**Big Data Mining Techniques (M161)**

**Winter Semester 2024-2025**

**ΟΝ/ΜΟ: ΔΙΒΡΙΩΤΗΣ ΑΝΑΣΤΑΣΙΟΣ**

**ΑΜ: 7110132300212**

**ΔΠΜΣ Η/Α**

***PART 1:***

Σκοπός:

Κατηγοριοποίηση άρθρων με βάση τον τίτλο και το περιεχόμενο. Για την εκπαίδευση χρησιμοποιούμε δύο μοντέλα, Support Vector Machines (SVM) και Random Forest, τα οποία εκπαιδεύουμε με άρθρα των οποίων η κατηγορία είναι γνωστή.

Μορφή Δεδομένων:

- train_set.csv: Id, Title, Content, Label
- test_without_labels.csv: Id, Title, Content

Υλοποίηση:

- Preprocessing:
  Για τη βελτίωση της απόδοσης του μοντέλου μετατρέπουμε όλους τους χαρακτήρες σε πεζούς (lowercasing), αφαιρούμε σημεία στίξης και ειδικούς χαρακτήρες, αφαιρούμε τις αριθμητικές τιμές, συχνές λέξεις χωρίς ιδιαίτερη σημασία για την κατηγοριοποίηση όπως “the”, “and” κλπ. αφού πρώτα κάνουμε tokenize το κείμενο. Τέλος δίνουμε στον τίτλο πενταπλάσιο βάρος από το περιεχόμενο.

```python
import nltk
from nltk.corpus import stopwords

# Load stopwords from nltk and convert to set for fast lookup
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))
```

```python
import re

# Preprocessing function - Remove punctuation, numbers and stopwords
def preprocess_text(text):
    if pd.isna(text):   # Handle NaN cases
        return ''
    text = text.lower()
    text = re.sub(r'[^\w\s]', ' ', text)  # Remove punctuation
    text = re.sub(r'\d+', ' ', text)  # Remove numbers
    text = [word for word in text.split() if word not in stop_words]  # Remove stopwords
    text = ' '.join(text)
    return text
```

```python
# Combine 'Title' and 'Content' to one column (give Title 5 times more weight) - apply preprocessing
X_train = (5 * (train_data['Title'] + ' ') + train_data['Content']).apply(preprocess_text)
y_train = train_data['Label'].to_numpy()

X_test = (5 * (test_data['Title'] + ' ') + test_data['Content']).apply(preprocess_text)
```

- Vectorization:

  Μετατρέπουμε το κείμενο σε αριθμητική απεικόνιση (Bag of Words) καθώς τα μοντέλα Machine Learning δεν δουλεύουν σωστά και γρήγορα με κείμενο.

```
from sklearn.feature_extraction.text import CountVectorizer

# Numerical representation (bag of words)
vectorizer = CountVectorizer()
X_train_vectorized = vectorizer.fit_transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)
```

- Cross-Validation:

  Με το 5-Fold Cross-Validation χωρίζουμε το train_set σε 5 μέρη, και κάθε φορά παίρνουμε ένα διαφορετικό μέρος του οποίου προσπαθούμε να μαντέψουμε το label εκπαιδεύοντας το μοντέλο μας με τα υπόλοιπα τέσσερα μέρη.
  Με αυτό τον τρόπο μπορούμε να εκτιμήσουμε την απόδοση του μοντέλου μας αλλά και να ρυθμίσουμε τις παραμέτρους αποφεύγοντας προβλήματα όπως το overfitting.
  Η διαδικασία επαναλαμβάνεται για το SVM και για το Random Forest.

```
from sklearn.model_selection import cross_val_score



from sklearn.svm import SVC

# 5-Fold Cross-Validation SVM
svm = SVC()
svm_scores = cross_val_score(svm, X_train_vectorized, y_train, cv=5)
print('SVM Accuracy (5-Fold CV on Subset):', svm_scores.mean())
```

```
SVM Accuracy (5-Fold CV on Subset): 0.9430654578162029
```

```
from sklearn.ensemble import RandomForestClassifier

# 5-Fold Cross-Validation Random Forest
rf = RandomForestClassifier()
rf_scores = cross_val_score(rf, X_train_vectorized, y_train, cv=5)
print('Random Forest Accuracy (5-Fold CV on Subset):', rf_scores.mean())
```

```
Random Forest Accuracy (5-Fold CV on Subset): 0.9212487141643187
```

