

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**ML-based Data Classification and Data
Aggregation on the Edge**

Erkan Karabulut

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**ML-based Data Classification and Data
Aggregation on the Edge**

**Edge-basierte Datenklassifikation und
Datenaggregation durch maschinelles
Lernen**

Author:	Erkan Karabulut
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Prof. Dr. Rute C. Sofia (fortiss GmbH)
Submission Date:	16.05.2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.05.2022

Erkan Karabulut

Acknowledgments

First of all, I would like to thank my advisor Professor Rute C. Sofia for supporting me in every step of the dissertation development and writing process. I was very lucky to work with her side by side and I've learned enormously about the world of Internet of Things and research in general in such a short time. This thesis would not have been possible without her help and support.

I would like to thank Professor Jörg Ott for accepting me as his thesis student, and guiding me during this journey. Without his support, I would not be able to reach this point.

My former colleague Nisrine Bnouhanna whom I worked for 1.5 years together was the person who inspired me to work on Internet of Things and to be a researcher since the beginning. It was highly educative and fun for me to do brainstorming with her. I sincerely appreciate her support and patience while teaching me things about research and Internet of Things.

Lastly, I would like to thank all of my colleagues in fortiss Industrial Internet of Things team, who have done whatever they can to help me when I am frustrated or stuck at some point. I consider myself extremely fortunate to be part of this wonderful team.

Abstract

A key challenge in the context of Internet of Things (IoT) that remains to be solved is the lack of interoperability between cyber-physical systems offered by different vendors. An ever increasing number of active IoT devices increases the interoperability problem, considering large-scale sensing infrastructures, where devices from different vendors are usually applied. In such environments, each vendor introduces its own platform to manage devices and data; therefore, integration of multiple platforms is always required with a heavy level of manual intervention. The required level of intervention can be reduced by considering standardised semantic models to describe sensors and data, that, if used by most vendors, can improve interoperability. Standards such as Web of Things (WoT) or One Data Model (OneDM) Semantic Definition Format (SDF) enable us to describe an IoT device semantically and are widely adopted to increase interoperability. Still, there are differences in terms of semantic information models applied by vendors, e.g., due to specific regulation across domains, or specific requirements from the vendor platform. For instance, sensors offered by different vendors and measuring a specific type of measurement data, e.g., temperature, often have a different attribute, e.g., "temp" and "temperature". Moreover, IoT services are also described by semantic models, often derived from standards, e.g., ETSI SAREF. A way to improve interoperability is to consider a semantic matchmaking approach that can, in a semi-automated way, provide a finer-grained matchmaking between IoT semantic descriptions, Thing Descriptions, and semantic IoT service descriptions, considering an ontological-based approach. In our proposal, each Thing Description (TD), after data cleaning (pre-processing), is categorized and matched onto ontology aggregation points (centroids of a graph), based on a centrality measure derived from semantic similarity. The output is an aggregated TD. When a service request is processed, the aggregated TD is matched to the service (matchmaking), and the averaged measurement is provided as result to the service. To better understand which type of matchmaking to consider, this dissertation evaluates three different semantic similarity algorithms for the proposed matchmaking process, on various far Edge devices. Evaluation results showed that semantic matchmaking can be performed in milliseconds on an average level Edge device, with a promising accuracy level even with a general purpose semantic dataset. The contributions of this dissertation are three-fold: i) specification of the software

Abstract

architecture for semantic matchmaking; ii) implementation of the proposed concept based on the fortiss TSMATCH open-source software and demonstrator (TRL6); iii) performance evaluation of a semantic text similarity approach and two Machine Learning (ML)-based algorithms to support the semantic matchmaking.

Contents

Acknowledgments	iii
Abstract	iv
List of Tables	viii
List of Figures	ix
1. Introduction	1
1.1. Motivation and Goals	1
1.2. Research Questions	3
1.3. Activities and Roadmap	3
1.4. Dissertation Scope	4
2. State of the Art	6
2.1. IoT Things Descriptions	6
2.2. Semantic Matchmaking Approaches	7
2.3. ML in far Edge devices	10
2.4. Tooling	13
2.4.1. Data Sets	13
2.4.2. TSMatch: Thing to Service Matching Middleware	14
2.4.3. Auxiliary Libraries and Other Tools	15
3. Use-case	16
4. Architectural Design	18
4.1. Setup phase	19
4.1.1. Ontology Interface	19
4.2. Runtime phase	22
4.2.1. Data Pre-processing	22
4.3. Data pre-processing	22
4.4. Semantic Matchmaking Algorithms	23
4.4.1. LEX-DB: Sentence Similarity	25
4.4.2. W2VEC: NLP and Word Embeddings	25

4.4.3. k-Means: Clustering	28
4.5. Data Aggregation	29
5. Implementation	31
5.1. Use-case Setup: fortiss IIoT Lab	31
5.1.1. Hardware Equipment	31
5.1.2. Software	31
5.2. Ontology Interface	32
5.3. TD to Ontology Matching	33
5.4. Data Aggregation	33
5.5. Data Flow	34
6. Performance Evaluation	37
6.1. Evaluation Plan and Experimental Settings	37
6.2. Datasets	37
6.2.1. TD Dataset	37
6.2.2. Ontology Dataset	38
6.2.3. Training and Testing Datasets	38
6.2.4. Baseline Testing Dataset	39
6.2.5. Word Vector Training Dataset	39
6.3. Results, Performance Comparison of Similarity Approaches	40
6.3.1. Similarity Threshold Impact on W2VEC	40
6.3.2. Similarity Threshold Impact on K-Means	46
6.3.3. Threshold Impact Analysis Summary	49
6.3.4. Algorithm Accuracy Analysis	50
6.4. Node Usage Analysis	55
7. Key Findings	58
8. Conclusions and Future Work	60
A. Bibliography	61
B. TD to Ontology Element Matching Algorithm	64
C. Source Code Documentation	66

List of Tables

2.1. List of the analysed semantic matchmaking related work.	8
2.2. Comparison of ML/DL frameworks and libraries	12
4.1. Coverage of word vectors of different sizes.	26
6.1. Size and accuracy comparison for word vector subsets of different sizes.	40
6.2. Settings for each tested algorithm.	41
6.3. Impact, key and value threshold, accuracy of W2VEC-300k.	42
6.4. Impact of key and value threshold finer.grain values on the accuracy of W2VEC-300k.	43
6.5. Impact, key and value thresholds, W2VEC-1k.	44
6.6. Impact, key and value thresholds, finer-grained approach, W2VEC-1k.	45
6.7. Impact, key and value thresholds, K-MEANS-300k accuracy.	46
6.8. Impact, key and value thresholds finer-grained values, K-MEANS-300k accuracy.	47
6.9. Impact, key and value thresholds, K-MEANS-1k accuracy.	48
6.10 Impact, key and value thresholds, finer-grained values, K-MEANS-1k accuracy.	49
6.11 Summary, best performing key and value thresholds.	49
6.12 Accuracy of LEX-DB.	50
6.13 Accuracy of W2VEC.	51
6.14 Accuracy of W2VEC-300k.	51
6.15 Accuracy of W2VEC-1K.	51
6.16 Accuracy of K-MEANS-TD.	51
6.17 Accuracy of K-MEANS-300K.	52
6.18 Accuracy of K-MEANS-1K.	52
6.19 Hardware details and operating system of each testing device.	55
6.20 Node usage analysis results for TESTING1.	56
6.21 Node usage analysis for C-TESTING.	57

List of Figures

1.1. A simple example of the IoT interoperability issue in a smart facility. . .	2
1.2. A Gantt chart showing the estimated start and end time for each of the activities.	4
2.1. Components of the Things to Service Matching (TSMatch) demonstrator in fortiss IIoT Lab. Image is taken from [6]	13
2.2. Partial graph representation of the FIESTA-IoT Ontology that shows some of the category elements for quantity kind.	14
3.1. A smart facility with various sensing devices and the semantic match-making engine deployed on the edge.	17
4.1. The proposed architecture diagram together with other components of the TSMatch	18
4.2. A diagram showing the ontology data import process.	20
4.3. A visualization of FIESTA-IoT ontology nodes showing the aggregation (centralized) points. Visualization is created using Neo4j browser. . . .	21
4.4. A diagram showing the data pre-processing steps.	23
4.5. Part of a TD from [25].	24
4.6. An example service request that includes elements from FIESTA-IoT ontology.	30
5.1. A diagram showing the messages exchanged during a new ontology import.	35
5.2. A diagram showing the messages exchanged when a new IoT device is discovered.	36
5.3. A diagram showing the messages exchanged when a new service request arrived.	36
6.1. Accuracy for category QK.	52
6.2. Accuracy for category Unit.	53
6.3. Accuracy for category SD.	53
6.4. Total accuracy.	54

1. Introduction

1.1. Motivation and Goals

The number of deployed IoT devices has already reached the level of tens of billions and it is expected to be around 30 billion by 2025 ¹. IoT devices are data producers that are interconnected to IoT platforms, where then raw data is analysed and processed. The integration of these devices is often done in proprietary platforms and therefore, the semantic description of the devices and of their measurement attributes is often based on vendor models, and specific vendor-based ontologies. In large-scale sensing environments, there are usually sensors and IoT platforms that are provided by different vendors. Interoperability usually requires a high degree of human intervention and therefore, over the last decades, there has been a continuous effort to achieve interoperability at a protocol level and also at a device description level. This is supported by semantic technologies and standards.

