Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Πολυτεχνική Σχολή

Bike-Sharing Analytics System Implementation

Μάθημα: Τεχνολογίες Αποκεντρωμένων Δεδομένων
Διδάσκοντες: Σ. Σιούτας, Α. Κομνηνός

Χειμερινό Εξάμηνο 2024-2025
1 Φεβρουαρίου 2025

Μέλη Ομάδας:

ΚΑΡΑΓΙΑΝΝΗΣ ΓΕΩΡΓΙΟΣ
1084586

ΚΟΛΑΓΚΗ ΕΥΑΓΓΕΛΙΑ
1084599

ΚΟΥΡΗ ΜΑΡΙΑ
1084526

ΜΑΝΤΕΣ ΜΗΛΤΙΑΔΗΣ
1084661

ΠΑΤΕΛΗ ΧΡΥΣΑΥΓΗ
1084513

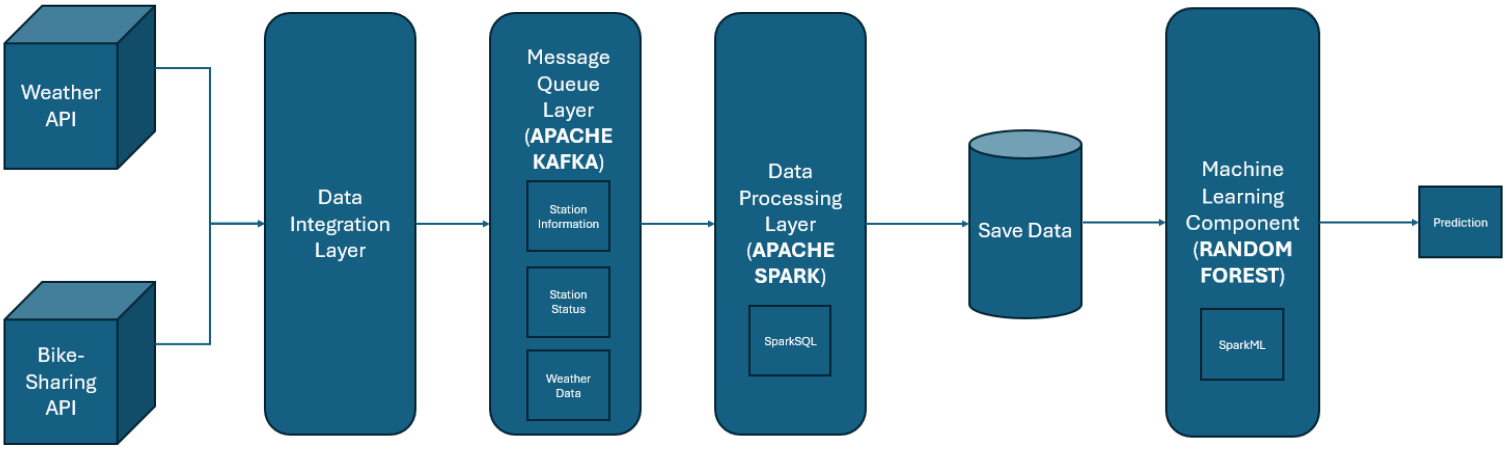
Περιεχόμενα

1. Εισαγωγή

Στο πλαίσιο αυτού του project, έχει δημιουργηθεί ένα σύστημα ανάλυσης δεδομένων σε πραγματικό χρόνο για ένα σύστημα κοινής χρήσης ποδηλάτων (bike-sharing). Δεδομένα συλλέγονται από διάφορες πηγές σε πραγματικό χρόνο μέσω APIs και επεξεργάζονται χρησιμοποιώντας τεχνολογίες κατανεμημένων συστημάτων, όπως το Apache Kafka και το Apache Spark. Ο στόχος της εργασίας είναι να αναλυθεί η χρήση των ποδηλάτων σε μια πόλη και να αναπτυχθεί μοντέλο πρόβλεψης για τη χρήση τους, με βάση τα δεδομένα που συλλέγονται από το δίκτυο κοινής χρήσης ποδηλάτων.

Τα δεδομένα επιλέχθηκε να προέρχονται από την πόλη του **Ντουμπάι** και συγκεκριμένα από την υπηρεσία κοινής χρήσης ποδηλάτων του Ντουμπάι (Dubai Public Bike System). Συλλέγονται πληροφορίες για την κατάσταση των σταθμών ποδηλάτων, τη διαθεσιμότητα των ποδηλάτων και άλλες σχετικές πληροφορίες μέσω των διαθέσιμων API της υπηρεσίας καθώς επίσης και δεδομένα για τον καιρό.

Παρακάτω παρουσιάζεται και η δομή του pipeline που ακολουθήθηκε κατά την υλοποίηση:



Συνημμένα αρχεία κώδικα

Μαζί με τη συγκεκριμένη αναφορά παραδίδονται και τα εξής αρχεία πηγαίου κώδικα:

Αρχείο	Περιγραφή/Σχόλιο
producer.py	Συγκέντρωση δεδομένων από APIs και αποστολή στα αντίστοιχα Kafka topics.
spark.py	Κατανάλωση δεδομένων από τον producer και επεξεργασία δεδομένων για να είναι έτοιμα για είσοδο στο ML.
random_forest.py	Εκπαίδευση μοντέλου, αξιολόγηση και πρόβλεψη χρήσης για την επόμενη ώρα on demand.
spark_data.csv	Τα δεδομένα που συλλέχθηκαν από τον spark consumer.

Github

Τέλος, παρατίθεται και ο αντίστοιχος σύνδεσμος για το Github της εργασίας, όπου περιλαμβάνοντα όλα τα απαραίτητα αρχεία: https://github.com/chryssa-pat/Decentralized-Data-Technologies

2. Περιβάλλον Υλοποίησης και Installations

2.1 WSL

Για την υλοποίηση του project χρησιμοποιήθηκε το **WSL** (Windows Subsystem for Linux). Το WSL μας παρέχει την δυνατότητα να τρέξουμε έναν πλήρη πυρήνα Linux απευθείας μέσα στο λειτουργικό σύστημα μας, χωρίς την ανάγκη για ξεχωριστή εικονική μηχανή. Το περιβάλλον αυτό επιλέχθηκε γιατί προσφέρει σταθερή και αποτελεσματική διαχείριση των πόρων του συστήματος μας, κάτι το οποίο είναι αναγκαίο αφού εφαρμογές όπως το Kafka και το Spark απαιτούν ταχύτητα, χαμηλή καθυστέρηση και υψηλή απόδοση για τη διαχείριση μεγάλων όγκων δεδομένων σε πραγματικό χρόνο.

Για την εγκατάσταση του έγιναν τα εξής βήματα :

- Σε command prompt εκτελείται η εντολή **wsl –install.**
- Εγκαθίσταται η έκδοση Linux που επιθυμούμε (Debian ή Ubuntu) με την εντολή **wsl --install -d Debian** ή **wsl --install -d Ubuntu.**

2.2 Εγκατάσταση Apache Kafka

Ο Apache Kafka είναι ένα σύστημα ανοιχτού κώδικα για τη ροή δεδομένων σε πραγματικό χρόνο και τη διαχείριση μηνυμάτων. Χρησιμοποιείται για την αποστολή, αποθήκευση και επεξεργασία δεδομένων που μετακινούνται από διάφορες πηγές, από APIs στην συγκεκριμένη περίπτωση.

Για την εγκατάσταση του kafka γίνεται εγκατάσταση του αρχείου **kafka_2.12-3.2.0.tgz** στο περιβάλλον μας με την εξής εντολή:

```
root@LAPTOP-PTVD2I4Q:~/kafka/kafka_2.12-3.2.0# wget https://downloads.apache.org/kafka/3.2.0/kafka_2.12-3.2.0.tgz
```

Με την εντολή ls θα δούμε ότι έγινε επιτυχώς εγκατάσταση. Επειδή, όμως, το αρχείο είναι σε μορφή **.tgz** θα γίνει extract ώστε να δημιουργηθεί ο φάκελος **kafka_2.12-3.2.0**, όπως φαίνεται στην συνέχεια.

```
root@LAPTOP-PTVD2I4Q:~/kafka# ls
kafka_2.12-3.2.0  kafka_2.12-3.2.0.tgz
```

Για να τρέξουμε τον Kafka χρειάζεται επίσης στο περιβάλλον μας να γίνει εγκατάσταση της **Java** (**OpenJDK**).

Αυτό πραγματοποιείται με την παρακάτω εντολή:

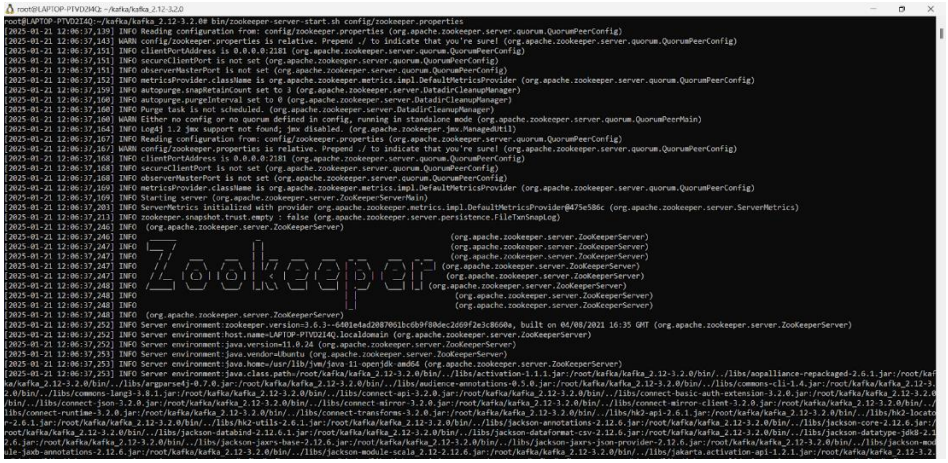
```
root@LAPTOP-PTVD2I4Q:~/kafka/kafka_2.12-3.2.0# sudo apt install openjdk-11-jdk -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

2.3 Apache Zookeeper

Ο **Apache Zookeeper** είναι ένα distributed coordination service που χρησιμοποιείται για τη διαχείριση και συγχρονισμό κατανεμημένων συστημάτων. Παρέχει έναν αξιόπιστο μηχανισμό που διευκολύνει τον συγχρονισμό μεταξύ των υπηρεσιών σε ένα cluster.

Η εκκίνηση Zookeeper γίνεται τρέχοντας το αρχείο **zookeeper.properties** με το εξής path:

bin/zookeeper-server-start.sh config/zookeeper.properties

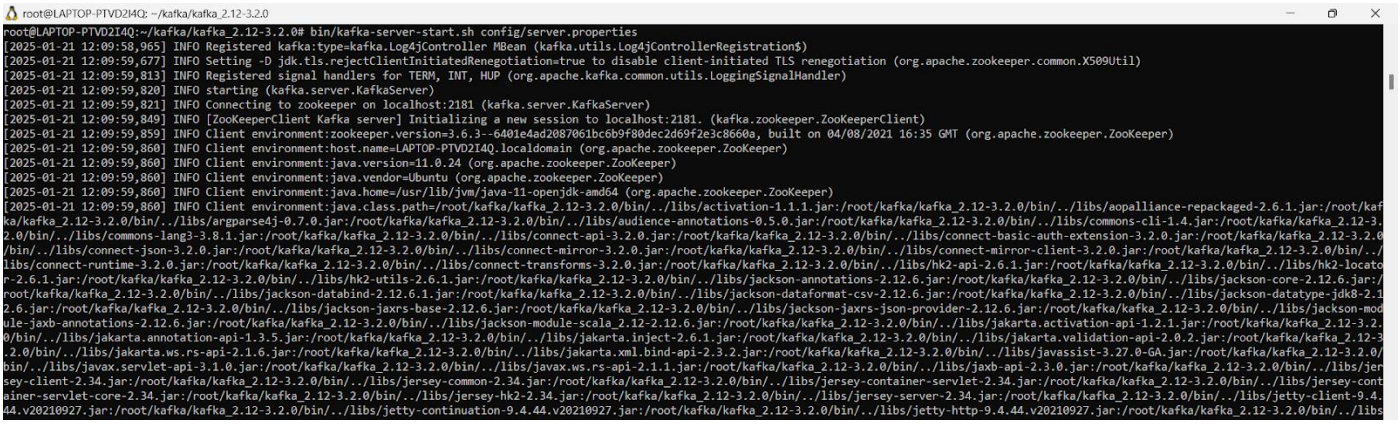


2.4 Kafka-server

Ο Kafka Server είναι ο διακομιστής (**broker**) του Apache Kafka, που είναι υπεύθυνος για τη διαχείριση και δρομολόγηση των μηνυμάτων μέσα στο σύστημα Kafka. Ο Kafka Server λειτουργεί ως κεντρικός κόμβος που **δέχεται δεδομένα** από τον producer, **αποθηκεύει τα μηνύματα** σε topics και partitions και **παραδίδει τα δεδομένα** στους consumers.

Η εκκίνηση Kafka-server γίνεται τρέχοντας το αρχείο **server.properties** με το εξής path:

bin/kafka-server-start.sh config/server.properties:



2.5 Apache Spark

Το Apache Spark είναι μια ανοιχτού κώδικα κατανεμημένη μηχανή επεξεργασίας δεδομένων που επιτρέπει τη γρήγορη ανάλυση μεγάλων δεδομένων (Big Data) μέσω παράλληλης επεξεργασίας. Χρησιμοποιείται για επεξεργασία δεδομένων σε real-time, όπως στο συγκεκριμένο project.

Διαθέτει κάποια βασικά Components που αποτελούν πολύ βοηθητικά εργαλεία, όπως το SparkSQL που χρησιμοποιούμε στην συνέχεια και επιτρέπει την εκτέλεση SQL ερωτημάτων πάνω σε μεγάλα δεδομένα.

Εγκατάσταση Apache Spark

Με την παρακάτω εντολή γίνεται εγκατάσταση του Apache Spark:

```
root@LAPTOP-PTVD2I4Q:~# wget https://downloads.apache.org/spark/spark-3.4.0/spark-3.4.0-bin-hadoop3.tgz
```

Τέλος, το αρχείο .tgz αποσυμπιέζεται:

```
root@LAPTOP-PTVD2I4Q:~# tar -xvzf spark-3.4.0-bin-hadoop3.tgz
```

3. Producer

Ο producer (εφαρμογή παραγωγής δεδομένων) είναι υπηρεσία του Apache Kafka και είναι υπεύθυνη για την συλλογή, την δημιουργία και την αποστολή δεδομένων σε ένα ή περισσότερα Kafka topics ενός Kafka Cluster. Τα δεδομένα συλλέγονται από τα δοθέντα API και προωθούνται στον Kafka για να είναι διαθέσιμα στον consumer (Apache Spark). Ο σχεδιασμός επιτρέπει την συνεχή ροή των δεδομένων, τον έλεγχο σφαλμάτων και την διαχείρισή τους, καθώς επίσης και την επικύρωση των δεδομένων.

Δημιουργείται ένας Kafka Producer, ο οποίος καλεί τα 3 εξωτερικά API και λαμβάνει δεδομένα (fetch_data) για τις καιρικές συνθήκες (weather API), πληροφορίες για τους σταθμούς ποδηλάτων (Station Information API) στην πόλη Dubai και δεδομένα για την κατάσταση των σταθμών ποδηλάτων (Station Status API). Οι κλήσεις πραγματοποιούνται κάθε 1 ώρα για τον καιρό και κάθε 5 λεπτά για τους σταθμούς. Οι προηγούμενοι χρόνοι επιλέγονται, καθώς είναι εύλογα χρονικά διαστήματα για να προλάβουν να πραγματοποιηθούν αλλαγές και να μεταβληθούν τα API. Ακολουθεί η ανάλυση του κώδικα:

Για την ανάπτυξη του κώδικα χρησιμοποιούνται οι ακόλουθες βιβλιοθήκες:

- **logging**: διαχειρίζεται την καταγραφή των πληροφοριών του συστήματος και τον σφαλμάτων.
- **json**: επιτρέπει την διαχείριση δεδομένων JSON.
- **time**: είναι υπεύθυνη για τις χρονοκαθυστερήσεις.
- **typing**: παρέχει τύπους για τις μεταβλητές .
- **requests**: ανταποκρίνεται σε αιτήματα HTTP.
- **jsonschema**: επικυρώνει τη δομή των δεδομένων JSON σύμφωνα με καθορισμένα σχήματα.
- **sys**: ενσωματώνεται για τη δρομολόγηση των καταγραφών στην κονσόλα.
- **confluent_kafka**: χρησιμοποιείται για την παραγωγή μηνυμάτων στον Kafka.

Αρχικά, για την καταγραφή των μηνυμάτων (logging) του προγράμματος χρησιμοποιείται η μέθοδος **basicConfig()**, η οποία καθορίζει την μορφή των μηνυμάτων καθώς και ποια μηνύματα θα καταγραφούν. Η σταθερά **INFO** καθορίζει ότι τα μηνύματα που θα καταγραφούν θα ανήκουν σε επίπεδα ίσα και μεγαλύτερα του **INFO** (WARNING, ERROR και CRITICAL). Επίσης, καθορίζει τους handlers, δηλαδή τους στόχους στους οποίους θα αποστέλλονται τα logs. Ο πρώτος είναι ο **FileHandler( )**, ο οποίος εξασφαλίζει ότι τα μηνύματα θα καταγράφονται στο αρχείο **kafka_producer.log**, άρα μπορουν να χρησιμοποιηθούν για μελλοντική ανάλυση. Ο δεύτερος handler είναι ο **StreamHandler( )**, ο οποίος εξασφαλίζει ότι τα μηνύματα θα εμφανίζονται στην κονσόλα, ώστε να παρακολουθούνται σε πραγματικό χρόνο. Τέλος, με την μέθοδο **getLogger( )** δημιουργείται ένα αντικείμενο logger, το οποίο θα χρησιμοποιείται σε όλο το πρόγραμμα για την καταγραφή μηνυμάτων.

```
# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        #logs to a file
        logging.FileHandler('kafka_producer.log'),
        #logs to the console
        logging.StreamHandler(sys.stdout)
    ]
)
logger = logging.getLogger(__name__)
```

Η μορφή των μηνυμάτων μέσα στο αρχείο **kafka_producer.log** είναι η εξής:

```
2025-01-03 22:38:45,668 - __main__ - INFO - Message delivered to station_status_topic
2025-01-03 22:43:47,229 - __main__ - INFO - Message delivered to station_info_topic
2025-01-03 22:43:47,231 - __main__ - INFO - Message delivered to station_status_topic
2025-01-03 22:48:48,759 - __main__ - INFO - Message delivered to station_info_topic
2025-01-03 22:48:48,762 - __main__ - INFO - Message delivered to station_status_topic
2025-01-03 22:53:50,270 - __main__ - INFO - Message delivered to station_info_topic
2025-01-03 22:53:50,274 - __main__ - INFO - Message delivered to station_status_topic
2025-01-03 22:58:51,767 - __main__ - INFO - Message delivered to station_info_topic
2025-01-03 22:58:51,769 - __main__ - INFO - Message delivered to station_status_topic
2025-01-03 23:03:53,505 - __main__ - INFO - Message delivered to station_info_topic
2025-01-03 23:03:53,509 - __main__ - INFO - Message delivered to station_status_topic
2025-01-03 23:08:54,952 - __main__ - INFO - Message delivered to weather_topic
2025-01-03 23:08:54,953 - __main__ - INFO - Message delivered to station_info_topic
2025-01-03 23:08:54,956 - __main__ - INFO - Message delivered to station_status_topic
2025-01-03 23:13:56,478 - __main__ - INFO - Message delivered to station_info_topic
2025-01-03 23:13:56,484 - __main__ - INFO - Message delivered to station_status_topic
2025-01-03 23:18:57,826 - __main__ - INFO - Message delivered to station_info_topic
2025-01-03 23:18:57,829 - __main__ - INFO - Message delivered to station_status_topic
```

Επίσης, ορίζονται σχήματα JSON που θα χρησιμοποιηθούν για την επικύρωση των δεδομένων που συλλέγονται από τα 3 API, διασφαλίζοντας ότι ακολουθούν τη σωστή δομή και μορφή. Κάθε σχήμα καθορίζει την αναμενόμενη δομή των δεδομένων, τα απαιτούμενα πεδία, τους τύπους δεδομένων και τους περιορισμούς, όπως τον αριθμό των στοιχείων σε έναν πίνακα. Τα σχήματα που δημιουργούνται είναι: το **WEATHER_SCHEMA**, **STATION_INFO_SCHEMA**, **STATION_STATUS_SCHEMA**.

Στην συνέχεια, δημιουργείται η συνάρτηση **validate_data()**, η οποία ελέγχει αν τα δεδομένα που δίνονται σαν είσοδος ακολουθούν ένα συγκεκριμένο σχήμα JSON (έχει οριστεί στην αρχή του κώδικα). Δεχεται σαν όρισμα ένα λεξικό **data** που περιέχει τα δεδομένα προς έλεγχο και ένα λεξικό **schema** που περιγράφει το σχήμα που πρέπει να ακολουθούν τα δεδομένα. Ο έλεγχος των δεδομένων πραγματοποιείται χρησιμοποιώντας την συνάρτηση **validate()**, η οποία ελέγχει αν το data ακολουθεί τους κανόνες που ορίζονται στο schema. Αν η επαλήθευση είναι επιτυχής η συνάρτηση επιστρέφει **True**. Σε διαφορετική περίπτωση δημιουργείται εξαίρεση τύπου **jsonschema.exceptions.ValidationError**, καταγράφεται στο log μέσω του logger και η συνάρτηση επιστρέφει **False**.