- Εκπαίδευση των μοντέλων:

  Τέλος, εκπαιδεύουμε και τα δύο μοντέλα και προβλέπουμε τα Labels στο test_without_labels.csv .

```
# Train with SVM
svm.fit(X_train_vectorized, y_train)

# Predict the labels for the test data
predictions = svm.predict(X_test_vectorized)

# Write result to file (for Kaggle)
output_df = pd.DataFrame({'Id': test_data['Id'], 'Predicted': predictions})
output_df.to_csv('testSet_categories_svm_prep.csv', index=False)
print("Output file 'testSet_categories_svm_prep.csv' created successfully.")
```

```
# Train with Random Forest
rf.fit(X_train_vectorized, y_train)

# Predict the labels for the test data
predictions = rf.predict(X_test_vectorized)

# Write result to file (for Kaggle)
output_df = pd.DataFrame({'Id': test_data['Id'], 'Predicted': predictions})
output_df.to_csv('testSet_categories_rf_prep.csv', index=False)
print("Output file 'testSet_categories_rf_prep.csv' created successfully.")
```

Επεξήγηση των μοντέλων:

- Το SVM είναι ένας supervised machine learning αλγόριθμος ο οποίος ταιριάζει κυρίως σε προβλήματα classification. Προσπαθεί να βρει το βέλτιστο υπερεπίπεδο σε ένα N-διάστατο χώρο για να χωρίζει τα δεδομένα σε διαφορετικές κλάσεις. Ο αλγόριθμος μεγιστοποιεί τα περιθώρια μεταξύ των κοντινότερων σημείων διαφορετικών κλάσεων.
- Το Random Forest είναι ένα σύνολο από πολλά decision trees που εκπαιδεύονται σε τυχαία δείγματα δεδομένων και χαρακτηριστικών. Η τελική απόφαση στη περίπτωση του classification προκύπτει από ψηφοφορία (αποτέλεσμα των περισσότερων δέντρων).

Δοκιμές:

- Και οι δύο αλγόριθμοι είχαν πολύ καλό accuracy ακόμα και χωρίς βάρος στον τίτλο, αλλά παρατηρήθηκε μικρή αύξηση στο accuracy όσο μεγαλώναμε το βάρος στο τίτλο(δοκιμές στο 20% του dataset).

```
#Cross-Validation SVM with no weight on title
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score

svm = SVC()
svm_scores = cross_val_score(svm, X_train_vectorized, y_train, cv=5)
print("SVM Accuracy (5-Fold CV on Subset):", svm_scores.mean())
```

✓ 20m 2.1s

```
SVM Accuracy (5-Fold CV on Subset): 0.9332706621763229
```

```
#Cross-Validation SVM with x3 weight on title
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score

svm = SVC()
svm_scores = cross_val_score(svm, X_train_vectorized, y_train, cv=5)
print("SVM Accuracy (5-Fold CV on Subset):", svm_scores.mean())
```

✓ 20m 42.6s

```
SVM Accuracy (5-Fold CV on Subset): 0.9430654578162029
```

```
#Cross-Validation SVM with x5 weight on title
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score

svm = SVC()
svm_scores = cross_val_score(svm, X_train_vectorized, y_train, cv=5)
print("SVM Accuracy (5-Fold CV on Subset):", svm_scores.mean())
```

✓ 19m 32.4s

```
SVM Accuracy (5-Fold CV on Subset): 0.9484323241529891
```

- Δοκιμάστηκε και ο linear kernel στο SVM αντί του RBF (Radial Basis Function - default) αλλά ο RBF είχε καλύτερα αποτελέσματα

Αποτελέσματα:

| Statistic Measure | SVM (BoW) | Random Forest (BoW) |
|---|---|---|
| Accuracy | 0.9675 | 0.9334 |

Σχολιασμός:

Αν και ο δύο αλγόριθμοι είχαν καλά αποτελέσματα, το SVM είχε σταθερά λίγο καλύτερο accuracy σε όλες τις δοκιμές. Επιπλέον παρατηρήθηκε ότι το Random Forest ήταν αρκετά πιο γρήγορο(807 min για το SVM και 80 min για το Random Forest στο cross validation και 230 min και 20 min για την εκπαίδευση των μοντέλων αντίστοιχα) .

```python
#Cross-Validation SVM
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score

svm = SVC()
svm_scores = cross_val_score(svm, X_train_vectorized, y_train, cv=5)
print("SVM Accuracy (5-Fold CV on Subset):", svm_scores.mean())
```