For the specific aspect of semantically describing devices, the WoT architecture proposes a Things Description (TD)², i.e., a semantic representation of devices. Via semantic description standards, each vendor can specify aspects such as type of data, measurement unit. However, the reality is that vendors still rely on specific semantic models (also known as information models) to describe devices.

Following the attempt to create a higher degree of automation in IoT systems, IoT services (see definition 1.1.1) provided by IoT platforms (e.g., environmental monitoring) can also be described by semantic models that vendors provide.

Definition 1.1.1 (IoT Service). A piece of software that uses data from an IoT sensing infrastructure in order to perform a specific operation or set of operations, e.g., a service that outputs temperature per room using temperature measurements from sensors placed in each room.

Figure 1.1 provides a high-level illustration of this situation. There are 2 temperature sensors that are defined differently as "temperature sensor" and "temp". The first sensor measures environment temperature in room A and the second one

¹<https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/>

²<https://www.w3.org/TR/wot-thing-description/>

measures temperature of an industrial machine that is placed in Room B. The IoT platform, that collects raw data and processes it to provide answers to service requests, isn't capable of understanding which specific sensor to rely upon, unless this configuration is manually provided. This is inefficient, error prone, and eventually unfeasible, assuming that there are thousands of sensors and ever-changing service requirements. Another example is the contact sensor which can be used for different purposes according to where it is placed, e.g., on a door or window. The measurement units or sensitivity of measurements can also cause an interoperability issue.

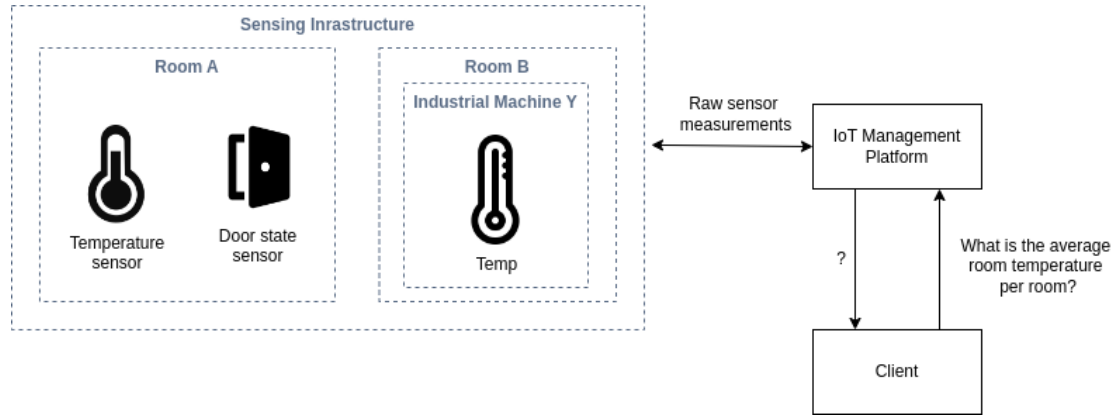


Figure 1.1.: A simple example of the IoT interoperability issue in a smart facility.

Adding to this problem, there are multiple semantic standards that can be applied. Besides WoT TD, other semantic standards quite used to described IoT devices are the Open Geospatial Consortium (OGC) SensorThings API Sensing part³ and the OneDM SDF⁴. Therefore, different vendors apply different standards.

A potential way to reduce the interoperability problem is to consider semi-automated ways to perform matchmaking of IoT Things Descriptions to IoT service semantic descriptions, considering a way also to integrate any existing ontology (see definition 1.1.2), and not just ontologies that are vendor-based, or domain-based.

Definition 1.1.2 (Ontology - W3C). An ontology defines the terms used to describe and represent an area of knowledge.⁵

This is the core focus of the proposed dissertation. This work assumes that each IoT device and IoT services can be semantically described. It also assumes that ontologies are available and can be used as a part of the semi-automated matchmaking process,

³<https://docs.ogc.org/is/18-088/18-088.html>

⁴<https://onedm.org/sdflanguage/>

⁵<https://www.w3.org/TR/webont-req/#onto-def>

to allow performing matchmaking between the IoT device descriptions and the service descriptions.

The dissertation proposes, implements, and assesses (based upon an existing testbed, Technology Readiness level 6) the proposed semi-automated matchmaking concept between IoT Things and service descriptions.

1.2. Research Questions

The need for a semi-automated matchmaking algorithm, and which solutions may provide the best performance can be formulated based on the following research questions:

- RQ1: Which functional blocks are required to support a semi-automated matchmaking process between IoT TDs and service descriptions?
- RQ2: How to define similarity thresholds that are adequate for the semantic matchmaking process?
- RQ3: Which approaches can be employed to support an ontology-based semantic matchmaking process, and can ML improve the semantic matchmaking?
- RQ4: Which approaches can be employed to support an ontology-based semantic matchmaking process, and can ML improve the semantic matchmaking in comparison to the usual word similarity approach? For how much?

1.3. Activities and Roadmap

A list of activities that are performed to answer the research questions given in previous section including the conceptual development, implementation and evaluation of the proposed approach are given below together with a Gantt chart in figure 1.2:

1. Analysis of state-of-the-art literature: Investigate semantic similarity metrics, select a subset of them based on a set of predefined criterion and compare ML-based approaches (accuracy and also node usage aspects, and open-source support) to be considered during the dissertation development.
2. Semantic matchmaking concept development: Develop a concept that uses the previously selected similarity heuristics and tools to do semantic matchmaking of IoT Things to services.

3. Implementation: Implement the proposed concept on the fortiss open-source IIoT Lab, within the tool TSMATCH (Thing to Service Matchmaking).
4. Evaluation: Evaluate the selected approaches based on matchmaking accuracy, running time and peak memory usage.
5. Documentation: Document the source code and write the dissertation.

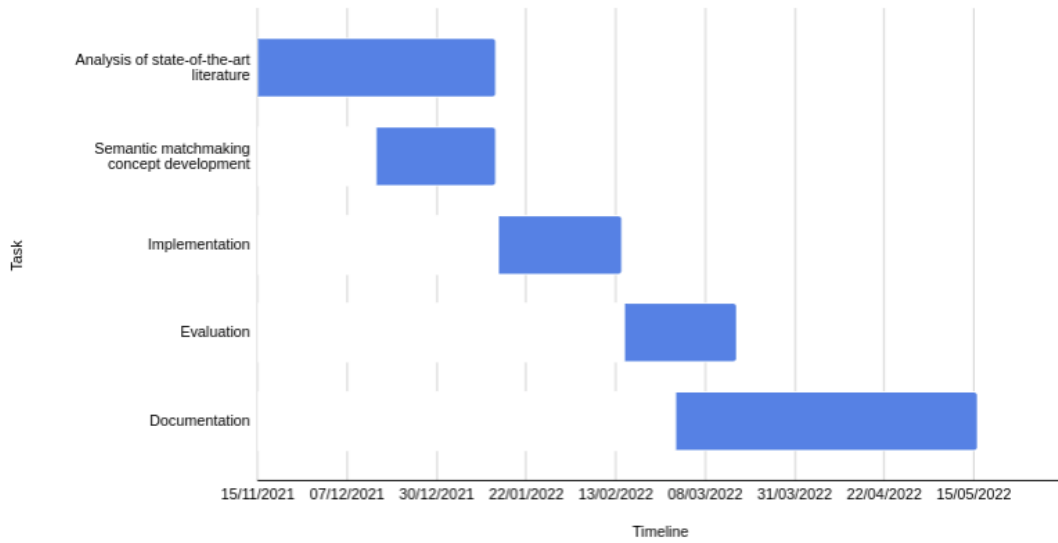


Figure 1.2.: A Gantt chart showing the estimated start and end time for each of the activities.

1.4. Dissertation Scope

The remainder of this dissertation is organized as follows. Chapter 2 describes the analysis of state of the art that has been carried out to provide initial answers to the research questions; assess which proposals are being used to support semantic matchmaking; analyse tools to use in the dissertation. Chapter 3 describes an exemplar use-case that allowed us to derive assumptions and requirements, and better define the development of the dissertation. Chapter 4.1 describes the semantic matchmaking mechanism architecture proposed, its functional blocks and interfaces. Chapter 5 is dedicated to the implementation of the proposed software architecture.

Chapter 6 describes the performance evaluation carried out via experiments developed on a realistic testbed (TRL6). Chapter 7 explains the results achieved deriving key findings. Chapter 8 concludes the dissertation and proposes directions for future work. Annex 1 provides a description of the algorithm to match IoT TDs to ontologies, while Annex C provides the source code structure.

2. State of the Art

During the first phase of work, and in order to look for some answers concerning the proposed research questions, a first activity carried out related with an analysis of related work on the following areas:

- IoT Descriptions.
- Semantic Matchmaking approaches.
- ML algorithms to support semantic matchmaking on Edge devices
- Tooling.

2.1. IoT Things Descriptions

As described before, IoT devices and services recur to semantic descriptions to improve interoperability.

Definition 2.1.1 (Web of Things Thing Description (WoT TD)). An information model by World Wide Web (W3C) that describes an IoT device, its attributes, meta-data, interfaces and overall properties. The WoT TD format can be used together with iotschema as optional semantic markup¹ and also quite flexible. It allows TDs to be enriched, if required, with additional meta-data.