```python
#Validate JSON data against a given schema.
def validate_data(data: Dict[str, Any], schema: Dict[str, Any]) -> bool:
    try:
        jsonschema.validate(instance=data, schema=schema)
        return True
    except jsonschema.exceptions.ValidationError as e:
        logger.error(f"Data validation error: {e}")
        return False
```

Επιπλέον, δημιουργείται η συνάρτηση **fetch_data()**, η οποία είναι υπεύθυνη για την ανάκτηση των δεδομένων από ένα API με την χρήση πρωτοκόλλου HTTP. Αρχικά χρησιμοποιεί την συνάρτηση **get()** για την αποστολή GET αιτήματος στην διεύθυνση URL, το χρονικό όριο για απάντηση είναι 10s. Έπειτα ελέγχει τον κωδικό κατάστασης του αιτήματος με την συνάρτηση **raise_for_status()** και αν υπάρξει πρόβλημα προκαλείται εξαίρεση. Τα δεδομένα που λαμβάνει από τα 3 API (weather_url, station_info_url, station_status_url) αποκωδικοποιούνται σε μορφή json με την συνάρτηση **json()** και χρησιμοποιώντας την συνάρτηση **validate_data()** ελέγχεται η εγκυρότητα των δεδομένων (data validation).

```python
#Fetch data from an API with rate limiting, retries, and error handling.
def fetch_data(url: str, max_retries: int = 5) -> Optional[Dict[str, Any]]:
    retries = 0
```

```
    backoff = 1  # Start with a 1-second backoff
    while retries < max_retries:
        try:
            response = requests.get(url, timeout=10) # Make a GET request
            response.raise_for_status()

            data = response.json() # Parse the JSON response

            # Validate data based on URL
            if 'weather' in url:
                if validate_data(data, WEATHER_SCHEMA):
                    return data
            elif 'station_information' in url:
                if validate_data(data, STATION_INFO_SCHEMA):
                    return data
            elif 'station_status' in url:
                if validate_data(data, STATION_STATUS_SCHEMA):
                    return data

            logger.warning(f"Data validation failed for {url}")
            return None
```

Για τη διαχείριση των σφαλμάτων γίνονται πολυεπίπεδοι έλεγχοι. Αρχικά, υλοποιείται Error Handling για τις αιτήσεις HTTPS, όπου στην περίπτωση σφάλματος κατά τη διάρκεια αποστολής του αιτήματος (π.χ. timeout, αποτυχία σύνδεσης, κ.ά.) το σφάλμα καταγράφεται (**requests.exceptions.RequestException**) και εφαρμόζεται εκθετική καθυστέρηση (Rate Limiting), πριν γίνει νέα προσπάθεια αίτησης HTTPS. Επιπλέον, εφαρμόζεται Error Handling για τη διαδικασία αποκωδικοποίησης των δεδομένων JSON. Εάν τα δεδομένα που επιστρέφονται δεν μπορούν να αποκωδικοποιηθούν σωστά (**json.JSONDecodeError**), το σφάλμα καταγράφεται και η διαδικασία διακόπτεται, επιστρέφοντας None.

```
  except requests.exceptions.RequestException as e:
            logger.error(f"Request error for {url}: {e}")
            time.sleep(backoff) # Wait before retrying
            backoff *= 2  # Exponential backoff
            retries += 1
        except json.JSONDecodeError as e:
            logger.error(f"JSON decoding error for {url}: {e}")
            return None
```

Ορίζεται επίσης το μέγιστο όριο επαναλήψεων 5 φορές και στην περίπτωση που ξεπεραστεί το σφάλμα καταγράφεται και επιστρέφεται **None**.

```
 logger.error(f"Failed to fetch data from {url} after {max_retries} retries.")
    return None
```

Ακόμη, δημιουργείται η συνάρτηση **delivery_report()**, η οποία δέχεται σαν όρισμα το σφάλμα κατά την προσπάθεια παράδοσης του μηνύματος (**err**) και ένα αντικείμενο (**msg**) που αναπαριστά το μήνυμα που παραδόθηκε ή απέτυχε να παραδοθεί. Χρησιμοποιείται ως callback που καλείται αυτόματα από τον Kafka Producer αφού ολοκληρωθεί η προσπάθεια παράδοσης ενός μηνύματος, ώστε να επιβεβαιωθεί η επιτυχία ή αποτυχία παράδοσης και να γίνει η αντίστοιχη καταγραφή.

```
# Kafka message delivery callback with console and file logging.
def delivery_report(err, msg):
    if err is not None:
        print(f"Message delivery FAILED: {err}")
        logger.error(f"Message delivery failed: {err}")
    else:
        print(f"Message delivered to {msg.topic()}")
        logger.info(f"Message delivered to {msg.topic()}")
```

Επιπλέον, δημιουργείται η συνάρτηση **produce_data()**, η οποία είναι υπεύθυνη για τη συνεχή συλλογή δεδομένων από APIs και την αποστολή τους σε topics του **Kafka**.Τα χρονικά ορόσημα **last_weather_update** και **last_station_update** αρχικοποιούνται στο 0, για να διασφαλιστεί ότι τα δεδομένα θα συλλεχθούν αμέσως κατά την εκκίνηση. Στην συνέχεια, η συνάρτηση εισέρχεται σε έναν ατέρμονα βρόχο όπου τα δεδομένα συλλέγονται και αποστέλλονται περιοδικά. ο τρέχων χρόνος αποθηκεύεται και συγκρίνεται με τα ορόσημα. Όσον αφορά τα δεδομένα καιρού ελέγχει αν έχει περάσει 1 ώρα από την τελευταία συλλογή των δεδομένων από το weather API και αν έχει περάσει καλεί την συνάρτηση **fetch_data()** για να συλλέξει τα δεδομένα από το url του weather. Αν τα δεδομένα είναι έγκυρα δημιουργεί ένα topic (**weather_topic**), αποστέλλει τα δεδομένα με callback το **delivery_report()** και ενημερώνει το χρονικό ορόσημο **last_weather_update.** Η ίδια διαδικασία εκτελείται και για την συλλογή των δεδομένων station Information API και Station Status API, με την διαφορά ότι συλλέγει τα δεδομένα ανά 5 λεπτά. Με την μέθοδο **flush()** διασφαλίζεται ότι όλα τα μηνύματα έχουν αποσταλεί πριν προχωρήσει στον επόμενο κύκλο και με την **sleep()** η εκτέλεση σταματά για 30 δευτερόλεπτα πριν επανέλθει στον έλεγχο των APIs. Σε περίπτωση που προκύψει οποιοδήποτε μη αναμενόμενο σφάλμα κατά την εκτέλεση καταγράφεται στο logger ως κρίσιμο σφάλμα και εμφανίζεται μήνυμα στην κονσόλα. Τέλος, ανεξάρτητα από το αν προκλήθηκε σφάλμα, διασφαλίζεται ότι όλα τα μηνύματα έχουν αποσταλεί (**flush()**) και ο Kafka Producer κλείνει (**close()**).

```
#Continuously fetch and produce data to Kafka topics with error handling.
def produce_data():
    print("Starting Kafka Producer...")

    # Initialize to 0 to ensure immediate fetching
    last_weather_update = 0
    last_station_update = 0

    try:
        while True:
            print("\n--- Fetching Data ---")
            current_time = time.time()

            # Fetch Weather API every 1 hour or first run
            if current_time - last_weather_update >= 3600 or last_weather_update ==
0:
                weather_data = fetch_data(weather_url)
                if weather_data:
                    print("Weather data fetched successfully")
                    producer.produce('weather_topic', key='weather',
value=json.dumps(weather_data), callback=delivery_report)
                    last_weather_update = current_time  # Update the last fetch
time

            # Fetch Station Info and Status API every 5 minutes or first run
```

```
            if current_time - last_station_update >= 300 or last_station_update ==
0:
                station_info_data = fetch_data(station_info_url)
                if station_info_data:
                    print("Station Info data fetched successfully")
                    producer.produce('station_info_topic', key='station_info',
value=json.dumps(station_info_data), callback=delivery_report)

                station_status_data = fetch_data(station_status_url)
                if station_status_data:
                    print("Station Status data fetched successfully")
                    producer.produce('station_status_topic', key='station_status',
value=json.dumps(station_status_data), callback=delivery_report)

                last_station_update = current_time  # Update the last fetch time

            producer.flush()  # Ensure all messages are sent
            print(f"Waiting for next fetch cycle...")
            time.sleep(30)  # Sleep for 30 seconds before checking again

    except Exception as e:
        print(f"Unhandled exception: {e}")
        logger.critical(f"Unhandled exception in produce_data: {e}")
    finally:
        producer.flush()  # Ensure no messages are left in the buffer
        producer.close()  # Close the producer
```

Τέλος, ξεκινά ο Kafka Producer καλώντας την συνάρτηση **produce_data( )** για να πραγματοποιηθεί η συλλογή και αποστολή δεδομένων στα Kafka topics και παρακολουθούνται οι διακοπές που μπορεί να προκαλέσει ο χρήστης μέσω του πληκτρολογίου του.