✓ 807m 6.4s

```
SVM Accuracy (5-Fold CV on Subset): 0.9675298537501676
```

```python
#Cross-Validation RF
rf = RandomForestClassifier()
rf_scores = cross_val_score(rf, X_train_vectorized, y_train, cv=5)
print("Random Forest Accuracy (5-Fold CV on Subset):", rf_scores.mean())
```

✓ 80m 11.0s

```
Random Forest Accuracy (5-Fold CV on Subset): 0.9334853973791315
```

```python
#Train with SVM
svm.fit(X_train_vectorized, y_train)
predictions = svm.predict(X_test_vectorized)  # Predict the labels for the test data
output_df = pd.DataFrame({'Id': test_data['Id'], 'Predicted': predictions})
output_df.to_csv('testSet_categories_prep10%.csv', index=False)
print("Output file 'testSet_categories.csv' created successfully.")
```

✓ 230m 52.5s

```
Output file 'testSet_categories.csv' created successfully.
```

```python
#Train with RF
rf.fit(X_train_vectorized, y_train)

predictions = rf.predict(X_test_vectorized) # Predict the labels for the test data

output_df = pd.DataFrame({'Id': test_data['Id'], 'Predicted': predictions})
output_df.to_csv('testSet_categoriesrf_prep10%.csv', index=False)
print("Output file 'testSet_categories.csv' created successfully.")
```

✓ 19m 53.6s

```
Output file 'testSet_categories.csv' created successfully.
```

***PART 2:***

Σκοπός:

Η επιτάχυνση της μεθόδου K-Nearest Neighbors χρησιμοποιώντας την τεχνική LSH.

Τι κερδίζουμε με το LSH:

Με το LSH μπορούμε να εντοπίσουμε ζευγάρια που είναι πιθανόν να μοιάζουν. Έτσι υπολογίζουμε την πραγματική ομοιότητα μόνο σε ζευγάρια όπου η αναμενόμενη ομοιότητά τους είναι πάνω από κάποιο όριο.

Υλοποίηση:

- Preprocessing:
  Για τη βελτίωση της απόδοσης του μοντέλου μετατρέπουμε όλους τους χαρακτήρες σε πεζούς (lowercasing), αφαιρούμε σημεία στίξης και ειδικούς χαρακτήρες, αφαιρούμε τις αριθμητικές τιμές, συχνές λέξεις χωρίς ιδιαίτερη σημασία για την κατηγοριοποίηση όπως “the”, “and” κλπ.

```
import nltk
from nltk.corpus import stopwords

# Load stopwords from nltk and convert to set for fast lookup
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))
```

```
import re

# Preprocessing function - Remove punctuation, numbers and stopwords
def preprocess_text(text):
    if pd.isna(text):   # Handle NaN cases
        return ''
    text = text.lower()
    text = re.sub(r'[^\w\s]', ' ', text)  # Remove punctuation
    text = re.sub(r'\d+', ' ', text)  # Remove numbers
    text = [word for word in text.split() if word not in stop_words]  # Remove stopwords
    text = ' '.join(text)
    return text
```

- Shingles:
  Δοκιμάσαμε να χωρίσουμε τα κείμενα σε shingles 1, 2, 3 και 5 λέξεων για να εξετάσουμε αν sets από shingles περισσότερων λέξεων στη σειρά διατηρούν περισσότερο context ώστε να βελτιώσουν την ακρίβεια της πρόβλεψης ομοιότητας.

```
# Create k-shingles to capture context inside given texts when comparing
def k_shingles(text, k=5):
    words = text.split(' ')
    if len(words) < k:
        return set([text]) # If text is too short, return as is
    return set([' '.join(words[i:i+k]) for i in range(len(words) - k + 1)])
```

Μετά την εκτέλεση του Brute-Force KNN αλγορίθμου για τις παραπάνω περιπτώσεις, παρατηρούμε ότι δε βοηθούν στο πρόβλημα του classification για το συγκεκριμένο dataset, αφού το καλύτερο accuracy στα predictions επιτεύχθηκε με shingles μίας λέξης (κοντινή απόδοση με τα shingles 2 λέξεων).