WoT Profile enables creating mechanisms that aims to obtain out-of-the-box interoperability among things and devices² that fit to a same profile. As part of a Profile, an information model can be defined with additional limitations such as mandatory fields, constraints, data types and formats [11].

OGC SensorThings API³ provides sensing functionality that describes various sensing entities and their properties. SDF⁴ is another description format that aims to increase interoperability in and across IoT networks.

¹<https://iot.mozilla.org/schemas/>

²<https://www.w3.org/TR/wot-profile/>

³<https://docs.ogc.org/is/18-088/18-088.html>

⁴<https://onedm.org/sdflanguage/>

WoT Web Thing Model (WTM)⁵ defines a model and set of requirements for a Web API that needs be satisfied when using WoT in order to increase interoperability among Things at communication protocol level. Some such requirements are supporting GET, POST, PUT and DELETE HTTP verbs or implementing WebSocket protocol. There are many open source implementations that satisfy the given requirements [24] by enabling retrieval and update of TDs, their properties and models operations.

2.2. Semantic Matchmaking Approaches

During this analysis of related work the aim was to understand which approaches are being applied to perform semantic matchmaking between IoT TDs and IoT services. We have analysed several related work, listed in Table 2.1.

The related work analysed show that 4 different approaches have been frequently in terms of IoT semantic matchmaking: i) ontology-based; ii) clustering-based; iii) statistics-based; iv) supervised learning-based. The first category of work (ontology-based) takes ontology data (or a set of classes) as input and matches or labels a given data point to one of the elements in the ontology using rule-based functions or queries that are developed specifically for the elements of that ontology [22] [15]. In case there are data points which are described based on another ontology, or the characteristics of the data isn't covered within currently used ontology than this approach will fail to do a matchmaking. Therefore it is strictly dependant on an ontology and also requires manual work each time a new term or a notion is introduced in the system. The proposed solution should work with any given ontology and therefore this approach isn't suitable.

The second category of work (clustering-based) creates clusters that consists of semantically similar elements [28] [20] [23]. When a new data point has detected, it is matched to the semantically closest cluster. This approach requires a training process per ontology or set of output classes. However, it can be accomplished without any source code changes or manual intervention and also no labelled data required for training. Assuming that an ontology change won't occur frequently, this approach can be a good candidate for semantic matchmaking of Things to services.

The third approach (statistical) matches texts that are semantically closer to each other. Notion of similarity can be obtained using various statistical approaches such as Jaccard similarity [8] and term frequency-inverse document frequency or also using a lexical database [12]. This approach doesn't include a learning process and it can be implemented without depending on an ontology.

⁵<https://www.w3.org/Submission/wot-model/>

Table 2.1.: List of the analysed semantic matchmaking related work.

Title	Type	Matching
A Hybrid Semantic Matchmaker for IoT Services [7]	Ontology-based	IoT services to requests
Semantic Matchmaking for Job Recruitment: An Ontology-Based Hybrid Approach [9]	Ontology-based	Job seekers to job postings
A Machine-Learning Approach for Semantic Matching of Building Codes and Building Information Models (BIMs) for Supporting Automated Code Checking [26]	Supervised learning-based	Building Codes to Building Information Models (BIMs)
Semantic Matching Across Heterogeneous Data Sources [28]	Cluster analysis for db schemas, supervised learning for instance-level matching	Data source matching, e.g. matching of 2 relational db
Ontology Matching: A Machine Learning Approach [8]	Supervised learning-based	Different ontologies in a single domain
Short Text Clustering Enhanced by Semantic Matching Model [20]	Clustering	Query to documents on (Short texts is social media) dataset
The k-Means Clustering Algorithm With Semantic Similarity To Estimate The Cost of Hospitalization [23]	Clustering	Clustering of patients using a semantic similarity as distance based on an ontology
Sentence Similarity Based on Semantic Nets and Corpus Statistics [12]	Statistics-based	Semantic meaning of different sentences
Machine learning in the Internet of Things: A semantic-enhanced approach [22]	Ontology-based	Sensor data to ontology elements

The last approach (supervised learning) doesn't fit large-scale sensing use-cases due to having unknown amount of different information models or ontologies. It requires a training process with all possible output classes which can't be known beforehand in this type of use-cases. As an example, assume that a model is trained using a supervised approach. When a new IoT sensing device that is described using a different information model is introduced to the system, the model wouldn't be able to understand the notions that exists only in the new information model.

As a result of this analysis, the following 3 different approaches that don't rely on a single ontology or requires labelled data have been selected to be implemented and evaluated: i) a statistical approach, based on sentence similarity; ii) a Natural Language Processing (NLP) neural network-based approach; iii) a clustering-based approach. The first approach is based on [12] and doesn't include any learning process. It uses the WordNet [18] lexical database to create a semantic vector representation of a sentence and also considers the order of the words in a sentence. Since there is no learning process, it can be implemented without being dependant on any information model or ontology. It focuses directly on computing the similarity between sentences and it can also be used short texts or phrases. Values in a TD are usually a couple of words long, or in the case of a "description" field it can be as long as a couple of sentences. Therefore this approach is a good candidate for semantic matchmaking of IoT things to services.

The second approach is based on NLP and uses a neural network model to learn word associations from a large corpus of text [16, 17]. In this context, Mikolov et al. introduced Continuous Bag-of-Words Model (CBOW) and Continuous Skip-gram Model (Skip-gram) that are used to create vector representations of words. The prior predicts a word based on the context it is used in while the latter predicts a set of words before and after a given word. The first model fits better to the described scenario since the goal is to predict output classes based on the context, e.g. match a given TD to an output class with the name "Temperature".

The third approach follows a clustering approach and is based on the k-Means algorithm. k-Means is a well known clustering method that is also used in semantic matchmaking [23]. Clustering-based algorithms have the drawback that the total number of clusters to be considered need to be defined beforehand, so for a semantic matchmaking process it is necessary to assess how that number can be obtained. For the proposed approach, the number of clusters will be the amount of sample data, e.g. category (see definition 2.2.1) elements, for an ontology node.

Definition 2.2.1 (Category). A category in an ontology refers to a list of possible values for an ontology element. E.g., "Temperature", "Humidity" and "Air Quality" are some of possible values for "Quantity Kind" ontology element in FIESTA-IoT ontology.

Then k-Means algorithm will match TDs to possible category elements. The algorithm requires a notion of distance between clusters and data points. Neural networks are used to create vector representations of text data as described previously. Cosine distance between vectors will be used to calculate distance of a data point to a cluster and while calculating the centroids (see definition 2.2.2). Clusters will include only the matching phrases from the neural network approach and not the whole TD. The centroids will be the average vector representations of the matching phrases. In this way, an average understanding of how people would describe a certain aspect for a given ontology is represented in a fully autonomous way.

Definition 2.2.2 (Centroid). A centroid of a cluster is the average values of all data points in that cluster.

2.3. ML in far Edge devices

The purpose of having a better grasp on sensor data is eventually to facilitate the data aggregation (see definition 2.3.1) process. Sensor data aggregation can be performed on the Cloud or on the Edge. When pushed closer to the end-user, if performed on "far Edge" devices (for instance, end-user devices, or devices that are directly connected to end-user devices), the resulting data aggregation process will result in lower latency and lower energy consumption. Continuous sensor data streaming from and to a centralized point such as a device in the Cloud is known to increase the overall latency and also requires more bandwidth. Edge computing helps overcome this by bringing computation closer to the data source. It also improves security of the network since the data don't have to leave the network.

Definition 2.3.1 (Data aggregation). Representing a set of data points in a smaller and summarized form.

On the other hand, in the case of intensive-computation applications, such as data processing and data analysis services, the resources provided by Edge devices may not be sufficient. Several related work focuses on the support to run computationally heavy tasks on Edge devices. Zhang et al. [27] evaluates different Deep Learning (DL) frameworks based on latency, memory footprint, and energy consumption for a computer vision task. In this paper, different implementations of a large-scale DL model and a small-scale DL model have been run on various devices; MacBook Pro, Intel's FogNode, NVIDIA Jetson TX2, RPI 3 B+, Huawei Nexus 6P. Luo et al. [14] have measured the inference ability of Android devices to classify images based on different DL frameworks. They have run 6 different image classification models on 5

different Android devices using Tensorflow Lite⁶, Pytorch Mobile⁷ and Caffe2⁸ which is now part of Pytorch as well.

However, we weren't able to find a study where performance of far Edge devices are compared, based on an IoT semantic matchmaking process. Therefore, a performance comparison of different devices based on the running time and memory usage is performed (see 6).

In order to implement the ML-based algorithms mentioned in the previous section, an analysis and comparison of the existing ML/DL tools is performed. The sentence similarity approach doesn't require a training process and for the clustering approach a customized version of the k-Means algorithm is used. Therefore, the tool to be selected will be used to train a CBOW model only. The comparison of the tools are based on the platform they can operate in, support for different programming languages, hardware acceleration capability, model optimization (e.g., quantization, pruning or clustering), open-source code availability, on-device training capability (e.g. is it possible to do training on a constrained device), federated learning capability and algorithms they can run. Table 2.2 shows the comparison results for Tensorflow Lite⁹, PyTorch Mobile¹⁰, Apache MXNet¹¹, ELL¹², and Gensim¹³ [21].