```
if __name__ == "__main__":
    try:
        print("Kafka Producer is starting...")
        produce_data()  # Start producing data
    except KeyboardInterrupt:
        print("\nKafka Producer stopped by user")
        logger.info("Kafka Producer stopped by user")
```

Η παραπάνω υλοποίηση παρέχει αξιοπιστία, καθώς ελέγχει τα δεδομένα και εξασφαλίζει την εγκυρότητα τους, ενώ ταυτόχρονα είναι ανθεκτική σε σφάλματα λόγω του πολυεπίπεδου error handling. Ο σχεδιασμός εξασφαλίζει την συνεχή λήψη και αποστολή δεδομένων, σε λογικά χρονικά πλαίσια.

4. Apache Spark

Αυτό το τμήμα κώδικα αποτελεί μια εφαρμογή real-time streaming με Apache Kafka και Apache Spark, που συνδυάζει τα δεδομένα από τις διάφορες πηγές για να υπολογίσει τη χρησιμοποίηση των σταθμών ποδηλάτων και να τα συνδυάσει με τα δεδομένα καιρού. Τα αποτελέσματα αποθηκεύονται σε CSV αρχεία με βάση την χρονική στιγμή που διαβάστηκαν.

Για την ανάπτυξη του κώδικα χρησιμοποιούνται οι ακόλουθες βιβλιοθήκες:

- **PySpark.sql (SparkSession):** Χρησιμοποιείται για την επεξεργασία δεδομένων μέσω του PySpark. Η κλάση SparkSession είναι η κύρια είσοδος για τη δημιουργία, παραμετροποίηση και εκτέλεση των Spark jobs.
- **pyspark.sql.functions:** Εισάγει μια σειρά από λειτουργίες του PySpark που επιτρέπουν τη μεταμόρφωση των δεδομένων και την εφαρμογή διάφορων συναρτήσεων π.χ φίλτρα, συγχώνευση δεδομένων κα.
- **pyspark.sql.types:** Εισάγεται για να δηλώσει τα σχήματα (schemas) των δεδομένων.
- **pyspark.sql.window:** Χρησιμοποιείται για εκτέλεση παραθύρων πάνω στα δεδομένα.
- **Datetime:** Χρησιμοποιείται για την δημιουργία timestamps στην αποθήκευση των δεδομένων.

Το **Apache Kafka** χρησιμοποιείται για να συλλεχθούν δεδομένα σε πραγματικό χρόνο από τα 3 διαφορετικά Kafka topics: station_info_topic, station_status_topic και weather_topic. Αυτά τα δεδομένα επεξεργάζονται σε πραγματικό χρόνο χρησιμοποιώντας το **Apache Spark Structured Streaming**, το οποίο επιτρέπει τη ροή των δεδομένων και την εφαρμογή επεξεργασίας πάνω σε αυτά.

Για την επεξεργασία των δεδομένων από κάθε topic, αρχικά, δημιουργείται ένα Sparksession με όνομα "**Kafka Multi Topic Streaming".** Η ενέργεια αυτή είναι το σημείο εκκίνησης για όλες τις λειτουργίες του Apache Spark.

```
# Create SparkSession
spark = SparkSession.builder \
    .appName("KafkaMultiTopicStreaming") \
    .getOrCreate()
```

Επειδή χρειάζεται από κάθε topic να συλλεχθεί ένα συγκεκριμένο κομμάτι των δεδομένων χρησιμοποιείται η μέθοδος **StructType([...])**, η οποία καθορίζει την δομή των δεδομένων που αναμένεται από τα Kafka topics.

Από το topic **Station Info** συλλέγεται το **station_id** το οποίο χρησιμεύει για την ομαδοποίηση με τα δεδομένα από το topic **Station Status,** και το **capacity** το οποίο χρησιμεύει για τον υπολογισμό του ποσοστού χρήσης των ποδηλάτων σε κάθε σταθμό.

Από το topic **Station Status** συλλέγεται το station_id για το join με τα δεδομένα από το topic **Station Info**, ο αριθμός των διαθέσιμων ποδηλάτων και τον διαθέσιμων θέσεων. Η μέθοδος αυτή διευκολύνει την επεξεργασία διότι τα δεδομένα που έρχονται από το Kafka είναι σε json μορφή και το Spark μπορεί εύκολα να αναλύσει τα JSON δεδομένα σε DataFrames με προκαθορισμένα πεδία.

```
# Schema definitions for topics
stationInfoSchema = StructType([
    StructField("station_id", StringType(), True),
    StructField("capacity", IntegerType(), True),
])

stationStatusSchema = StructType([
    StructField("station_id", StringType(), True),
    StructField("num_bikes_available", IntegerType(), True),
    StructField("num_docks_available", IntegerType(), True),
```

```
])

dataSchema1 = StructType([StructField("stations", ArrayType(stationInfoSchema),
True)])
dataSchema2 = StructType([StructField("stations", ArrayType(stationStatusSchema),
True)])

kafkaSchema1 = StructType([StructField("last_updated", IntegerType(), True),
StructField("ttl", IntegerType(), True), StructField("data", dataSchema1, True)])
kafkaSchema2 = StructType([StructField("last_updated", IntegerType(), True),
StructField("ttl", IntegerType(), True), StructField("data", dataSchema2, True)])
```

Έπειτα, γίνεται χρήση του **spark.readstream** για να διαβαστούν σε πραγματικό χρόνο τα δεδομένα από τα topics. Για κάθε ένα από τα topics ορίζεται ο Kafka Broker στον οποίο θα γίνει σύνδεση και εν συνεχεία εγγραφή για το διάβασμα των δεδομένων. Παρακάτω αναλύονται οι ενέργειες που γίνονται για κάθε topic:

- **Station Info:** Τα δεδομένα μετατρέπονται σε μορφή JSON με την εντολή **(CAST (value AS STRING))**, εφαρμόζεται το σχήμα **kafkaSchema1** και εξάγονται πληροφορίες για τους σταθμούς. Η μέθοδος **explode** χρησιμοποιείται για να διαχωριστούν οι εγγραφές στο DataFrame. Επειδή στα ονόματα των σταθμών δεν αναφέρεται πάντα το όνομα της πόλης “Dubai”, προστίθεται μια στήλη με την ονομασία **city_name** στην οποία καταχωρείται η τιμή "Dubai" για κάθε σταθμό. Αυτό βοηθά στη μετέπειτα συνένωση των δεδομένων. Το τελικό αποτέλεσμα είναι το **stationInfoDF** με τις στήλες station_id, name, capacity και city_name.

```
# Read station information
stationInfoDF = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "station_info_topic") \
    .load() \
    .selectExpr("CAST(value AS STRING) as json") \
    .select(from_json(col("json"), kafkaSchema1).alias("data")) \
    .select(explode("data.data.stations").alias("station")) \
    .select(
        col("station.station_id").alias("station_id"),
        col("station.capacity").alias("capacity"),
        lit("Dubai").alias("city_name")
    )
```

- **Station Status**: Τα δεδομένα αυτά επίσης μετατρέπονται σε μορφή JSON, και στη συνέχεια εφαρμόζεται το σχήμα **kafkaSchema2** για την εξαγωγή της κατάστασης των σταθμών. Επιπλέον, προστίθεται μια στήλη με το όνομα **status_timestamp**, η οποία περιέχει την ώρα κατά την οποία ελήφθησαν τα δεδομένα από το topic. Η στήλη αυτή βοηθά στον έλεγχο των καθυστερημένων δεδομένων του Station Status μέσω της χρήσης του **watermark**. Στην προκειμένη περίπτωση, το watermark είναι 10 λεπτά, δηλαδή δεδομένα που έχουν καθυστερήσει μέχρι 10 λεπτά μπορούν να επεξεργαστούν από το σύστημα, ενώ δεδομένα που έχουν ξεπεράσει αυτό το χρονικό όριο απορρίπτονται. Το τελικό αποτέλεσμα είναι το **stationStatusDF**, το οποίο περιλαμβάνει τις στήλες station_id, num_bikes_available, num_docks_available και status_timestamp.

```
# Read station status with watermark
```

```
stationStatusDF = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "station_status_topic") \
    .load() \
    .selectExpr("CAST(value AS STRING) as json", "timestamp as status_timestamp") \
    .select(from_json(col("json"), kafkaSchema2).alias("data"),
col("status_timestamp")) \
    .select(explode("data.data.stations").alias("station"),
col("status_timestamp")) \
    .select(
        col("station.station_id").alias("station_id"),
        col("station.num_bikes_available").alias("num_bikes_available"),
        col("station.num_docks_available").alias("num_docks_available"),
        col("status_timestamp")
    ) \
    .withWatermark("status_timestamp", "10 minutes")
```

- **Weather**: Τα δεδομένα λαμβάνονται απευθείας από το αρχείο JSON χρησιμοποιώντας την εντολή **get_json_object( )**, αφού πρώτα τα εισερχόμενα δεδομένα μετατραπούν σε μορφή JSON. Επιπλέον, εφαρμόζεται κατάλληλο **watermark** των 10 λεπτών για τη διαχείριση των καθυστερημένων δεδομένων. Επειδή στο τελικό DataFrame που θα επιστρέφεται από το Spark πρέπει να υπολογιστεί το ποσοστό βροχόπτωσης στην περιοχή από την οποία λαμβάνονται τα δεδομένα, και η πόλη που έχει επιλεγεί είναι το Dubai (λίγες βροχοπτώσεις), συνήθως η τιμή αυτή είναι NULL και το JSON αρχείο περιέχει το πεδίο "clouds": {"all": 0}. Ωστόσο, αν υπάρξει βροχόπτωση, αυτή μπορεί να ληφθεί από το πεδίο **rain.1h**, το οποίο δημιουργείται στο JSON αρχείο που λαμβάνεται από το API του καιρού.

```
# Read weather data with watermark
weatherDF = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("subscribe", "weather_topic") \
    .load() \
    .selectExpr("CAST(value AS STRING) as json", "timestamp as weather_timestamp")
\
    .select(
        get_json_object(col("json"), "$.name").alias("city_name"),
        get_json_object(col("json"),
"$.main.temp").cast("double").alias("temperature"),
        get_json_object(col("json"),
"$.wind.speed").cast("double").alias("wind_speed"),
        get_json_object(col("json"),
"$.clouds.all").cast("int").alias("cloudiness"),
        get_json_object(col("json"),
"$.rain.1h").cast("double").alias("precipitation"),
        col("weather_timestamp")
    ) \
    .withWatermark("weather_timestamp", "10 minutes")
```

Εφόσον διαβάστηκαν και αποθηκεύτηκαν τα δεδομένα από κάθε topic σε κατάλληλα dataframe, στην συνέχεια πρέπει να δημιουργηθούν περιλήψεις της μιας ώρας για τα ποσοστά χρήσης των σταθμών σε όλη την πόλη. Αρχικά υλοποιείται το **join** των δεδομένων **Station Info** και **Station Status**, με βάση το station_id και έπειτα υπολογίζεται το ποσοστό χρήσης των σταθμών (utilization rate) το οποίο είναι ο αριθμός των ποδηλάτων του σταθμού που είναι διαθέσιμα προς την συνολική χωρητικότητα του σταθμού. Επειδή χρειάζεται να δημιουργηθούν περιλήψεις των δεδομένων της μιας ώρας, δημιουργούνται κατάλληλα παράθυρα τα οποία ομαδοποιούν τα δεδομένα για χρονική διάρκεια 60 λεπτών.

```
# Join and process data
stationMetricsDF = stationInfoDF.join(stationStatusDF, "station_id") \
    .withColumn("utilization_rate",
                (col("num_bikes_available") / col("capacity")) * 100) \
    .withColumn("window", window(col("status_timestamp"), "60 minutes"))
```

Προκειμένου να βρεθούν τα στατιστικά στοιχεία **average,max, min και std** του ποσοστού χρήσης των σταθμών, ομαδοποιούνται τα δεδομένα ανά χρονικό παράθυρο και τα ποσοστά υπολογίζονται με τις κατάλληλες συναρτήσεις του spark και το utilization_rate που υπολογίστηκε προηγουμένως.