| | | |
|---|---|---|
| testSet_categories_brute_force_knn.csv<br>Complete · 3d ago · Brute-Force KNN (1-shingle) | 0.96556 |
| testSet_categories_brute_force_knn.csv<br>Complete · 3d ago · Brute-Force KNN (2-shingles) | 0.95408 |
| testSet_categories_brute_force_knn.csv<br>Complete · 3d ago · Brute-Force KNN (3-shingles) | 0.92451 |
| testSet_categories_brute_force_knn.csv<br>Complete · 3d ago · Brute-Force KNN (5-shingles) | 0.80317 |

- Vectorization:

  Για πιο γρήγορους υπολογισμούς κάνουμε vectorize τα set λέξεων που έχουν προκύψει μετά το αρχικό preprocessing (χρησιμοποιώντας τον CountVectorizer), αναπαριστώντας το σύνολο των εγγράφων ως ένα sparse array όπου κάθε γραμμή αντιστοιχεί σε ένα κείμενο και κάθε στήλη σε μία λέξη του λεξιλογίου (vocabulary) από όλες τις πιθανές λέξεις του dataset. Κάθε κελί του πίνακα έχει τιμές 1 ή 0 ανάλογα με το αν η αντίστοιχη λέξη του vocabulary υπάρχει ή όχι στο αντίστοιχο κείμενο.

```
from sklearn.feature_extraction.text import CountVectorizer

# binary=True means Binary representation: 1 if exists, 0 otherwise - for Jaccard Similarity calculation
vectorizer = CountVectorizer(binary=True, analyzer=lambda x: x)
X_train_vectorized = vectorizer.fit_transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)
```

- K-Nearest-Neighbors:

  Για κάθε έγγραφο στο test dataset υπολογίζουμε την ομοιότητα με όλα τα έγγραφα του train dataset με βάση το συντελεστή Jaccard

  $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$

  και κρατάμε τα k=7 έγγραφα με την μεγαλύτερη ομοιότητα.

  Στη συνέχεια μπορούμε να κάνουμε προβλέψεις με βάση την πλειοψηφία των Labels από τους 7 επιλεγμένους Nearest Neighbors.

  Σημείωση: Επειδή η αποθήκευση των αποτελεσμάτων σε συνδυασμό με το μεγάλο μέγεθος των datasets ξεπερνάει τα όρια μνήμης, αποθηκεύουμε τα αποτελέσματα σε ένα προσωρινό αρχείο.

```python
import time
import numpy as np

# Number of k-Nearest Neighbors
k = 7
brute_top_k_filename = 'brute_force_top_k.txt'

start_time = time.time()

# Precompute the word counts for each training document
train_word_counts = np.array(X_train_vectorized.sum(axis=1)).ravel()
num_tests = X_test_vectorized.shape[0]

# Open temporary file to write results (in order to avoid memory overflow)
with open(brute_top_k_filename, 'w') as file:
    for i in range(num_tests):
        # Get the batch's test rows
        test_rows = X_test_vectorized[i]

        test_word_counts = test_rows.sum()
        intersection = test_rows.dot(X_train_vectorized.T).toarray().ravel()

        # Union = train_tokens + test_tokens - intersection
        union = train_word_counts + test_word_counts - intersection

        # If union = 0, turn to 1 (avoid division by 0)
        union[union == 0] = 1

        # Compute Jaccard similarity for the current batch
        jaccard_similarities = intersection / union

        # Find the indices of the top K highest similarity values
        top_k_indices = np.argpartition(-jaccard_similarities, k)[:k]

        file.write(f'{top_k_indices.tolist()}\n')

        if i % 1000 == 0:
            file.flush()

query_time = time.time() - start_time
print(f'Brute-force Query Time: {query_time:.2f} seconds')
```

```
Brute-force Query Time: 10621.21 seconds
```

- Locality Sensitive Hashing:

Χρησιμοποιούμε τη μέθοδο Min-Hash LSH (υλοποίηση της βιβλιοθήκης datasketch) ώστε να μειώσουμε το χρόνο εκτέλεσης, περιορίζοντας το πλήθος των συγκρίσεων μόνο σε έγγραφα που είναι πιο πιθανό να είναι όμοια μεταξύ τους, σύμφωνα με τα signatures που παράγονται από τα hashes. Δημιουργούμε το LSH index από τα έγγραφα του train dataset και στη συνέχεια για κάθε έγγραφο στο test dataset κάνουμε query με βάση το Min-Hash του, ώστε να μας επιστρέψει τα υποψήφια όμοια έγγραφα (candidates). Έπειτα υπολογίζουμε αναλυτικά το Jaccard Similarity του εγγράφου με τα candidates που επέστρεψε το query και επιλέγουμε τα 7 πιο όμοια.

```python
from datasketch import MinHash, MinHashLSH

# Function to create MinHash from a set of shingles
def create_minhash(shingles, num_perm):
    minhash = MinHash(num_perm=num_perm)
    for shingle in shingles:
        minhash.update(shingle.encode('utf8'))
    return minhash

# Function to build the LSH index
def build_lsh_index(train_docs, num_perm, b, r, threshold=0.9):
    start_time = time.time()

    lsh = MinHashLSH(threshold=threshold, num_perm=num_perm, params=(b, r))
    train_minhashes = {}

    # Add train documents to LSH index
    for idx, shingles in enumerate(train_docs):
        minhash = create_minhash(shingles, num_perm)
        lsh.insert(str(idx), minhash)
        train_minhashes[idx] = minhash

    build_time = time.time() - start_time
    return lsh, train_minhashes, build_time
```

```python
# Function to perform LSH-based nearest neighbor search
def lsh_nearest_neighbors(X_test, X_test_vectorized, X_train_vectorized, lsh, num_perm, b, k=7):
    top_k_filename = f'lsh_{num_perm}_{b}_top_k.txt'
    total_comparisons = 0
    start_time = time.time()
    # Precompute the word counts for each training document
    train_word_counts = np.array(X_train_vectorized.sum(axis=1)).ravel()
    num_tests = X_test_vectorized.shape[0]

    # Open temporary file to write results (in order to avoid memory overflow)
    with open(top_k_filename, 'w') as file:
        for i in range(num_tests):
            # Get the batch's test row
            test_row = X_test.iloc[i]
            # Find the candidate neighbors from LSH
            minhash = create_minhash(test_row, num_perm)
            candidates = lsh.query(minhash)  # Get candidate neighbors from LSH
            candidate_indices = np.array(list(map(int, candidates)))
            # No need to compare if candidates <= k
            if len(candidate_indices) <= k:
                file.write(f'{candidate_indices.tolist()}\n')
                continue

            # Compute intersection: test_row dot product with candidate train rows
            test_row = X_test_vectorized[i]
            intersection = test_row.dot(X_train_vectorized[candidate_indices, :].T).toarray().ravel()
            total_comparisons += len(candidate_indices)

            # Compute union: train_word_counts + test_word_count - intersection
            test_word_count = test_row.sum()
            candidates_word_counts = train_word_counts[candidate_indices]
            union = candidates_word_counts + test_word_count - intersection
            union[union == 0] = 1   # Avoid division by zero
            # Compute Jaccard similarity
            jaccard_similarities = intersection / union
            top_k = min(k, len(jaccard_similarities))
            # Find the indices of the top K highest similarity values
            top_k_indices = np.argpartition(-jaccard_similarities, top_k)[:top_k]
            top_k_indices = candidate_indices[top_k_indices]

            file.write(f'{top_k_indices.tolist()}\n')
            if i % 1000 == 0:
                file.flush()

    query_time = time.time() - start_time
    average_computed_similarities = total_comparisons/num_tests
    return top_k_filename, query_time, average_computed_similarities
```

Επιλογή Παραμέτρων:

Για να πετύχουμε το όριο ομοιότητας να είναι κοντά στο threshold τ=0.9, επιλέγουμε τις τιμές των παραμέτρων να είναι τέτοιες ώστε $\left(\frac{1}{b}\right)^{\frac{1}{r}} \cong 0.9$.

Άρα για num_permutations={16,32,64} επιλέγουμε αντίστοιχα:

- $b = 2, r = 8 \rightarrow \tau \cong 0.917$ ή $b = 4, r = 4 \rightarrow \tau \cong 0.707$
- $b = 2, r = 16 \rightarrow \tau \cong 0.958$ ή $b = 4, r = 8 \rightarrow \tau \cong 0.841$
- $b = 2, r = 32 \rightarrow \tau \cong 0.979$ ή $b = 4, r = 16 \rightarrow \tau \cong 0.917$ ή $b = 8, r = 8 \rightarrow \tau \cong 0.771$

Παρατηρούμε στον παρακάτω πίνακα ότι για τις παραπάνω περιπτώσεις τα queries τερματίζουν γρήγορα αλλά με πολύ χαμηλό ποσοστό ταιριάσματος συγκριτικά με τον Brute-Force αλγόριθμο. Αυτό οφείλεται στο γεγονός ότι ο περιορισμός για την ομοιότητα είναι πολύ υψηλός, ειδικά αν λάβουμε υπόψιν ότι έχουμε αφαιρέσει τα stopwords κατά τη διάρκεια του preprocessing, με αποτέλεσμα να μειώσουμε σε κάποιο βαθμό την ομοιότητα μεταξύ των εγγράφων. Έτσι, το πλήθος των candidates που επιστρέφει το LSH είναι πολύ μικρό.