Even though TensorFlow and PyTorch supports federated learning, this feature is not yet available on TensorFlow Lite and PyTorch Mobile. The same situation applies to on-device training with PyTorch Mobile. Training process requires data from not only a small local network but a larger network that can provide TDs written using different information models. Collecting such a huge data from different sources and then processing it in a constrained device would be inefficient unless it is performed on separate constraint devices simultaneously using federated learning paradigm. For the sake of simplicity and also due to lack of data on-device training and federated learning features are neglected while selecting a tool. All of the tools except ELL provides an implementation for word2vec. Gensim provides an easy implementation of word2vec algorithms and some components of the TSMatch is already written in Python. Therefore the Gensim library is preferred.

⁶<https://www.tensorflow.org/lite>

⁷<https://pytorch.org/mobile/home/>

⁸<https://caffe2.ai/>

⁹<https://www.tensorflow.org/lite/>

¹⁰<https://pytorch.org/mobile/home/>

¹¹<https://mxnet.apache.org/versions/1.9.0/>

¹²<https://microsoft.github.io/ELL/>

¹³<https://radimrehurek.com/gensim/>

2. State of the Art

Table 2.2.: Comparison of ML/DL frameworks and libraries

Framework / Library	TensorFlow Lite	PyTorch Mobile	Apaxhe MXNet	ELL	Gensim
Platform	Android, iOS, Embedded Linux (ARM64) Microcontrollers	Android, iOS	Linux, MacOS, Windows, Cloud, RPI, NVIDIA Jetson	Windows, Ubuntu, MacOS, RPI, Arduino, micro:bit	Any platform that supports Python and NumPy
Language	Java, Swift, Objective-C, C++, Python	Python, C++, Java	Python, Scala, Java, Clojure, R, Julia, Perl, C++	Python, C++	Python
Hardware Acceleration	+	+	+	-	-
Model Optimization	+	+	+	-	+
Open Source	+	+	+	+	+
On-device Training	+	-	+	-	-
Federated Learning	-	-	+	-	-
Algorithms	computer vision tasks, pose estimation, question answering, text classification ...	computer vision tasks, neural machine translation, question answering, vision transformers ...	computer vision tasks, word embeddings, language model, machine translation, sentiment analysis ...	Image classification, object detection, audio keyword spotting	Natural Language Processing, Information Retrieval

2.4. Tooling

2.4.1. Data Sets

Agarwal et al. have developed the FIESTA-IoT Ontology [1] that aims to ease sensor observation data exchange between testbeds that integrate or interact with IoT devices. Categorizations [2] for 4 different aspects have also been developed and published together with the ontology [3] by the same authors; quantity kind, measurement unit, domain of interest and sensing device. A partial graph visualization of the published ontology obtained using the Neo4j browser¹⁴ is shown in Figure 2.2.

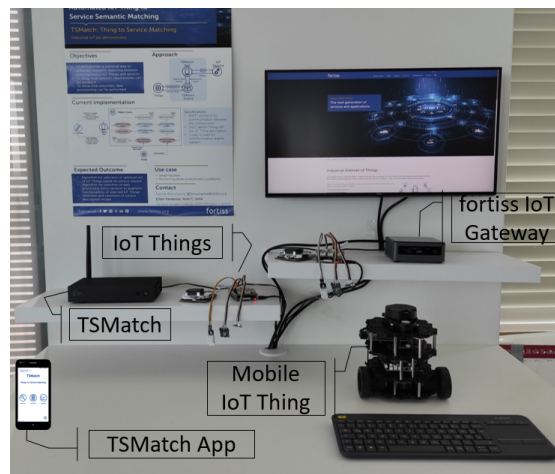


Figure 2.1.: Components of the TSMATCH demonstrator in fortiss IIoT Lab. Image is taken from [6]

The FIESTA-IoT Ontology mentioned above is created by benefiting multiple domains and therefore is a good representation of a real-life scenario. The concepts introduced in this dissertation will be demonstrated and evaluated on the FIESTA-IoT Ontology due to its wide use, but can be applied on any given ontology or even in multiple ontologies.

WordNet¹⁵ is a lexical database for English language. Besides being a dictionary and containing a vast list of synonyms, WordNet also provides a hierarchical structure of words. As an example, a "teacher" is a "person" and a "person" is a "living organism" and a "living organism" is eventually an entity. WordNet is a lexical database [12], widely used due to the features it provides. We use WordNet to train the statistical similarity approach.

¹⁴<https://neo4j.com/developer/neo4j-browser/>

¹⁵<https://wordnet.princeton.edu/>

document for semantic matchmaking [6]. Based on the matching, a set of available Things are selected, then grouped and an aggregated object representing this grouping is again stored in a TSMatch database. TSMatch has been applied in experimental pilots (TRL6) across the H2020²⁰ European Connected Factory Platform for Agile Manufacturing (EFPF) project²¹, and a demonstrator is available and interconnected to the EFPF data spine via the fortiss IIoT Lab²². The best ML-based approach analysed in this dissertation will be included in the TSMatch open-source software release v2.0.

2.4.3. Auxiliary Libraries and Other Tools

Natural Language Toolkit (NLTK)²³ is an open-source Python natural language processing library. It contains several lexical resources including the WordNet lexical database and also provides text processing functionalities. These functionalities are needed to implement data pre-processing steps, cf. to Chapter 4 for details. All of the data pre-processing steps can be implemented using NLTK including tokenization and lemmatization.

Eclipse Paho MQTT client library²⁴ is an open-source MQ Telemetry Transport (MQTT) client implementation for Python language. It provides all of the functionalities needed to communicate with an MQTT broker, such as connect, disconnect, publish and subscribe and very easy to use. Paho MQTT client is used to communicate with the MQTT broker (see chapter 4 for details) in all of the modules written in Python.

NumPy²⁵ is an open-source easy-to-use Python package that provides scientific computation functions. This library is utilized whenever a vector operation is required, e.g. calculating distance between two word vectors.

Finally, the official Neo4j Python driver²⁶ is used to realize graph db operations. It can be used to send db queries and receive replies. This library is used in every module that is written in Python and requires to connect the graph db.

²⁰https://ec.europa.eu/info/research-and-innovation/funding/funding-opportunities/funding-programmes-and-open-calls/horizon-europe_en

²¹<https://www.efpf.org/>

²²<https://www.fortiss.org/en/research/fortiss-labs/detail/iiot-lab>

²³<https://www.nltk.org/>

²⁴<https://www.eclipse.org/paho/>

²⁵<https://numpy.org/>

²⁶<https://neo4j.com/docs/api/python-driver/current/>

3. Use-case

This section describes a generic use-case which helped to dimension the development of the proposal, by considering assumptions and requirements; actors involved.

The use-case is based on the fortiss TSMatch demonstrator, which is currently being applied in IIoT environments, and which is also being used as a pilot service with different SMEs in the context of the EFPF project, as described in section 2.4.2.

This use-case considers the notion of a smart factory, where today there are multiple sensors integrated into the different factory environments, e.g., shop-floor, warehouse, and is represented in Figure 3.1. There are machines from different vendors, with coupled sensors attached on them. Moreover, sensors are also used to monitor the environment, e.g., CO₂, temperature, humidity. Employees of this facility are using wearable devices, tablets and smart phones which also have sensing capabilities. In this scenario, different IoT platforms have been acquired to different vendors. Therefore, each platform considers different semantic standards to support an interoperable data exchange. Data exchange is supported by a data bus across the factory, and the different platforms rely on specific communication protocols to exchange data, e.g., OPC UA, MQTT Sparkplug. Different services, e.g., data analytics tooling, environmental monitoring services, certification services, are interconnected to the data spine via software-based connectors that have been specifically devised for this purpose, by the different vendors, or by an integrator. Some of these services run on the so-called Edge (close to the field-level devices, e.g., on the shop-floor) and others run on the Cloud.

On this scenario, the semantic matchmaking process can occur on the Cloud, or on the Edge. Placing the matchmaking on the Edge is expected to lower latency, and to also reduce energy consumption, as most of the data processing (including aggregation) is performed closer to the end-user.

For this scenario, the following assumptions are considered in the context of the dissertation:

- semantic matchmaking occurs on the Edge, to reduce latency.
- Any IoT devices are represented by a standardised Things Description.
- Any IoT service is represented by a standardised semantic description.