```
windowedStatsDF = stationMetricsDF.groupBy("city_name", "window") \
    .agg(
        avg("utilization_rate").alias("average_docking_station_utilisation"),
        max("utilization_rate").alias("max_docking_station_utilisation"),
        min("utilization_rate").alias("min_docking_station_utilisation"),
        stddev("utilization_rate").alias("std_dev_docking_station_utilisation")
    )
```

Στην συνέχεια, για την συσχέτιση των δεδομένων με τον εκάστοτε καιρό που επικρατεί στο Dubai γίνεται κατάλληλη συνένωση του **weatherDF** με το DF, το οποίο περιέχει τα στατιστικά στοιχεία χρήσης των σταθμών ανά μία ώρα. Η συνένωση γίνεται βάση του ονόματος της πόλης και εφόσον το weather_timestamp ανήκει μέσα στο χρονικό παράθυρο των περιλήψεων. Έτσι κατασκευάζεται το **dataframe** με τις ακόλουθες στήλες :

- **timestamp**
- **city_name**
- **temperature**
- **wind_speed**
- **precipitation**
- **cloudiness**
- **average_docking_station_utilisation**
- **max_docking_station_utilisation**
- **min_docking_station_utilisation**
- **std_dev_docking_station_utilisation**

```
finalDF = windowedStatsDF.join(
    weatherDF,
    (windowedStatsDF.city_name == weatherDF.city_name) &
    (windowedStatsDF.window.start <= weatherDF.weather_timestamp) &
```

```
    (windowedStatsDF.window.end > weatherDF.weather_timestamp)
) \
.select(
    col("window.start").alias("timestamp"),
    windowedStatsDF.city_name,
    col("temperature"),
    col("wind_speed"),
    col("precipitation"),
    col("cloudiness"),
    col("average_docking_station_utilisation"),
    col("max_docking_station_utilisation"),
    col("min_docking_station_utilisation"),
    col("std_dev_docking_station_utilisation")
)
```

Τέλος καθώς δημιουργείται το τελικό dataframe, με την μέθοδο **writeStream,** καλείται ανά 10 λεπτά, για να μην χαθούν δεδομένα, η συνάρτηση **writeToCSV** η οποία αποθηκεύει το dataframe αυτό σε ένα αρχείο csv. Για να μην χαθούν παλιά δεδομένα, λόγω νέων εγγραφών, χρησιμοποιείται η μέθοδος **append** η οποία προσθέτει τα νέα δεδομένα στα ήδη υπάρχοντα αρχεία δεδομένων χωρίς να τροποποιούνται ή να διαγράφονται. Αυτό εκτελείται για όσο το spark λαμβάνει δεδομένα από τα topic και δημιουργεί περιλήψεις της μιας ώρας για αυτά.

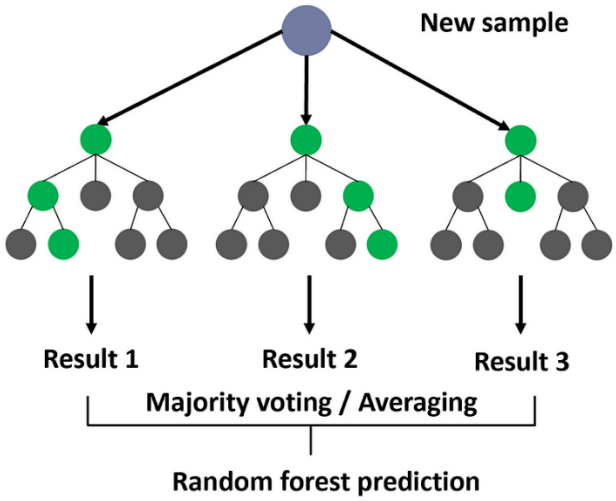
```
# Function to write DataFrame to CSV
def writeToCSV(df, epoch_id):
    # Get current timestamp for filename
    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

    # Write to CSV
    df.write \
        .mode("append") \
        .csv(f"bike_usage_statistics/date={timestamp[:8]}", header=True)

# Write to CSV
csvQuery = finalDF.writeStream \
    .outputMode("append") \
    .trigger(processingTime='10 minutes') \
    .foreachBatch(writeToCSV) \
    .start()

# Wait for termination
spark.streams.awaitAnyTermination()
```

5. Machine Learning

5.1 Εισαγωγή

Για την αξιοποίηση των δεδομένων προκειμένου να γίνει πρόβλεψη μελλοντικών τιμών χρήσης του σταθμού συναρτήσει του καιρού, χρησιμοποιείται ο αλγόριθμος μηχανικής μάθησης Random Forest. Η ιδέα του βασίζεται στη μέθοδο **ensemble learning**, δηλαδή στο συνδυασμό πολλών μοντέλων για να παραχθεί ένα ισχυρότερο και πιο ακριβές μοντέλο. Ο αλγόριθμος μηχανικής μάθησης Random Forest μπορεί να χρησιμοποιηθεί είτε για ταξινόμηση (classification), είτε για παλινδρόμηση (regression). Στην συγκεκριμένη περίπτωση αξιοποιείται για παλινδρόμηση, καθώς θέλουμε να γίνει πρόβλεψη μιας συνεχούς τιμής (bike utilization).

Κατά την εκτέλεση του συγκεκριμένου αλγορίθμου δημιουργούνται πολλαπλά δέντρα απόφασης (decision trees), όπου από τον συνδυασμό τους προκύπτει ένα δάσος (forest). Για την κατασκευή των δέντρων απόφασης ο αλγόριθμος δημιουργεί τυχαία υποσύνολα του αρχικού συνόλου δεδομένων (bootstrapping) μέσω αντικατάστασης και αναθέτει κάθε υποσύνολο σε ένα δέντρο. Στη συνέχεια, κάθε δέντρο επιλέγει ένα υποσύνολο χαρακτηριστικών πάνω στο οποίο θα εκπαιδευτεί και κατά τη διάρκεια της εκπαίδευσης σε κάθε σημείο διαχωρισμού χρησιμοποιείται κάποια μετρική σφάλματος (π.χ. MSE) ως κριτήριο διαχωρισμού των δεδομένων. Πιο συγκεκριμένα, στόχος του κάθε δέντρου είναι να ελαχιστοποιήσει τη συγκεκριμένη μετρική με συνεχόμενες επαναλήψεις μέχρις ότου να μην υπάρχουν δεδομένα για περαιτέρω διαχωρισμό. Όσον αφορά τη πρόβλεψη, κάθε δέντρο προβλέπει μια δική του συνεχή τιμή βάσει των διαθέσιμων δεδομένων του που τροφοδοτήθηκαν στην είσοδο και εν τέλει η γενική πρόβλεψη προκύπτει από τον μέσο όρο των προβλέψεων όλων των δέντρων. Ο συγκεκριμένος αλγόριθμος είναι αρκετά ευέλικτος και η ύπαρξη πολλών δέντρων προσφέρει ανθεκτικότητα στο overfitting, συνεπώς επιτυγχάνει καλή ακρίβεια στην πρόβλεψη.



Η παραπάνω εικόνα απεικονίζει ένα τυχαίο δάσος πρόβλεψης, το οποίο αποτελείται από τρία δέντρα απόφασης. Από ένα τυχαίο δείγμα προκύπτουν τρία δέντρα και ο μέσος όρος των προβλέψεων χρησιμοποιείται ως πρόβλεψη του δάσους.

Για να κάνουμε αποδοτικό tuning του μοντέλου μας είναι επίσης κρίσιμο να ρυθμίσουμε σωστά τις υπερπαραμέτρους του πριν προχωρήσουμε με την εκπαίδευση. Στον Random Forest Regressor οι βασικές υπερπαράμετροι που μπορούμε να ρυθμίσουμε είναι οι εξής:

1. **numTrees:** Καθορίζει τον αριθμό δέντρων στο δάσος.
2. **maxDepth:** Καθορίζει το μέγιστο βάθος κάθε δέντρου.
3. **maxBins:** Καθορίζει τον μέγιστο αριθμό κατηγοριών (bins) που επιτρέπεται για τα συνεχόμενα χαρακτηριστικά κατά τη δημιουργία διακλαδώσεων στα δέντρα.

4. **minInstancesPerNode:** Καθορίζει τον ελάχιστο αριθμό δειγμάτων που πρέπει να υπάρχουν σε κάθε φύλλο (leaf node) του δέντρου.
5. **featureSubsetStrategy:** Καθορίζει τον αριθμό των χαρακτηριστικών που εξετάζονται σε κάθε διαχωρισμό κατά την κατασκευή των δέντρων. Στην **παλινδρόμηση**, η προεπιλεγμένη τιμή είναι sqrt(number_of_features), δηλαδή η τετραγωνική ρίζα του πλήθους των χαρακτηριστικών που χρησιμοποιούνται.
6. **subSamplingRate**: Καθορίζει το ποσοστό των δεδομένων εκπαίδευσης που χρησιμοποιείται για την κατασκευή κάθε δέντρου.

Καταλήγουμε ότι οι ιδανικές τιμές των υπερπαραμέτρων που πρέπει να χρησιμοποιήσουμε είναι:

numTrees	maxDepth	maxBins	minInstancesPerNode	featureSubsetStrategy	subSamplingRate
100	10	32	5	sqrt	0.8

Για την ανάπτυξη του κώδικα χρησιμοποιούνται οι ακόλουθες βιβλιοθήκες:

- **pyspark.sql.SparkSession:** Δημιουργεί και διαχειρίζεται μια συνεδρία Spark. Το SparkSession είναι το σημείο εκκίνησης για τη χρήση του PySpark στον κώδικα.
- **pyspark.ml.feature.VectorAssembler:** Συνδυάζει πολλαπλές στήλες χαρακτηριστικών (features) σε ένα διάνυσμα (vector), το οποίο είναι απαραίτητο για μοντέλα μηχανικής μάθησης στο Spark.
- **pyspark.ml.feature.StandardScaler:** Κανονικοποιεί τα χαρακτηριστικά σε κοινή κλίμακα με μέσο όρο 0 και διασπορά 1, προκειμένου να μην έχουμε μεγάλη επίδραση στη τελική πρόβλεψη των χαρακτηριστικών με μεγάλες τάξεις μεγέθους.
- **pyspark.ml.regression.RandomForestRegressor:** Υλοποιεί τον αλγόριθμο Random Forest για παλινδρόμηση.
- **pyspark.sql.types.StructType:** Ορίζει το schema του DataFrame στο Spark και επιτρέπει στο DataFrame να περιλαμβάνει πολλαπλά StructFields, όπως **FloatType**, **TimestampType**, **StringType**, **IntegerType**.
- **pyspark.sql.functions.col**: Χρησιμοποιείται για να αναφερθούμε σε μια στήλη όταν χρησιμοποιείται σε πράξεις.
- **pyspark.sql.functions.mean**: Υπολογίζει τον μέσο όρο τιμών σε μια στήλη.
- **pyspark.sql.functions.to_timestamp**: Μετατρέπει τα δεδομένα σε χρονική σήμανση τύπου timestamp προκειμένου να είναι συμβατά.
- **pyspark.sql.functions.lit**: Εισάγει σταθερές τιμές (literals) σε DataFrames.
- **pyspark.sql.functions.hour**: Εξάγει την ώρα από τα πεδία τύπου Timestamp.
- **pyspark.sql.functions.dayofweek**: Επιστρέφει την ημέρα της εβδομάδας από τα πεδία τύπου Timestamp (1=Κυριακή και 7=Σάββατο).
- **pyspark.sql.functions.when**: Δημιουργεί συνθήκες τύπου "if-else" για μετασχηματισμό δεδομένων.
- **matplotlib.pyplot**: Χρησιμοποιείται για τη δημιουργία των γραφημάτων που εξάγουμε.
- **seaborn**: Χρησιμοποιείται για τη δημιουργία των heatmaps που εξάγουμε.
- **pandas**: Χρησιμοποιείται για την επεξεργασία των δεδομένων με DataFrames.

- **datetime.timedelta**: Χειρίζεται χρονικά διαστήματα, επιτρέποντας προσθαφαίρεση ημερών και ωρών. Το χρησιμοποιούμε για να υπολογίσουμε ποια θα είναι η επόμενη ώρα για την οποία θα γίνει η πρόβλεψη.

Ξεκινάμε δημιουργώντας μια συνεδρία Spark και στη συνέχεια ορίζουμε το schema των δεδομένων που θα χρησιμοποιήσουμε. Αυτό εξασφαλίζει ότι κάθε στήλη έχει τον σωστό τύπο δεδομένων, δηλαδή τη χρονοσφραγίδα (**timestamp**), το όνομα της πόλης (**city_name**), τα καιρικά χαρακτηριστικά, όπως **temperature**, **wind_speed**, **precipitation**, **cloudiness**, καθώς και τα στατιστικά χαρακτηριστικά, όπως **avg_docking_station_utilisation**, **min_docking_station_utilisation**, **max_docking_station_utilisation** και **std_dev_docking_station_utilisation**. Έπειτα, διαβάζουμε τα δεδομένα τα οποία έχουμε αποθηκεύσει από το προηγούμενο στάδιο σε ένα αρχείο csv και τα αποθηκεύουμε στο DataFrame **bike_data**.

5.2 Προεπεξεργασία δεδομένων

Παρατηρούμε ότι το χαρακτηριστικό **precipitation** στις περισσότερες πλειάδες του dataset έχει τιμή NULL. Αυτό σημαίνει ότι πρέπει να κάνουμε συμπλήρωση των ελλιπών τιμών για να μπορέσουμε να χρησιμοποιήσουμε αποδοτικά τη συγκεκριμένη στήλη στην εκπαίδευση του μοντέλου. Για να διαχειριστούμε τις ελλιπείς τιμές αντικαθιστούμε τη τιμή NULL με τον αριθμητικό μέσο όρο των τιμών της στήλης **precipication**.