Ενδεικτικά, για τα 7 παραπάνω queries, ο αριθμός των πιθανών όμοιων εγγράφων που επιστρέφει το LSH να είναι κατά μέσο όρο 0.041, 0.482, 0.013, 0.06, 0.008, 0.026 και 0.109

ανά έγγραφο του test dataset αντίστοιχα. Αυτό σημαίνει ότι για τα περισσότερα έγγραφα του test dataset βρέθηκαν 0 ή το πολύ 1 έγγραφα του training με τα οποία η ομοιότητα είναι αρκετά υψηλή, με αποτέλεσμα να είναι πρακτικά αδύνατη η χρήση του KNN για τον εντοπισμό όμοιων εγγράφων.

Συνεπώς, δοκιμάζουμε να μειώσουμε την αυστηρότητα του threshold για την ομοιότητα, δοκιμάζοντας επιπλέον τις εξής παραμέτρους για num_permutations={16,32,64} αντίστοιχα:

- $b = 8, r = 2 \rightarrow \tau \cong 0.354$
- $b = 8, r = 4 \rightarrow \tau \cong 0.595$ ή $b = 16, r = 2 \rightarrow \tau \cong 0.25$
- $b = 16, r = 4 \rightarrow \tau \cong 0.5$ ή $b = 32, r = 2 \rightarrow \tau \cong 0.177$

Με αυτόν τον τρόπο αυξάνουμε το κόστος των queries, αλλά αυξάνουμε σημαντικά το πλήθος των πιθανών όμοιων εγγράφων σε 824.291, 1.152, 1356.402, 2.19 και 2606.671 ανά έγγραφο του test dataset αντίστοιχα. Συνεπώς, παρατηρούμε αρκετά μεγάλη αύξηση και στο ποσοστό K-πιο όμοιων εγγράφων που είναι κοινά μεταξύ του Min-Hash LSH και του Brute-Force αλγορίθμου, το οποίο σημαίνει ότι μπορούμε να πετύχουμε αρκετά καλά ποσοστά και στις αντίστοιχες προβλέψεις.

| Type | BuildTime (sec) | QueryTime (sec) | TotalTime (sec) | Faction of K most similar reported by LSH | Parameters |
|---|---|---|---|---|---|
| Brute-Force Jaccard | 0.0 | 10621.21 | 10621.21 | 100% | - |
| LSH-Jaccard (Perm=16) | 151.61 | 67.88 | 219.49 | 3.00% | Perm=16, b=2, r=8, τ=0.917 |
| LSH-Jaccard (Perm=16) | 166.49 | 78.04 | 244.53 | 6.33% | Perm=16, b=4, r=4, τ=0.707 |
| LSH-Jaccard (Perm=16) | 176.54 | 236.44 | 412.97 | 32.33% | Perm=16, b=8, r=2, τ=0.354 |
| LSH-Jaccard (Perm=32) | 190.26 | 71.42 | 261.67 | 1.86% | Perm=32, b=2, r=16, τ=0.958 |
| LSH-Jaccard (Perm=32) | 194.94 | 77.53 | 272.47 | 3.59% | Perm=32, b=4, r=8, τ=0.841 |

| | | | | | |
|---|---|---|---|---|---|
| LSH-Jaccard (Perm=32) | 191.95 | 83.06 | 275.01 | 8.10% | Perm=32, b=8, r=4, τ=0.595 |
| LSH-Jaccard (Perm=32) | 185.64 | 287.87 | 473.52 | 46.60% | Perm=32, b=16, r=2, τ=0.25 |
| LSH-Jaccard (Perm=64) | 179.61 | 78.57 | 258.17 | 1.11% | Perm=64, b=2, r=32, τ=0.979 |
| LSH-Jaccard (Perm=64) | 180.69 | 76.08 | 256.77 | 2.35% | Perm=64, b=4, r=16, τ=0.917 |
| LSH-Jaccard (Perm=64) | 174.21 | 73.10 | 247.32 | 4.23% | Perm=64, b=8, r=8, τ=0.771 |
| LSH-Jaccard (Perm=64) | 174.81 | 77.94 | 252.76 | 10.77% | Perm=64, b=16, r=4, τ=0.5 |
| LSH-Jaccard (Perm=64) | 177.90 | 446.88 | 624.78 | 65.59% | Perm=64, b=32, r=2, τ=0.177 |