3. Use-case

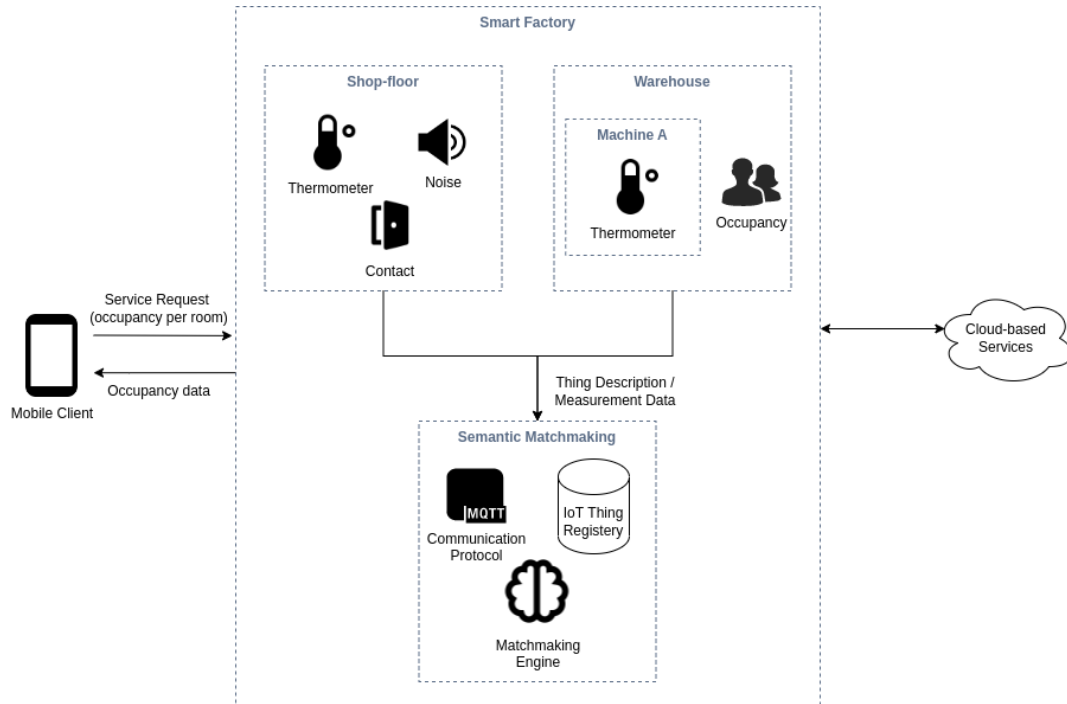


Figure 3.1.: A smart facility with various sensing devices and the semantic match-making engine deployed on the edge.

- A set of ontologies can be used, to improve the matchmaking.
- The IoT device discovery is handled by an existing process.

On this scenario, the semantic matchmaking engine TSMATCH is installed locally on the Edge. Sensors are automatically discovered via the coaty.io framework, that TSMATCH employs. The TSMATCH client is used by the end-user to monitor the environment.

4. Architectural Design

The proposed concept is based on the fortiss TSMatch concept and therefore, the functional blocks of the architecture are based on the overall TSMatch design, and the functional blocks highlighted in blue correspond to the functional blocks of the developed concept.

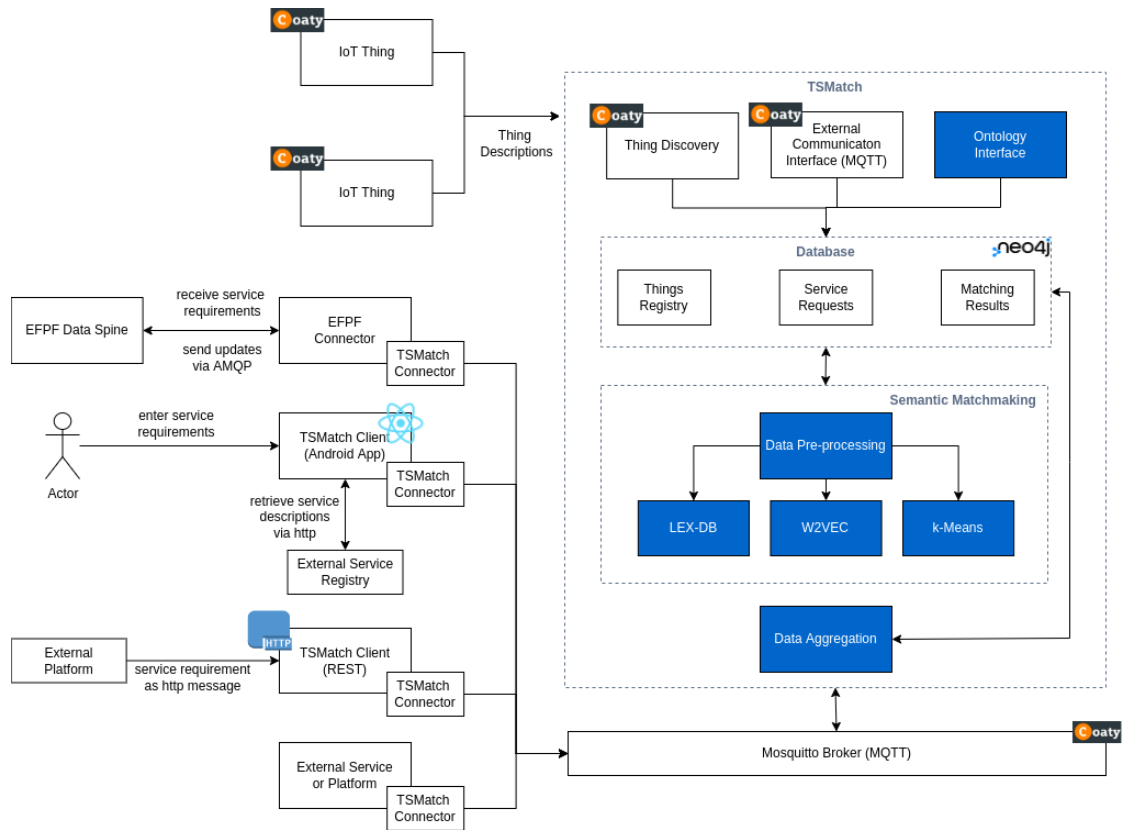


Figure 4.1.: The proposed architecture diagram together with other components of the TSMatch

TSMatch v1.0 is therefore middleware that relies on the coaty.io framework to discover IoT Things. IoT TDs are stored on the TSMatch Things Registry. IoT Things

are interconnected via MQTT (Mosquitto) with coaty.io.

A graph database implemented in Neo4j stores IoT TDs, service requests, Match-making results.

Multiple connectors have been developed to allow interoperability of TSMatch with different platforms. For instance, the EFPF connector corresponds to a MQTT-Advanced Message Queuing Protocol (AMQP) connector. A REST connector is used to interconnection with external service platforms.

The design of the proposed semantic matchmaking process considers the following functional blocks (blue):

- Ontology interface.
- Data pre-processing.
- Semantic matchmaking, where the three different approaches selected as discussed in section 2.2 have been implemented: i) statistical approach based on cosine similarity (LEX-DB); ii) NLP neural-network model approach (W2VEC), iii) clustering-based approach (k-Means).
- Data Aggregation.

The proposed mechanism runs in 2 different phases. During setup, an ontology is imported into the TSMatch via the ontology interface module. During runtime, both the TD dataset(s) and the ontology dataset(s) are pre-processed, and then passed to the semantic matchmaking functional block, to be handled by one of the three proposed algorithms. The algorithm then matches TDs to the ontology centroids (aggregation points). Service requests are captured by the data aggregation module and matched to the aggregated TD, so an aggregated (averaged) value is provided to the service.

4.1. Setup phase

As discussed in chapter 2, TSMatch performs the semantic matchmaking based on any given ontology. In order to import ontology data into the TSMatch, a module named **ontology interface** has been developed.

4.1.1. Ontology Interface

The ontology interface provides a REST API that allows importing an ontology into the graph database. Currently, it accepts both JSON and OWL files. When a new ontology import is triggered, the following steps are executed in order:

4. Architectural Design

- Delete matching of sensors to ontology elements in the graph database.
- Delete currently used ontology nodes.
- Convert the given ontology to JSON if necessary.
- Create new nodes and edges that corresponds to the classes and relations in the given ontology.
- Find and mark aggregation (centralized) nodes (see definition 4.1.1) if not already given inside the HTTP request.
- Trigger TD to ontology matching service to match available sensors to the new ontology elements.

An illustration of the process is shown in Figure 4.2.

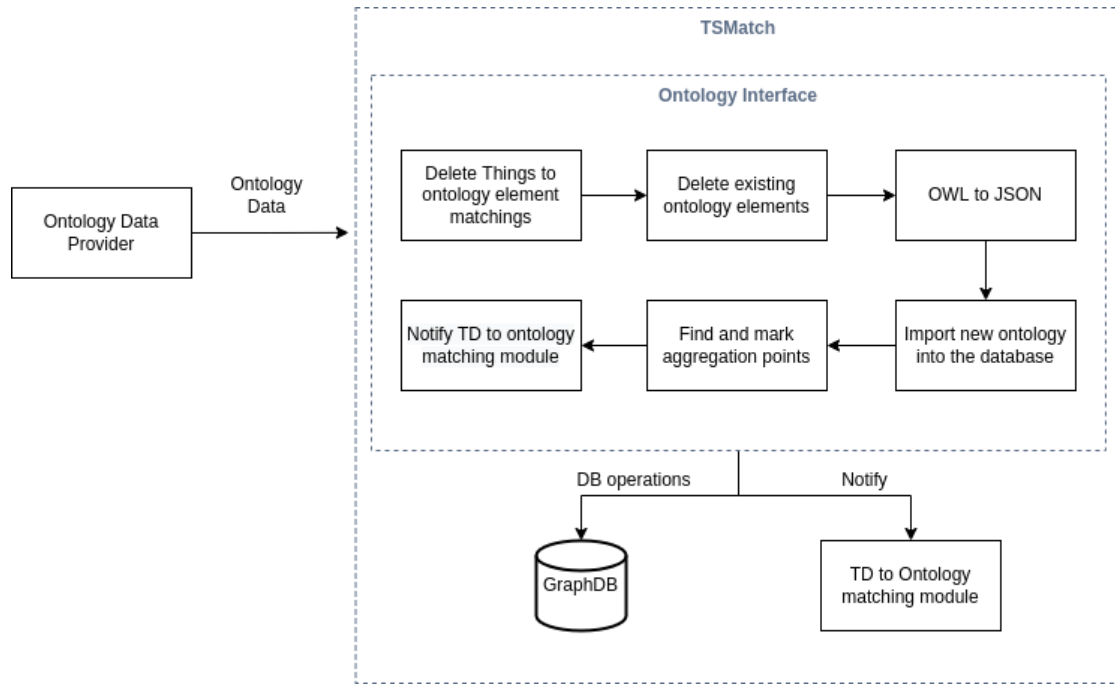


Figure 4.2.: A diagram showing the ontology data import process.