```
# Calculate mean for 'precipitation' column
mean_precipitation = bike_data.select(mean(col("precipitation"))).collect()[0][0]

# Handle None case for mean_precipitation
if mean_precipitation is None:
    mean_precipitation = 0.0  # Assign default value

# Ensure the value is of the correct type
mean_precipitation = float(mean_precipitation)

# Replace NULL values with the mean
bike_data = bike_data.fillna({"precipitation": mean_precipitation})
```

Στη συνέχεια, προχωράμε με την προεπεξεργασία στα χρονικά δεδομένα. Πρώτα, γίνεται μετατροπή της στήλης **timestamp** σε τύπο TimestampType. Έπειτα, αφαιρούνται οι πλειάδες με τυχόν περιεχόμενο NaN και εφαρμόζεται Feature Engineering προσθέτοντας τα ακόλουθα χρονικά χαρακτηριστικά σε κάθε πλειάδα: **hour_of_day** (ώρα της ημέρας), **day_of_week** (μέρα της εβδομάδας), **is_weekend** (boolean μεταβλητή για τον έλεγχο αν η τρέχουσα μέρα ανήκει στο Σαββατοκύριακο). Ο λόγος που επιλέχθηκε η εφαρμογή Feature Engineering είναι ότι τα δεδομένα συχνά περιέχουν χρήσιμες πληροφορίες που δεν είναι άμεσα εμφανείς. Έτσι, με τα χαρακτηριστικά που προστέθηκαν μπορούμε να διακρίνουμε τυχόν εποχικότητα ή τάσεις στη χρήση των ποδηλάτων, μοτίβα στις καθημερινές σε σχέση με το Σαββατοκύριακο ή διαφορετικά μοτίβα στη χρήση των ποδηλάτων κατά τη διάρκεια του Σαββατοκύριακου. Συνεπώς, βελτιώνεται η ποιότητα των δεδομένων και έτσι γίνεται καλύτερη κατανόηση της συμπεριφοράς των χρηστών, ενώ παράλληλα αυξάνεται η ακρίβεια πρόβλεψης στο μοντέλο μας.

```
# Clean data by removing rows with NaN values in critical columns
bike_data_cleaned = bike_data.dropna(subset=["timestamp", "temperature",
"wind_speed", "precipitation", "cloudiness"])

# Feature Engineering for Time
bike_data_cleaned = bike_data_cleaned.withColumn("hour_of_day",
hour(col("timestamp")))
bike_data_cleaned = bike_data_cleaned.withColumn("day_of_week",
dayofweek(col("timestamp")))
```

```
bike_data_cleaned = bike_data_cleaned.withColumn("is_weekend",
when((col("day_of_week") == 1) | (col("day_of_week") == 7), 1).otherwise(0))
```

Η τελευταία διαδικασία της προεπεξεργασίας περιλαμβάνει τη κανονικοποίηση των χαρακτηριστικών. Πρώτα μετατρέπονται τα χαρακτηριστικά σε διανύσματα μέσω του **VectorAssembler** και στη συνέχεια χρησιμοποιείται ο **StandardScaler** για την κανονικοποίηση. Πιο συγκεκριμένα, ο **StandardScaler** μετασχηματίζει τα δεδομένα έτσι ώστε να έχουν μέση τιμή (mean) ίση με 0 και τυπική απόκλιση (standard deviation) ίση με 1. Η διαδικασία που ακολουθεί είναι να υπολογίσει τη μέση τιμή και τη τυπική απόκλιση για κάθε feature του dataset και έπειτα να εφαρμόσει σε αυτό τον ακόλουθο μετασχηματισμό: z = (x - μ) / σ. Έτσι, δεν θα υπάρχουν υπερβολικές διαφοροποιήσεις στις τιμές των διαφορετικών features όσον αφορά τη κλίμακα ή την τάξη μεγέθους.

```
# Feature Scaling
feature_columns = [
    'temperature', 'wind_speed', 'precipitation', 'cloudiness',
    'hour_of_day', 'day_of_week', 'is_weekend'
]
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
data = assembler.transform(bike_data_cleaned)

scaler = StandardScaler(inputCol="features", outputCol="scaled_features")
scaler_model = scaler.fit(data)
data = scaler_model.transform(data)
```

Αφού τελειώσει η διαδικασία της προεπεξεργασίας, το τελικό DataFrame πλέον θα περιέχει:

- Τις τελικές κανονικοποιημένες τιμές χαρακτηριστικών στη στήλη **scaled_features**, δηλαδή τις εισόδους (features) για το μοντέλο.
- Την τιμή-target που θέλουμε να προβλέψουμε (**average_docking_station_utilisation**).

```
data = data.select("scaled_features", "average_docking_station_utilisation")
```

5.2 Εκπαίδευση και αξιολόγηση μοντέλου

Αρχικά, το σύνολο δεδομένων που προέκυψε διαχωρίζεται σε training και validation set, με αναλογία 80-20. Στη συνέχεια, ορίζεται το μοντέλο μέσω του αντικειμένου **RandomForestRegressor**, στο οποίο έχουν ρυθμιστεί οι υπερπαράμετροι όπως αναφέρθηκε πιο πάνω. Η εκπαίδευσή πραγματοποιείται μέσω της συνάρτησης **fit( )** πάνω στα **train_data**, ενώ οι προβλέψεις γίνονται με τη μέθοδο **transform( ).**

```
# Split dataset
train_data, validation_data = data.randomSplit([0.8, 0.2], seed=42)

# Train Random Forest Regressor
rf_regressor = RandomForestRegressor(
    featuresCol="scaled_features",
    labelCol="average_docking_station_utilisation",
    numTrees=100,
    maxDepth=10,
    minInstancesPerNode=5,
    maxBins=32,
    featureSubsetStrategy="auto",
    subsamplingRate=0.8
```

```
)
rf_model = rf_regressor.fit(train_data)

