλοποίηση των παραπάνω μεθόδων, παρατηρούμε πως στα δεδομένα μας η Support Vector Machines είναι η μέθοδος με την καλύτερη ακρίβεια όταν χρησιμοποιούμε την συνάρτηση πυρήνα rbf.