Definition 4.1.1 (Aggregation Point). It refers to the centralized nodes, nodes with excessively high number of neighbors, in an ontology data. Child nodes of an aggregation point, e.g. sub-classes in an ontology, represents possible values for that aggregation point.

Aggregation points refer to centroids of the graph that defines an ontology. This is based on the assumption that aggregation points have distinctively more neighbors than the other ontology elements since they refer to a list of possible values, e.g., a category. Figure 4.3 shows a partial visualization of FIESTA-IoT ontology nodes. There are 3 highly centralized nodes with a high number of neighbors that corresponds to "QuantityKind", "Unit" and "SensingDevice" ontology elements. A given TD will be matched to one of the child nodes for each of the aggregation points. An example matching would be; "QuantityKind": "Temperature", "Unit": "DegreeCelsius" and "SensingDevice": "Thermometer".

In case these aggregation points aren't given in the request to the ontology interface, then they can be found by running an anomaly detection method on the number of neighbors that each node has. The proposed implementation includes a simple anomaly detection method that is based on the empirical rule¹. According to the empirical rule, 99.7 percent of the values are within a range that corresponds to 3 times the standard deviation.

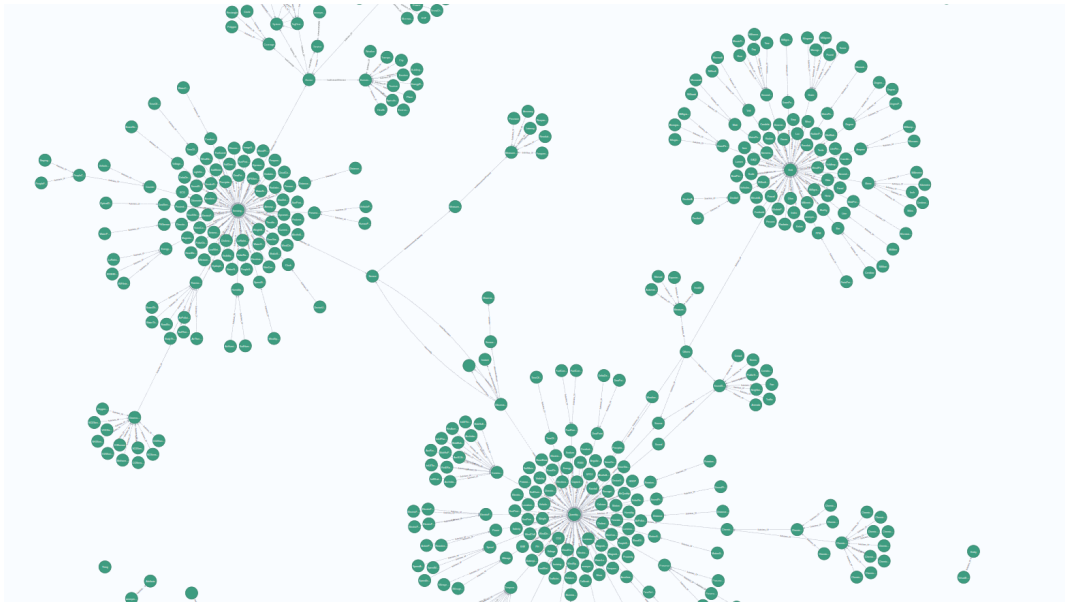


Figure 4.3.: A visualization of FIESTA-IoT ontology nodes showing the aggregation (centralized) points. Visualization is created using Neo4j browser.

¹https://en.wikipedia.org/wiki/68-95-99.7_rule

4.2. Runtime phase

4.2.1. Data Pre-processing

The data pre-processing functional block in the proposed architecture comprises well-known text pre-processing steps that have been described in section 4.3. Pre-processed TDs are passed to one of the 3 selected semantic matchmaking algorithms. The semantic matchmaking approach matches the given TD to ontology elements that are given by the user. Then it creates an edge in the graphdb between the matched ontology nodes and the TD. The Ontology interface can be used to import any ontologies into TSMATCH (rf. to section 4.1.1).

4.3. Data pre-processing

The data pre-processing module consists of well-known pre-processing steps that are frequently used in Natural Language Processing (NLP). It includes the following steps which are illustrated in figure 4.4:

- Tokenization: Separate text into sentences.
- Remove punctuations.
- Fix camelCase: Some words in the TDs are written in camelCase² format. A word written in this format might not exist in the word2vec model word list. Therefore they need to be separated into multiple words, e.g., "camelCase" to "camel case".
- Lowercasing.
- Remove stopwords: Python NLTK Toolkit[5] includes a list of stopwords in English. Remove those words from the TDs.
- Lemmatization: Find stem of the words using "WordNetLemmatizer" from the NLTK toolkit.
- Remove Trivial Words: Some words that appear commonly in TDs create a non-realistic similarity, i.e., the word "sensor". As an example, the CBOW model produces a higher similarity score for "temperature sensor" and "luminance sensor" pair than "temperature" and "luminance".

²https://en.wikipedia.org/wiki/Camel_case

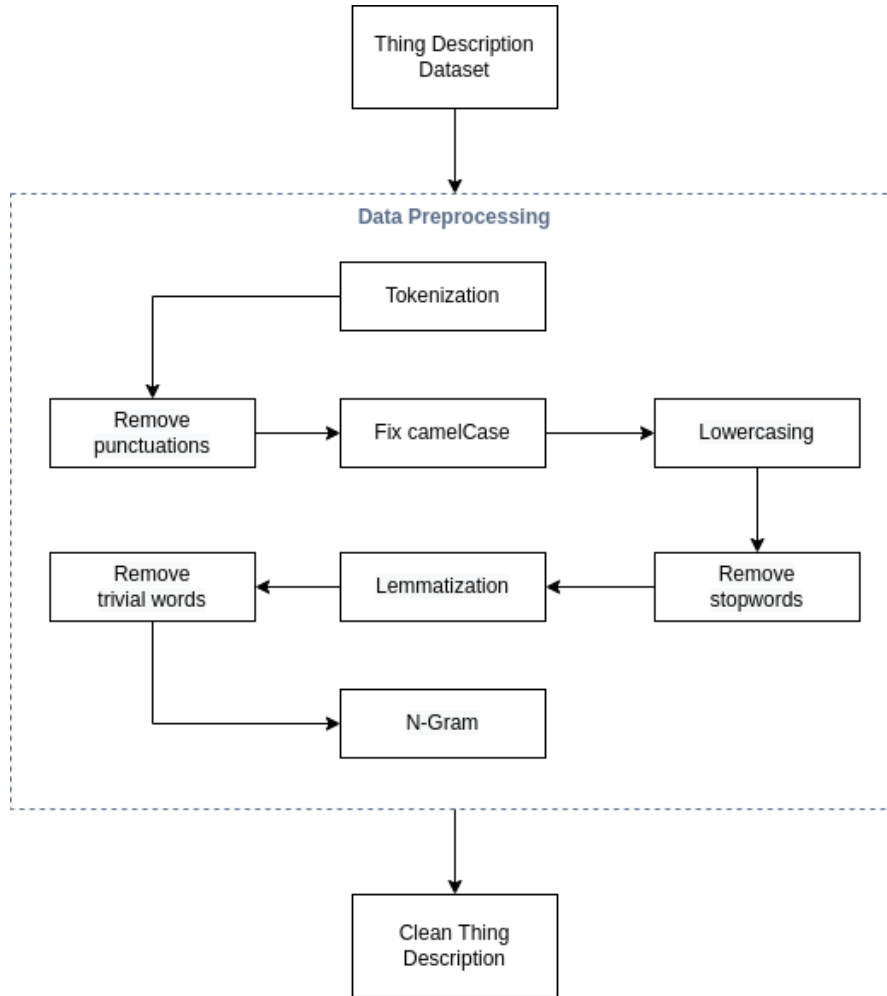


Figure 4.4.: A diagram showing the data pre-processing steps.

After the pre-processing step, the TDs are passed to one of the semantic similarity algorithms. The algorithm matches the given TD to the ontology elements. Then a relation is created in the graph database between the TD and the ontology nodes that it matched to.

4.4. Semantic Matchmaking Algorithms

This section defines how TD to ontology element matchmaking is performed using the following 3 semantic similarity algorithms. Besides small differences which are

described in individual sections next, the matchmaking process is given in Algorithm 1 and 2. To better illustrate the concept developed to match TDs using semantic matchmaking algorithms, figure 4.5 shows a part of a TD where a sensor is defined.

```

...
"sensor":{
  "name":"LightIntensity",
  "description":"Light intensity in Lux",
  "type":"object",
  "readOnly":true,
  "title":"LightIntensity",
  "properties":{
    "LightIntensity":{
      "type":"number",
      "readOnly":true
    }
  },
  "forms":[
    {
      "href":"https://w3ctest.iadstg.iot.ocs.oraclecloud.com/iot/api/v2/apps/0-AB/devices/-
113972C0-6CFC-4FBD-A586-87E7FFD385F1/deviceModels/urn%3Acom%3Aoracle%3Asolarpanel/attributes/-
LightIntensity",
      "contentType":"application/json"
    }
  ],
  "uuid":"51509897-9b2a-4b88-9b39-7f1cf19f8be1"
}
...