# Evaluate the model on the training data
train_predictions = rf_model.transform(train_data)
```

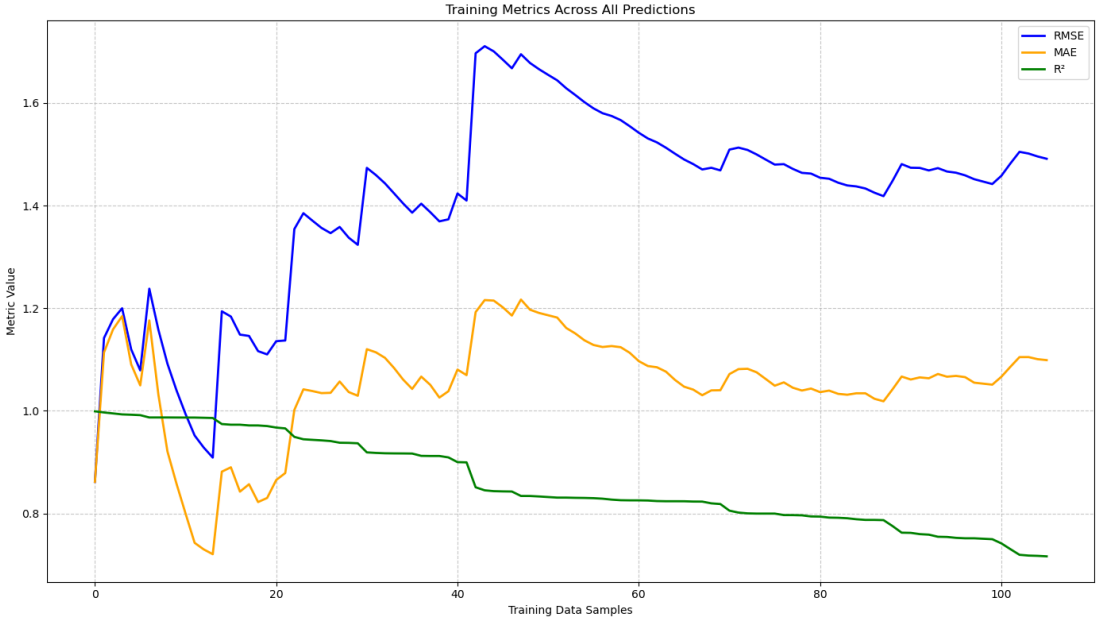
Οι προβλέψεις (δηλαδή η στήλη **prediction**) μαζί με τη στήλη **average_docking_station_utilisation** ενώνονται σε ένα DataFrame και υπολογίζονται οι μετρικές σφάλματος για κάθε πρόβλεψη ως εξής:

- **absolute_error**: Υπολογίζεται η απόλυτη διαφορά μεταξύ της στήλης **prediction** και της στήλης **average_docking_station_utilisation**.
- **squared_error**: Υπολογίζεται το τετράγωνο της διαφοράς της στήλης **prediction** και της στήλης **average_docking_station_utilisation**.

Έπειτα, υπολογίζονται η **σωρευτική ρίζα του μέσου τετραγωνικού σφάλματος** (RMSE), δηλαδή η μέση τιμή του τετραγώνου των λαθών όλων των προβλέψεων, η **σωρευτική μέση απόλυτη τιμή σφάλματος** (MAE), δηλαδή η μέση τιμή των απόλυτων λαθών όλων των προβλέψεων και το **σωρευτικό R^2** το οποίο υπολογίζει πόσο καλά εξηγείται η μεταβλητότητα της εξαρτημένης μεταβλητής από το μοντέλο.

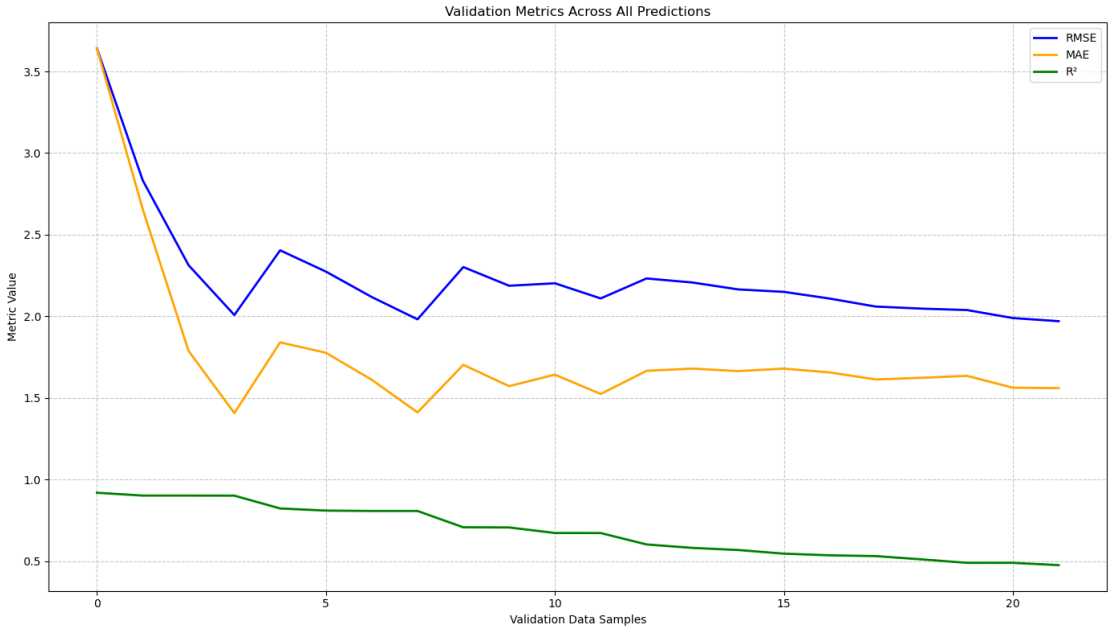
```
# Convert predictions to Pandas DataFrame for further analysis
train_predictions_df = train_predictions.select("prediction",
"average_docking_station_utilisation").toPandas()

# Calculate additional metrics for each prediction
train_predictions_df["absolute_error"] = abs(train_predictions_df["prediction"] -
train_predictions_df["average_docking_station_utilisation"])
train_predictions_df["squared_error"] = (train_predictions_df["prediction"] -
train_predictions_df["average_docking_station_utilisation"]) ** 2

