ΜΙΧΑΗΛ ΜΠΙΖΙΜΗΣ ΑΜ: **1115201500102**

ΚΩΝΣΤΑΝΤΙΝΤΟΣ ΤΣΙΑΡΑΣ ΑΜ: **1115201500165**

# ΤΕΧΝΙΚΕΣ ΕΞΟΡΥΞΗΣ ΔΕΔΟΜΕΝΩΝ

# ΕΡΓΑΣΙΑ 2η

# Classification report

**Κώδικας**

* To αρχείο visualization.py το οποίο πραγματοποιεί την οπτικοποίηση των 5 πρώτων διαδρομών (με διαφορετικά journeyPatternId) του train\_set.csv όπως ζητείται από το Ερώτημα 1.
* Το αρχείο dtw\_neighbours.py με την υλοποίηση του Ερωτήματος 2 (Α-1).
* Το αρχείο LCSS.py με την δικιά μας υλοποίηση του αλγορίθμου LCSS και το αρχείο lcss\_neighbours.py που κάνει χρήση αυτού για την υλοποίηση του Ερωτήματος 2 (Α-2).
* Το αρχείο KNN.py με την δικιά μας υλοποίηση του KNN.
* Τα αρχεία knn\_classification.py και knn\_crossval.py τα οποία παράγουν το ζητούμενο αρχείο “testSet\_JourneyPatternIDs.csv” και πραγματοποιούν 10-fold Cross Validation αντίστοιχα, όπως περιγράφεται στο Ερώτημα 3.
* Το αρχείο util.py όπου υλοποιούνται συναρτήσεις που χρησιμοποιούνται από πολλαπλά αρχεία όπως η *harvesineDist()* και η *plotMap()*.

Για όλα τα ερωτήματα, τα datasets θεωρούμε πως υπάρχουν σε έναν φάκελο “./datasets/” στον φάκελο του παραδοτέου.

**Ερώτημα 1**

**Οπτικοποίηση των Δεδομένων**

|  |  |  |
| --- | --- | --- |
| JP\_ID: 015B0001 | JP\_ID: 015B1001 | JP\_ID: 083A1001 |
| JP\_ID: 00160001 | JP\_ID: 00161001 |  |

**Ερώτημα 2 (Α-1)**

**Εύρεση κοντινότερων γειτόνων**

Λόγω ταχύτητας και επειδή χρησιμοποιείται και από το Ερώτημα 3, επιλέξαμε να χρησιμοποιήσουμε την έτοιμη βιβλιοθήκη fastdtw (<https://pypi.org/project/fastdtw/>).

Ο DTW κάνει «stretch» δυο χρονοσειρές (εδώ: διαδρομές) κατάλληλα και επιστρέφει την απόκλισή τους. Όσο μικρότερη είναι η απόκλιση αυτή τόσο πιο παρόμοιες είναι δύο χρονοσειρές.

Παρατηρούμε για όλα τα Test Trips του test\_set οι 5 κοντινότεροι γείτονες βάσει DTW προκύπτουν να έχουν αρκετά παρόμοιες διαδρομές και μάλιστα το ίδιο JPID μεταξύ τους, πράγμα λογικό αφού τέτοιες διαδρομές μπορούν να γίνουν «stretched» ώστε να έχουν μικρή έως ελάχιστη απόκλιση.

|  |  |  |
| --- | --- | --- |
| Test Trip 1  Δt = 67 sec | *Neighbor 1*  JP\_ID: 01501001  DTW: 0.0km | *Neighbor 2*  JP\_ID: 01501001  DTW: 3.5km |
| *Neighbor 3*  JP\_ID: 01501001  DTW: 3.8km | *Neighbor 4*  JP\_ID: 01501001  DTW: 4km | *Neighbor 5*  JP\_ID: 01501001  DTW: 4.1km |

|  |  |  |
| --- | --- | --- |
| Test Trip 2  Δt = 41 sec | *Neighbor 1*  JP\_ID: 01200001  DTW: 0.0km | *Neighbor 2*  JP\_ID: 01200001  DTW: 2.8km |
| *Neighbor 3*  JP\_ID: 01200001  DTW: 3.4km | *Neighbor 4*  JP\_ID: 01200001  DTW: 3.4km | *Neighbor 5*  JP\_ID: 01200001  DTW: 3.5km |