Επιρροή παραμέτρων στο χρόνο και στην απόδοση:

Αυξάνοντας τον αριθμό των permutations χρειάζεται να υπολογιστούν περισσότερα hash functions για κάθε έγγραφο, με αποτέλεσμα την αύξηση του χρόνου για το build και στη συνέχεια για το query, οδηγώντας όμως παράλληλα σε μεγαλύτερη ακρίβεια, αφού βελτιώνεται η εκτίμηση για τον υπολογισμό του Jaccard similarity.

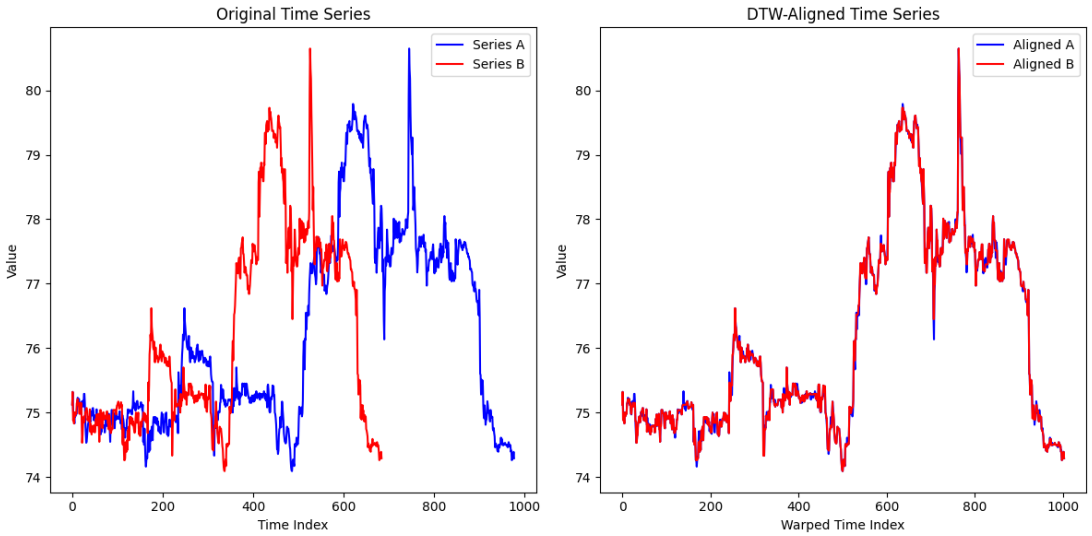
Αντίστοιχα, αυξάνοντας το πλήθος των bands (b) και μειώνοντας το πλήθος των rows (r), αυξάνουμε λίγο τους χρόνους για το build και query του LSH. Η σημαντικότερη επιρροή των 2 αυτών παραμέτρων όμως είναι ότι μειώνουμε επίσης το όριο του αναμενόμενου similarity μεταξύ των εγγράφων που επιστρέφει το query του LSH, με αποτέλεσμα να επιστρέφονται περισσότερα έγγραφα ως candidates (και περισσότερα False Positives μεταξύ αυτών). Αυτό βέβαια έχει ως αποτέλεσμα να αυξάνεται ο συνολικός χρόνος του query, ο οποίος περιλαμβάνει και τον ακριβή υπολογισμό της ομοιότητας του test εγγράφου με όλα τα πιθανά όμοιά του που επιστρέφει το LSH, και κατόπιν την εύρεση των k=7 πιο όμοιων εξ αυτών.

Με βάση τα αποτελέσματα του παραπάνω πίνακα, φαίνεται ότι για το dataset μας απαιτείται λιγότερο αυστηρό όριο για το threshold της αναμενόμενης ομοιότητας, το οποίο πιθανόν να οφείλεται στο ότι δεν υπάρχουν πολλά αρκετά όμοια έγγραφα στο test dataset σε σχέση με το train dataset. Βέβαια, ακόμη και για την πιο αργή εκτέλεση (με 64 permutations, b=32, r=2), ο συνολικός χρόνος εκτέλεσης μειώνεται σημαντικά σε σχέση με τον Brute-Force αλγόριθμο, αφού με τη χρήση του Min-Hash LSH πετυχαίνουμε περίπου 17 φορές γρηγορότερη εκτέλεση, εκτελώντας 2,606.671 συγκρίσεις ανά έγγραφο (έναντι των 111,795 του Brute-Force αλγορίθμου) και διατηρώντας το 65.59% των K-Nearest Neighbors που επέστρεψε ο Brute-Force αλγόριθμος.