# Cumulative metrics calculations
train_predictions_df["cumulative_rmse"] =
(train_predictions_df["squared_error"].expanding().mean()) ** 0.5
train_predictions_df["cumulative_mae"] =
train_predictions_df["absolute_error"].expanding().mean()
train_predictions_df["cumulative_r2"] = 1 - (
    train_predictions_df["squared_error"].expanding().sum()
    / ((train_predictions_df["average_docking_station_utilisation"] -
train_predictions_df["average_docking_station_utilisation"].mean()) ** 2).sum()
)
```

Οι τρεις αυτές μετρικές εκτυπώνονται και βλέπουμε τις γραφικές τους παρακάτω:



Παρατηρείται ότι οι τιμές των μετρικών RMSE, MAE είναι σχετικά χαμηλές στην αρχή της εκπαίδευσης και παρουσιάζουν αρκετή μεταβλητότητα στις πρώτες παρατηρήσεις. Με την αύξηση ωστόσο του αριθμού των δειγμάτων, οι τιμές σταθεροποιούνται. Αυτό δείχνει ότι το μοντέλο μαθαίνει καλά τα δεδομένα εκπαίδευσης και τείνει να μειώνει τα σφάλματα. Η μετρική R^2 δείχνει επίσης υψηλές τιμές και σταθεροποιείται στο 0.8 (αρκετά κοντά στο 1 και μακριά από το 0) καθώς αυξάνονται τα δεδομένα, κάτι που δείχνει ότι το μοντέλο εξηγεί καλά την ποικιλία στα δεδομένα εκπαίδευσης.

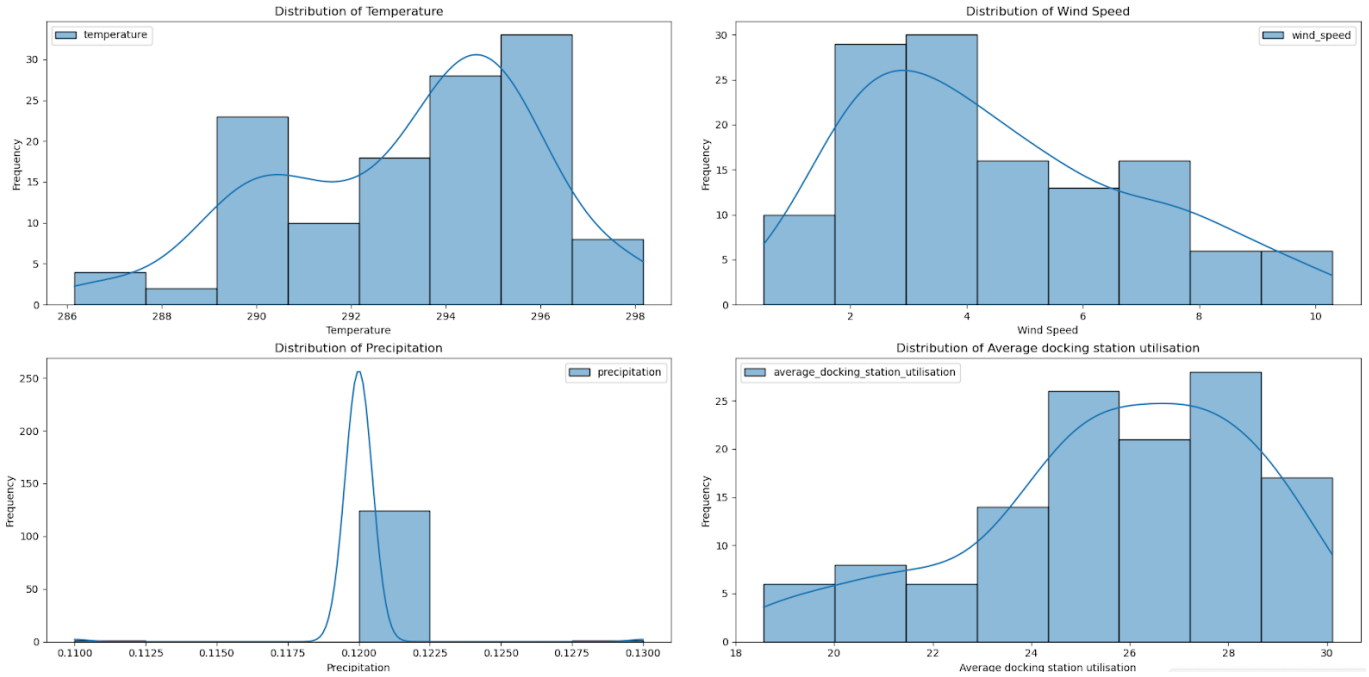
Με τον ίδιο ακριβώς τρόπο υπολογίζουμε και τις μετρικές για το validation set, από όπου λαμβάνουμε τις εξής γραφικές:



Οι μετρικές επικύρωσης (RMSE και MAE) ξεκινούν από υψηλές τιμές, αλλά σταδιακά μειώνονται καθώς αυξάνεται ο αριθμός των δειγμάτων επικύρωσης. Ωστόσο, η βραδεία μείωση και η σχετικά υψηλότερη τιμή των σφαλμάτων επικύρωσης σε σύγκριση με την εκπαίδευση υποδηλώνουν ότι το μοντέλο μπορεί να αντιμετωπίζει ελαφρύ **overfitting**. Από την άλλη, το R^2 μειώνεται σταθερά και παραμένει σε θετική τιμή (κοντά στο 0.5), δείχνοντας ότι το μοντέλο έχει λογική απόδοση στα δεδομένα επικύρωσης αλλά δεν είναι τόσο ισχυρό όσο στην εκπαίδευση.

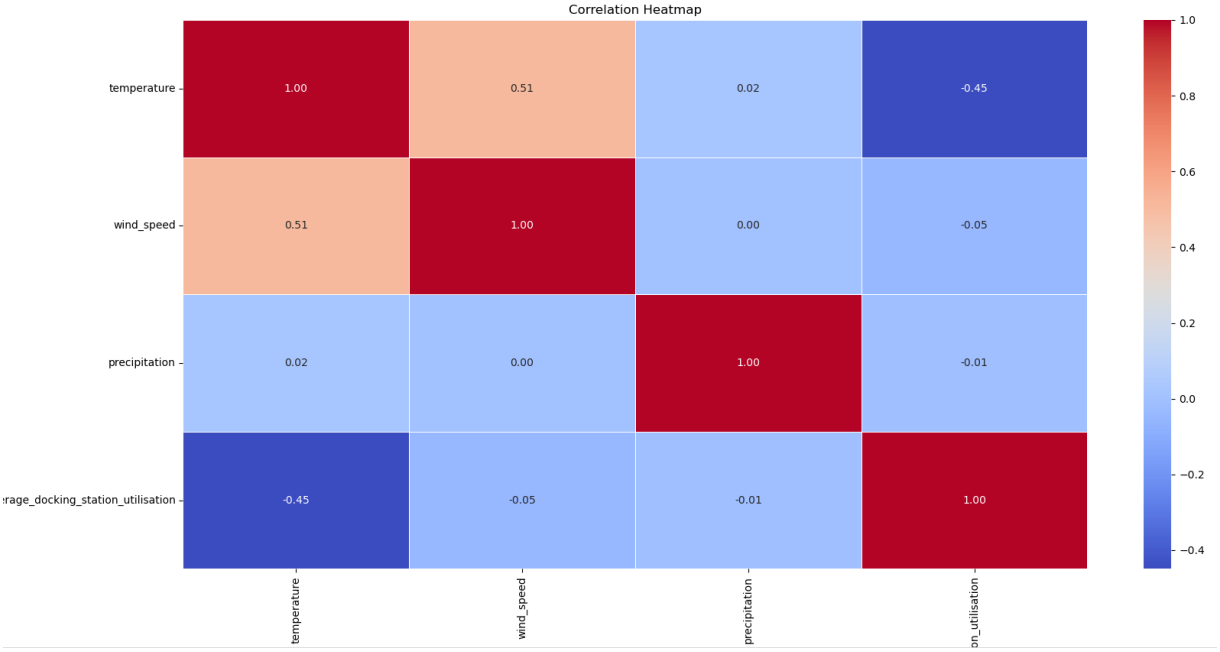
Συνεπώς, το Random Forest Regressor φαίνεται να μαθαίνει καλά τα δεδομένα εκπαίδευσης, ωστόσο η απόδοσή του όσον αφορά τη γενίκευση παρουσιάζει μια ελαφριά πτώση. **Αναμένεται δηλαδή η πρόβλεψη που θα πραγματοποιηθεί στη συνέχεια να είναι σχετικά καλή, αλλά όχι ιδανική.**

Επιπρόσθετα, γίνεται χρήση των **ιστογραμμάτων** (histograms) για την απεικόνιση των κατανομών των ανάμεσα στα τρία χαρακτηριστικά που ζητούνται **(temperature**, **precipitation**, **wind_speed**) καθώς και τον στόχο **average_docking_station_utilisation**, η οποία προσφέρει μια καλή οπτική του πως κατανέμονται τα δεδομένα.



Παρατηρούμε πως η κατανομή της θερμοκρασίας είναι σχετικά κανονική (Gaussian-like), με ελαφριά ασυμμετρία. Επίσης, εμφανίζει σημαντική διακύμανση, γεγονός που είναι θετικό και ιδιαίτερα χρήσιμο, καθώς δίνει τη δυνατότητα στο Random Forest Regressor να βρει συσχετίσεις με το target μας. Όσον αφορά τη ταχύτητα του ανέμου, η κατανομή είναι ελαφρώς ασύμμετρη προς τα δεξιά και η διακύμανση είναι επαρκής, αλλά τα δεδομένα συγκεντρώνονται περισσότερο σε χαμηλές τιμές ταχύτητας ανέμου. Συνεπώς, μπορεί να έχει μέτρια επίδραση στο target. Η κατανομή της βροχόπτωσης είναι εξαιρετικά συγκεντρωμένη γύρω από μία τιμή, γεγονός που υποδηλώνει χαμηλή διακύμανση. Επομένως το συγκεκριμένο χαρακτηριστικό θα έχει πιθανόν περιορισμένη συνεισφορά στην πρόβλεψη, καθώς δεν προσφέρει πολλές πληροφορίες στο μοντέλο. Τέλος, η κατανομή του target έχει καλή διακύμανση και συμμετρία, ενώ παράλληλα προσεγγίζει και τη κατανομή της θερμοκρασίας. Αυτό υποδηλώνει ότι η θερμοκρασία πιθανώς έχει ισχυρή συσχέτιση με τη χρήση των σταθμών, το οποίο είναι αναμενόμενο καθώς σε υψηλότερες θερμοκρασίες οι άνθρωποι τείνουν να χρησιμοποιούν περισσότερο τα ποδήλατα.

Τέλος, γίνεται και κατασκευή ενός heatmap, το οποίο δείχνει γραφικά την συσχέτιση ανάμεσα στα τρία χαρακτηριστικά που ζητούνται (**temperature**, **precipitation**, **wind_speed**) καθώς και τον στόχο **average_docking_station_utilisation**.



Από το heatmap των συσχετίσεων εξάγονται τα εξής συμπεράσματα:

- **Ανάμεσα στα features**: Υπάρχει μέτρια θετική συσχέτιση μεταξύ θερμοκρασίας και ταχύτητας ανέμου. Αυτό δείχνει ότι όταν η θερμοκρασία αυξάνεται, η ταχύτητα ανέμου τείνει να αυξάνεται επίσης. Η βροχόπτωση από την άλλη δεν παρουσιάζει σχεδόν καμία συσχέτιση με τη θερμοκρασία ή την ταχύτητα ανέμου. Αυτό υποδηλώνει ότι η βροχόπτωση είναι ανεξάρτητη από αυτά τα χαρακτηριστικά.
- **Ανάμεσα σε features και target**: Υπάρχει μέτρια αρνητική συσχέτιση ανάμεσα στη θερμοκρασία και το target. Αυτό σημαίνει ότι καθώς η θερμοκρασία αυξάνεται, η μέση χρήση των dock stations τείνει να μειώνεται. Επίσης, η συσχέτιση ανάμεσα σε ταχύτητα ανέμου και target είναι σχεδόν μηδενική, γεγονός το οποίο σημαίνει ότι η ταχύτητα ανέμου δεν έχει σημαντική επίδραση στη χρήση των σταθμών. Τέλος, η βροχόπτωση επίσης δεν έχει ουσιαστική συσχέτιση με τη χρήση των σταθμών.

Από τις παραπάνω δύο γραφικές είναι αναμενόμενο ότι το χαρακτηριστικό **temperature** θα επηρεάσει σημαντικά την τελική πρόβλεψη, ενώ τα υπόλοιπα δύο χαρακτηριστικά του καιρό το πιο πιθανόν είναι να συνεισφέρουν πολύ λιγότερη πληροφορία στο μοντέλο.

5.4 Πρόβλεψη

Αρχικά, εισάγονται από τον χρήστη τα ζητούμενα δεδομένα, δηλαδή η πόλη (**city_name**), η ώρα και ημερομηνία (**timestamp**), η θερμοκρασία (**temperature**), η ταχύτητα του ανέμου (wind_speed), η βροχόπτωση (**percipication**) και επιπλέον η συννεφιά (**cloudiness**). Έπειτα, ορίζεται το κατάλληλο schema για τα δεδομένα που εισήγαγε ο χρήστης, τα οποία εισάγονται στο PySpark DataFrame **next_hour_data**. Επίσης, υπολογίζονται τα επιπλέον χρονικά χαρακτηριστικά του Feature Engineering με βάση το **timestamp** που εισήγαγε ο χρήστης και προστίθενται οι επιπλέον στήλες στο ίδιο DataFrame. Η λίστα **next_hour_feature_columns** περιέχει όλες τις στήλες χαρακτηριστικών που θα χρησιμοποιηθούν για την πρόβλεψη και αφού τα χαρακτηριστικά αυτά ενωθούν στο διάνυσμα εισόδου μέσω του **VectorAssembler** τροφοδοτούνται στο μοντέλο. Το μοντέλο καλεί την **transform( )** πάνω στο διάνυσμα εισόδου και η τελική πρόβλεψη αποθηκεύεται στο DataFrame **next_hour_prediction**, από το οποίο εξάγουμε τη στήλη **prediction** και την εκτυπώνουμε.

Εν τέλει, ο χρήστης εισάγει τα παρακάτω δεδομένα στο μοντέλο και λαμβάνεται η εξής πρόβλεψη:

Enter city name: **Dubai**

Enter the date and time for prediction (YYYY-MM-DD HH:MM:SS): **2025-01-22 12:30:00**

Enter the temperature: **294.16**

Enter wind speed: **5.66**

Enter precipitation: **0.11**

Enter cloudiness: **20**

Predicted utilization for the next hour (2025-01-22 12:30:00) is: **25.56854893555339**

6. Προκλήσεις

Επειδή υπήρχαν δυσκολίες στον ορισμό του σχήματος του JSON που επιστρεφόταν από το API του καιρού, τα δεδομένα λαμβάνονται απευθείας από το JSON που στέλνεται από τον Kafka. Αντίθετα, για τα topics **station_status** και **station_info** δημιουργούνται κατάλληλα σχήματα με βάση τα αντίστοιχα JSON, έτσι ώστε να ορίσουμε τη δομή του DataFrame που θα χρησιμοποιηθεί.

7. Πιθανές τροποποιήσεις

Το ML θα μπορούσε να κάνει real-time πρόβλεψη ενσωματώνοντας το εκπαιδευμένο μοντέλο στο pipeline επεξεργασίας δεδομένων σε πραγματικό χρόνο που βασίζεται στο Apache Spark. Τα τελικά δεδομένα θα συλλέγονται από τα Kafka topics και θα υποβάλλονται σε προεπεξεργασία μέσω Spark Structured Streaming. Αυτή η διαδικασία θα περιλαμβάνει την εφαρμογή των βημάτων προεπεξεργασίας που εφαρμόστηκε κατά την εκπαίδευση του μοντέλου, όπως δημιουργία χαρακτηριστικών για Feature Engineering (π.χ., ώρα της ημέρας, μέρα της εβδομάδας), κανονικοποίηση και καθαρισμός δεδομένων. Στη συνέχεια, το εκπαιδευμένο μοντέλο Random Forest θα φορτώνεται από την αποθηκευμένη του μορφή και θα εφαρμόζεται στα εισερχόμενα δεδομένα μέσω της μεθόδου **transform( )** του Spark MLlib. Οι προβλέψεις που παράγονται θα αποθηκεύονται σε Kafka topics ή real-time βάσεις δεδομένων όπως MongoDB για άμεση χρήση. Το pipeline μπορεί επίσης να προσαρμοστεί ώστε να καλεί το μοντέλο ανά προκαθορισμένα χρονικά διαστήματα ή σε απόκριση σε νέα δεδομένα, διασφαλίζοντας ότι οι προβλέψεις είναι πάντα ενημερωμένες.

8. Αναφορές

[1] https://techcommunity.microsoft.com/discussions/windows11/how-to-install-the-linux-windows-subsystem-in-windows-11/2701207

[2] https://kafka.apache.org/quickstart

[3] https://sandeepkattepogu.medium.com/python-spark-transformations-on-kafka-data-8a19b498b32c

[4] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[5] https://www.geeksforgeeks.org/random-forest-regression-in-python/