|  |  |  |
| --- | --- | --- |
| Test Trip 3  Δt = 51 sec | *Neighbor 1*  JP\_ID: 00791001  DTW: 0.0km | *Neighbor 2*  JP\_ID: 00791001  DTW: 4.7km |
| *Neighbor 3*  JP\_ID: 00791001  DTW: 4.8km | *Neighbor 4*  JP\_ID: 00791001  DTW: 4.8km | *Neighbor 5*  JP\_ID: 00791001  DTW: 4.9km |

|  |  |  |
| --- | --- | --- |
| Test Trip 4  Δt = 42 sec | *Neighbor 1*  JP\_ID: 00010002  DTW: 0.0km | *Neighbor 2*  JP\_ID: 00010002  DTW: 2.5km |
| *Neighbor 3*  JP\_ID: 00010002  DTW: 2.8km | *Neighbor 4*  JP\_ID: 00010002  DTW: 3.2km | *Neighbor 5*  JP\_ID: 00010002  DTW: 3.5km |

|  |  |  |
| --- | --- | --- |
| Test Trip 5  Δt = 40 sec | *Neighbor 1*  JP\_ID: 01300001  DTW: 0.0km | *Neighbor 2*  JP\_ID: 01300001  DTW: 4.3km |
| *Neighbor 3*  JP\_ID: 01300001  DTW: 4.5km | *Neighbor 4*  JP\_ID: 01300001  DTW: 4.6km | *Neighbor 5*  JP\_ID: 01300001  DTW: 4.7km |

**Ερώτημα 2 (Α-2)**

**Εύρεση κοντινότερων υποδιαδρομών**

Τον LCSS τον υλοποιήσαμε εμείς στο *LCSS.py* .

Ο LCSS επιστρέφει την μέγιστη κοινή υπακολουθία δύο χρονοσειρών (εδώ διαδρομών). Σημειώνεται ότι, στα πλαίσια της υλοποίησής μας, δύο σημεία στο χάρτη θεωρούνται «κοινά» εάν η Harversine απόσταση μεταξύ τους είναι μικρότερη ή ίση των 200m.

Εδώ παρατηρούμε ότι καθώς, σε αντίθεση με τον DTW, δεν ψάχνουμε ομοιότητα μεταξύ των διαδρομών στο σύνολό τους οι γείτονες που προκύπτουν για κάθε Test Trip του test\_set σε κάποιες περιπτώσεις είναι αισθητά διαφορετικά διαδρομές μεταξύ τους, σε ορισμένες περιπτώσεις ακόμα και με διαφορετικά JPID. Αυτό είναι αναμενόμενο, αφού το μόνο που μας ενδιαφέρει εδώ είναι το πλήθος των matching points των δύο διαδρομών.

|  |  |  |
| --- | --- | --- |
| Test Trip 1  Δt = 348 sec | *Neighbor 1*  JP\_ID: 040D1002  #Matching Points: 78 | *Neighbor 2*  JP\_ID: 040D1002  #Matching Points: 78 |
| *Neighbor 3*  JP\_ID: 040D1002  #Matching Points: 76 | *Neighbor 4*  JP\_ID: 040D1002  #Matching Points: 76 | *Neighbor 5*  JP\_ID: 040D1002  #Matching Points: 75 |

|  |  |  |
| --- | --- | --- |
| Test Trip 2  Δt = 351 sec | *Neighbor 1*  JP\_ID: 040D1002  #Matching Points: 82 | *Neighbor 2*  JP\_ID: 040D1002  #Matching Points: 78 |
| *Neighbor 3*  JP\_ID: 040D1002  #Matching Points: 75 | *Neighbor 4*  JP\_ID: 040D1002  #Matching Points: 74 | *Neighbor 5*  JP\_ID: 040D1002  #Matching Points: 73 |