***PART 3:***

Σκοπός:

Υπολογισμός της απόστασης μεταξύ χρονικών σειρών χρησιμοποιώντας Dynamic Time Warping (DTW) και Ευκλείδεια απόσταση.




Υλοποίηση:

- Δεδομένα:

  Μετατροπή των σειρών σε np.array

```
import pandas as pd

# Load dataset
dataset_path = 'dtw_test.csv'
data = pd.read_csv(dataset_path)
```

```
import numpy as np

# Convert string representation of lists to actual lists
data['series_a'] = data['series_a'].apply(lambda x: np.array(eval(x)))
data['series_b'] = data['series_b'].apply(lambda x: np.array(eval(x)))
```

- Ευκλείδεια απόσταση:

  Δημιουργία συνάρτησης που υπολογίζει την απόσταση μεταξύ των σημείων

```
# Euclidean Distance Calculation
def euclidean_distance(a, b):
    return np.sqrt((a - b) ** 2)
```

- DTW:

  Δημιουργία συνάρτησης που υπολογίζει την DTW απόσταση μεταξύ των σημείων

```
# DTW Distance between 2 given time series (a, b)
def dtw_distance(series_a, series_b):
    n, m = len(series_a), len(series_b)
    dtw_matrix = np.full((n + 1, m + 1), np.inf)
    dtw_matrix[0, 0] = 0

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            cost = euclidean_distance(series_a[i - 1], series_b[j - 1])
            dtw_matrix[i, j] = cost + min(dtw_matrix[i - 1, j - 1],
                                          dtw_matrix[i - 1, j],
                                          dtw_matrix[i, j - 1])


    return dtw_matrix[n, m]
```

Επεξήγηση DTW:
Μέσω του DTW μπορούμε να ευθυγραμμίσουμε δύο χρονικές σειρές, δηλαδή να βρούμε την καλύτερη δυνατή αντιστοίχιση των σημείων τους αν αυτές έχουν διαφορετική ταχύτητα, ώστε να βρούμε τη μικρότερη απόσταση που μπορούν να έχουν. Για να το καταφέρουμε, δημιουργούμε τον πίνακα κόστους (dtw_matrix), έναν πίνακα δηλαδή που περιέχει το συνολικό κόστος της βέλτιστης ευθυγράμμισης για τα πρώτα i στοιχεία της πρώτης χρονικής σειράς με τα πρώτα j στοιχεία της δεύτερης χρονικής σειράς. Έτσι η θέση [n,m] όπου n, m το μήκος των δύο σειρών, περιέχει τη συνολική απόσταση. Για τον υπολογισμό κάθε κελιού αθροίζουμε τη πραγματική απόσταση των αντίστοιχων σημείων και τη μικρότερη από τις τρεις προηγούμενες θέσεις (πάνω, αριστερά και διαγώνια).

Χρόνος εκτέλεσης:
Για τον υπολογισμό των αποτελεσμάτων χρειάστηκαν 45 λεπτά και 30.1 δευτερόλεπτα ( συν 4.3 δευτερόλεπτα για τη μετατροπή των δεδομένων).

```
# Convert string representation of lists to actual lists
data['series_a'] = data['series_a'].apply(lambda x: np.array(eval(x)))
data['series_b'] = data['series_b'].apply(lambda x: np.array(eval(x)))
```

✓ 4.3s

```
# Compute DTW distances
results = []
start_time = time.time()
for idx, row in data.iterrows():
    distance = dtw_distance(row['series_a'], row['series_b'])
    results.append([idx + 1, distance])
end_time = time.time()

time_taken = end_time - start_time
print(f"Total time taken: {time_taken:.4f} seconds")
```

✓ 45m 30.1s

```
Total time taken: 2730.1174 seconds
```

Σημείωση:
Η αρίθμηση των id στο τελικό αρχείο ξεκινάει από το 1 όπως φαίνεται και στον πίνακα της εκφώνησης της άσκησης.