```

Figure 4.5.: Part of a TD from [25].

Some attributes in a TD appear in different information models. Common terms are "name", "description" or "title" that exist in WoT TD, OGC SensorThings API Sensing part, and OneDM SDF. These fields include valuable information regarding the type of the sensor, its measurement unit for instance. As an example, the sample TD shared in Figure 4.5 has name and description fields where it mentions the ontology "quantity kind" (Light intensity) and measurement unit (Lux) of a sensor. Therefore, the algorithm initially checks if a matching can be found using these fields in line 9. "ap["name"]" refers to the name of an aggregation point, e.g., "QuantityKind". "ap["category"]" refers to the children of an aggregation point, e.g., ["Temperature", "AirQuality", "Humidity", ...].

Each collected TD has one or more sensor description. Secondly, the algorithm checks if a matching can be found using these sensor descriptions in line 11. If this is also unsuccessful, then it checks the remaining parts of the TD to find a matching to the aggregation points that are extracted earlier in line 13.

Algorithm 2 in Annex B describes how the semantic similarity approaches are used while matching a TD to child nodes of an aggregation point. The method named "similarity" refers to one of the 3 semantic similarity algorithms. First of all, the algorithm tries to find a match between keys in a TD and aggregation points

between lines 4 and 10. For instance, it checks if there is a key that is similar to "QuantityKind" or "Unit". Each of the semantic similarity algorithm produces a similarity score between 0 and 1. In case the similarity score is higher than a `KEY_SIMILARITY_THRESHOLD`, then it is accepted as a matching key and consider the value of that KEY only in the next step.

In case there is an attribute that has a higher similarity to the given ontology aggregator point name, then the algorithm tries to find a matching value to the children of that aggregation point between lines 11 and 24. If there is no matching key then the algorithm checks similarity of each value in the given TD to the children of an aggregation point.

4.4.1. LEX-DB: Sentence Similarity

The applied sentence similarity approach LEX-DB is based the work by Li et al. and an open-source implementation of their approach[12, 10] is used for semantic matchmaking. It produces a similarity score between 0 and 1 using semantic and syntactic information contained in the given pair of texts. Initially, the algorithm creates raw semantic vectors and word order vectors for the sentences with the assistance of the lexical database WordNet (rf. to section 2). Contribution from each word to the meaning of a sentence is marked by assigning it a weight value using a text corpus and then the weights are combined with the raw semantic vector. Finally, an overall semantic similarity score is calculated using a weighted word order similarity and semantic similarity. Li et al. argue that syntax has more effect on the semantic processing of a text and hence the authors use higher weight for syntactic similarity than the word order similarity.

4.4.2. W2VEC: NLP and Word Embeddings

Word embeddings refers to vector representations of words in terms of real numbers. Our W2VEC approach is based on the work by Mikolov et al. where the authors introduce CBOW [16, 17] (rf. to section 2). As mentioned in 2.2, CBOW approach is implemented to train word vectors from a TD dataset. Word vectors are usually trained using GBs of corpus data. Since the training data size is in the order of megabytes, word vectors published by Google are used as well [4].

However, Google published 3 million vectors with 300 features that takes around 6 GBs of space when it is extracted into a text file. Keeping all of the word vectors in the memory wouldn't be efficient. Two options were considered; i) dimensionality reduction, ii) using a subset of word vectors instead of the entire dataset. The original paper where the CBOW method is described, different dimensions including 300 are

compared. Results show that further increasing the dimension size doesn't increase the accuracy significantly. However, reducing the number of dimensions causes a significant decrease in the accuracy. Therefore the dimensionality reduction option is eliminated.

A second option to circumvent this problem is to consider a subset of vectors. The vector list published by Google is ordered based on the popularity of each word. This option is based on the assumption that the most popular X words will appear in a TD very frequently and the effect of the remaining words will be negligible. In order for the justification of this assumption the following experiment is carried out: Subsets of different sizes from the word vectors published by Google is extracted. Then a percentage value representing the ratio of words that are both in the collected TD dataset and also in the extracted subset to all words in the TD dataset is calculated. Results are given in Table 4.1.

Table 4.1.: Coverage of word vectors of different sizes.

Subset size	Coverage 1	Coverage 2
1,000	20.22	30.18
2,000	31.89	50
5,000	52.52	70.4
10,000	67.28	86.2
20,000	79.68	91.95
50,000	91.25	96.93
100,000	95.62	97.99
200,000	97.94	98.85
300,000	98.35	99.46

Definition for the Coverage 1 and 2 are given in definition 4.4.1. Using subsets with different sizes also effects the amount of space required and accuracy of the semantic matchmaking. Table 6.1 in section 6.1, shows how much space is required to store each subset and also how they effect the accuracy of semantic matchmaking.

Definition 4.4.1 (Coverage). Ratio of words in the TD dataset that are also inside the word vector subset

t = list of words that are in TD dataset.

ws = list of words that are in word vector subset.

unique(a) = return a list of unique words (remove duplicates) in list a.

|a| = size of the list a.

$$\text{Coverage 1} = \frac{|t \cap ws|}{|ws|}, \text{ Coverage 2} = \frac{|unique(t \cap ws)|}{|ws|}$$

According to the results in Table 4.1, size of the subset exceeds 1 GB when approximately 270,000 words are selected. Coverage starts to increase very slowly after taking the most popular 100,000 words. Since the coverage values aren't increasing considerably after exceeding a certain point, we have opted to take a subset of word vectors instead of using all 3 millions of them.

Cosine similarity approach is used while calculating similarity of an ontological element to a value field in a TD. In order to obtain a single vector for a text field that consists of multiple words, an average vector is calculated. However, while comparing two texts with different sizes, e.g. 2 words and 10 words, average vector representation might not reflect the actual similarity even the 2 words inside the first text appear in the second text. As an example, consider the following 2 short texts: "temperature sensor" and "a highly sensitive temperature sensor with model number X from company Y". Even though it can be inferred that that these two texts describes a "temperature sensor", the extra words in the second sentence ("high", "sensitive", "model", "number", "X", "company", "Y") changes the average representation of the sentence and hence when put on a vector space, it gets further away from the vector that represents "temperature sensor" only.

To overcome this, the n-gram method with a dynamic n value is utilized. In the n-gram method, the words that appear together in a long text are put in the same set and such multiple sets of words created for a single text. After removing the stopwords and lemmatization, an N-gram representation for the second text given in the previous paragraph with an N value of 3 looks as follows:

- (high, sensitive, temperature)
- (sensitive, temperature, sensor)
- (temperature, sensor, model)
- (sensor, model, number)
- (model, number, X)
- (number, X, company)
- (number, company, Y)

The words "temperature" and "sensor" appear in the sets 2 and 3 with an additional word. In this case, comparing one of these sets with the first text "temperature sensor" would produce a higher similarity score than comparing the whole text. Elements in a category, e.g. child nodes of an aggregation point are compared to values in a TD. Therefore the value that is being searched is the name of the category node which can consist of multiple words. Value of N should be at least as high as the number of words in the category node name so that a matching value in the TD can be captured. If the value of N is much bigger than the category node name, then this would result in the same situation that is given in the previous example with the temperature sensors. Since the length of category node names vary, e.g. 1 word "Temperature", 2 words "Air Quality", 3 words "Volatile Organic Compound", picking a static number for N is ineffective. Therefore the number N is dynamically selected in a way that it will be equal to the length of category node name to obtain an exact matching.

The following steps are used for calculating semantic similarity between an aggregation point name or category node name and a key or value in a TD:

- assign the word count in a category node name or aggregation point name (output class name in short) to N
- if N is smaller than 2, then assign 2 to N
- calculate average vector representation for the output class name
- separate a given key or value in a TD into multiple sets based on the value of N
- calculate average vector representations for each set
- calculate cosine similarity between the vector that represents the output class and vectors that represent each set
- find the set with the highest cosine similarity
- if the cosine similarity is higher than a threshold then accept it as a match

4.4.3. k-Means: Clustering

In the k-Means approach, a set of clusters per aggregation point is created. Each set contains clusters for all possible elements, e.g. all elements in a category. For the FIESTA-IoT ontology, this means that there are 4 sets of clusters for "QuantityKind", "Unit", "SensingDevice" and "DomainOfInterest", each containing a list of clusters

that corresponds to the category elements. As an example to "QuantityKind" aggregation point, there are one cluster per "Temperature", "Air Quality", "Humidity" etc.

As stated in chapter 2, a customized version of the well-known k-Means algorithm has been considered as the clustering method to be applied. k-Means requires the total number of clusters to be provided. In the original version of the algorithm, different number of clusters are evaluated to find the right amount of clusters that can best represent the variety among the data points. One common way to select initial centroids for each cluster is to pick data points that have the highest betweenness centrality value. Then the algorithm assigns data points to the nearest cluster. After this initialization phase, centroids for each cluster are calculated again, this time by taking the average values of data points in that cluster. Then data points are reassigned to the closest clusters. This final step is repeated for a predefined amount of time or until no data point is assigned to a new cluster.