|  |  |  |
| --- | --- | --- |
| Test Trip 3  Δt = 180 sec | *Neighbor 1*  JP\_ID: 01451001  #Matching Points: 40 | *Neighbor 2*  JP\_ID: 00790001  #Matching Points: 40 |
| *Neighbor 3*  JP\_ID: 067X0001  #Matching Points: 40 | *Neighbor 4*  JP\_ID: 079A0001  #Matching Points: 40 | *Neighbor 5*  JP\_ID: 01451008  #Matching Points: 40 |

|  |  |  |
| --- | --- | --- |
| Test Trip 4  Δt = 265 sec | *Neighbor 1*  JP\_ID: 00790001  #Matching Points: 59 | *Neighbor 2*  JP\_ID: 00790001  #Matching Points: 59 |
| *Neighbor 3*  JP\_ID: 00790001  #Matching Points: 59 | *Neighbor 4*  JP\_ID: 00790001  #Matching Points: 59 | *Neighbor 5*  JP\_ID: 00790001  #Matching Points: 59 |

|  |  |  |
| --- | --- | --- |
| Test Trip 5  Δt = 320 sec | *Neighbor 1*  JP\_ID: 01200001  #Matching Points: 73 | *Neighbor 2*  JP\_ID: 01200001  #Matching Points: 73 |
| *Neighbor 3*  JP\_ID: 01200002  #Matching Points: 72 | *Neighbor 4*  JP\_ID: 01200001  #Matching Points: 72 | *Neighbor 5*  JP\_ID: 01200001  #Matching Points: 71 |

**Ερώτημα 3**

**Κατηγοριοποίηση**

Για το Ερώτημα αυτό τροποποιήσαμε την δική μας υλοποίησή του KNN από την πρώτη εργασία με τις εξής διαφορές:

* Αντί για σημεία το *fit()* εφαρμόζεται πάνω σε χρονοσειρές (δηλαδή αποθηκεύουμε χρονοσειρές και όχι σημεία).
* Αντί για EuclideanDistance μεταξύ των σημείων γίνεται χρήση της απόστασης που επιστρέφει η fastdtw μεταξύ χρονοσειρών με παράμετρο τη HarversineDistance.
* Αντί για heap sorting για εύρεση των top-K αποστάσεων κρατάμε ανά πάσα στιγμή τις Κ μικρότερες αποστάσεις σε έναν πίνακα μεγέθους Κ (το οποίο έχει καλύτερη πολυπλοκότητα – *O(Κn)* έναντι *Ο(nlogn)*).

Σε αντίθεση με την προηγούμενη εργασία όπου χρησιμοποιούσαμε majority voting, εδώ χρησιμοποιήσαμε άλλο voting scheme. Συγκεκριμένα σκεφτήκαμε ότι η «ψήφος» του i-οστού γείτονα θα πρέπει να έχει μεγαλύτερη βαρύτητα από αυτή του i+1 γείτονα, αφού ο πρώτος είναι περισσότερο παρόμοιος με την Test χρονοσειρά.

<formulas>

Σε ό,τι αφορά το Cross Validation, προκειμένου να μπορέσουμε να εφαρμόσουμε Stratified K-fold (και συγκεκριμένα με *n\_splits=10*) ήταν απαραίτητο να παραλείψουμε από το train\_set τις διαδρομές εκείνες με journeyPatternId που εμφανίζονταν λιγότερες από 10 φορές. Αυτό δεν είναι παράλογο καθώς δεν θέλουμε να υπάρχει διαδρομή σε test fold με journeyPatternId που δεν εμφανίζεται (ή εμφανίζεται ελάχιστες φορές) στα (9) train folds διότι έτσι δεν εφικτή η ορθή πρόβλεψή της.

Θεωρούμε ότι το accuracy που λαμβάνουμε έτσι ανταποκρίνεται στην πραγματικότητα καθώς τα Test Trips που θα κληθεί να προβλέψει το πρόγραμμά μας θα έχουν journeyPatternIds τα οποία εμφανίζονται επαρκή αριθμό φορών (9) στο train\_set ώστε να έχει γίνει fit για αυτά ο KNN.

Με το preprocessing αυτό στο 10-fold Cross Validation πέτυχαμε **Accuracy: 0.985411** .