The variation that has been considered in this dissertation considers that the number of clusters is equal to the total number of aggregation points obtained from the used ontology/ies.

In the used FIESTA ontology, there are 178 different classes defined for "QuantityKind", which correspond to the ontology aggregators (cluster centroids). Hence, in total the algorithm considers 178 clusters. Same logic applies for other aggregation points as well. Initially, the cluster centroids will be the vectorized version of the class names inside each aggregation point. Vectors are obtained from the previously described word embeddings approach. As a notion of distance between data points or centroids, cosine distance is used since the data is in vector format. Each cluster contains a list of phrases that were found similar to the centroid of that cluster. Iteration step is same as the original version with a maximum iteration count of 10.

4.5. Data Aggregation

Data aggregation is performed based on the assumption that every IoT service can be described semantically according to an ontology. This assumption suggests that a service request will include elements from an ontology. Upon receiving a service request, data aggregation module looks for matching sensing devices according to the ontological elements inside the request. Then the engine subscribes to data from those sensors and aggregate using a simple average function.

As an example, figure 4.6 shows a sample service request which includes category elements from the FIESTA-IoT ontology. Data aggregation module searches for sensor descriptions in the graphdb that has a relation to "Illuminance", "Lux", "LightSensor"

```
{  
  "QuantityKind": "Illuminance",  
  "Unit": "Lux",  
  "SensingDevice": "LightSensor",  
  "DomainOfInterest": "Environment",  
  ...  
}
```

Figure 4.6.: An example service request that includes elements from FIESTA-IoT ontology.

and "Environment" nodes. It sends a response back to the requestor that contains a list of sensor and TDs. Lastly, the module subscribes to data from those IoT devices, aggregates data using an average function (to be improved) and continuously sends the aggregated data to the requestor until the service request is deleted.

5. Implementation

The architecture proposed and described in Chapter 4 has been implemented following a micro-service architecture. The source code is available via the fortiss git upon request¹ and also directly via this² link.

Implementation details are described next. Moreover, Annex C includes the source code file structure for each of the 3 modules developed.

5.1. Use-case Setup: fortiss IIoT Lab

5.1.1. Hardware Equipment

During the development and evaluation phases the code has been developed in the fortiss TSMATCH software and tested in the demonstrator. The testbed components that are utilized during this study are: i) Raspberry Pi 3B+ and Raspberry Pi 4B with 5 sensors attached to each, measuring temperature, humidity, CO2 concentration, particle size in the air and noise, ii) An Intel NUC NUC10i7FNH³ device that hosts dockerized TSMATCH Engine, the message broker and the graph database.

5.1.2. Software

The auxiliary software used are listed in this section.

Coaty

Coaty⁴ is a communication middleware that allows any to any communication between applications that uses Coaty. It is designed to facilitate various event-based communication patterns in a decentralized system. Coaty is built on top of interchangeable open-standard pub/sub messaging protocols, i.e. MQTT⁵. In TSMATCH,

¹https://git.fortiss.org/iiot/demonstrator2/-/tree/erkan/matching_improvement

²<https://gitlab.com/erkankarabulut/master-thesis-implementation>

³<https://www.intel.com/content/www/us/en/products/sku/188811/intel-nuc-10-performance-kit-nuc10i7fnh/specifications.html>

⁴<https://coaty.io/>

⁵<https://mqtt.org/>

Coaty is used to discover IoT devices in the system, notify different modules in case of an event and continuously publish or consume data to/from other external or internal modules.

Eclipse Mosquitto MQTT Broker

MQTT is a lightweight pub/sub based messaging protocol that is developed for IoT systems. Eclipse Mosquitto⁶ is an open source message broker that implements MQTT protocol and a part of the Eclipse Foundation⁷. Eclipse Mosquitto is used as the underlying messaging component of the Coaty.

Neo4j Graph Database

Neo4j graph database is an ACID compliance graph database that supports processing data in terabytes level. Ontology elements are stored in the database. When an IoT device is discovered a node in the graph containing the TD and one node per sensor descriptions that are inside the TD is created. Matching of ontology elements to sensor descriptions are shown as relations on the graph db. Thing and sensor descriptions together with all relations are deleted a when they are no longer available. Service requests are also stored in the graph db as separate nodes as long as the request is active and matching of service requests to sensor descriptions are also shown as relations

5.2. Ontology Interface

Ontology interface provides a REST API where ontology data can be imported. REST API is implemented using Django⁸ in Python and dockerized using a Python image version 3 as a base image. It accepts both a URL to the ontology data and also the data itself in either JSON or OWL format, see *Views* class under *ontology_interface/app/* folder. Lohmann et al.[13], provides multiple tools (e.g. OWL2VOWL or WebVOWL⁹) that can be used to convert OWL files into JSON formats. An executable jar file for OWL2VOWL tools is placed under *ontology_interface/ontology/converter/* directory. OWL2VOWL tool is integrated into the ontology interface in order to convert OWL files to JSON for ease of development. It runs on port 8080 and

⁶<https://mosquitto.org/>

⁷<https://www.eclipse.org/>

⁸<https://www.djangoproject.com/>

⁹<http://vowl.visualdataweb.org/webvowl.html>

accepts new ontologies on the path 'ontology/'. URL mapping can be found in *ontology_interface/web/urls.py* file.

The ontology interface is connected to the graph database and Eclipse Mosquitto MQTT broker. DB connection is established inside the *BaseRepository* class under *ontology_interface/app/repository* folder and an MQTT client is created under *ontology_interface/app/service* folder. For all the implemented modules, the class files that includes db related operations are in the *repository* folder and MQTT client is in the *service* folder. Upon receiving a request, it deletes the existing ontology data together with all the relations to the ontology nodes and imports the new ontology data. Afterwards it publishes a message over MQTT in order to notify TD to ontology matching module about the ontology change. This process is implemented in *Views* class.

5.3. TD to Ontology Matching

This module developed in Python and runs inside a docker container that uses Python version 3 as a base image. It is also connected to the graph database and the MQTT broker similar to the previous modules. Semantic similarity algorithms and data pre-processing steps are implemented inside this module as shown in Figure 4.1. Individual files for all 3 algorithms can be found under *td_to_ontology_matching/src/algorithm* folder. Data pre-processing classes are under *td_to_ontology_matching/src/preprocessing* folder.

The module communicates with others over the MQTT broker. When a new IoT device is discovered, its TD is published over a discovery topic. TD to Ontology matching module subscribes to discovery messages and matches newly discovered sensors to ontology elements. It also captures messages published by the ontology interface when a new ontology is imported. In this case, it extracts all available sensors in the graph database and matches them to the new ontology elements. This actual matching operation is implemented in *TDToOntologyMatching* class that is under *td_to_ontology_matching/src/service* directory.

5.4. Data Aggregation

Similar to the previously described modules, the data aggregation module is developed in Python and dockerized using Python version 3 as a base image. The data aggregation module is also connected to the graph database and communicates with other modules and outside world through the MQTT broker. Its duties are to receive service requests, find matching sensors in the graph database, subscribe to the data

from those sensors and aggregate the results. The results are then published back to the requestor.

Sensor observation and service request arrival events are handled in *ObservationEventHandler* and *ServiceRequestHandler* classes that are located under *data_aggregation/src/handler* directory. *data_aggregation/src/service/ServiceRequest* class keeps a list of active service requests and also a list of matched Things descriptions per service request.

5.5. Data Flow

This section describes the data flow between the modules in TSMATCH for a selected set of scenarios that includes all improvements made to the TSMATCH in the scope of this thesis.

Figure 5.1 shows the sequence of actions when a new ontology is imported into TSMATCH via the ontology interface. First, an external ontology data provider sends the ontology data using the REST API of the ontology interface via a POST request. The request body includes a JSON data with "url" or "data", and "type" keys. Type refers to the type of the ontology file which can be JSON or OWL. If the type of file is OWL, then it is converted to JSON using the OWL2VOWL tool mentioned in section 5.2. Since JSON files can be converted to "dict" type in Python and therefore easier to process. An ontology provider can either send the ontology data itself with the "data" key or send a URL where the ontology data is hosted. In the later case, data is downloaded and then converted to JSON if necessary.

As a second step, the ontology interface deletes the existing ontology data from the graph database together with the matchings of TDs to the ontology nodes. Then it stores the new ontology data. Next, it publishes a message on "fortiss-org.TSMATCH.NDATA.ontology_CHANGED" topic to notify TD to ontology matching module. This module then extracts available TDs and aggregation points from the graph database. It then runs the algorithm given in 1 in order to match TDs to new ontology elements using one of the similarity approaches from section 4.4.

Figure 5.2 shows the sequence of actions when a new IoT thing is discovered. An IoT device with a Coaty agent running on it publishes its TD via the MQTT broker on "fortiss-org.TSMATCH.NDATA.DISCOVERY" topic. TD to ontology matching module subscribes to the same topic and listens for discovery messages right after it starts. It also keeps a list of aggregation points and their child nodes in the memory at all times. This due to not sending a db query each time a matching is done. Upon receiving the new TD, the module matches it to ontology elements and stores the matching result in the graph database as relations between sensor descriptions and

5. Implementation

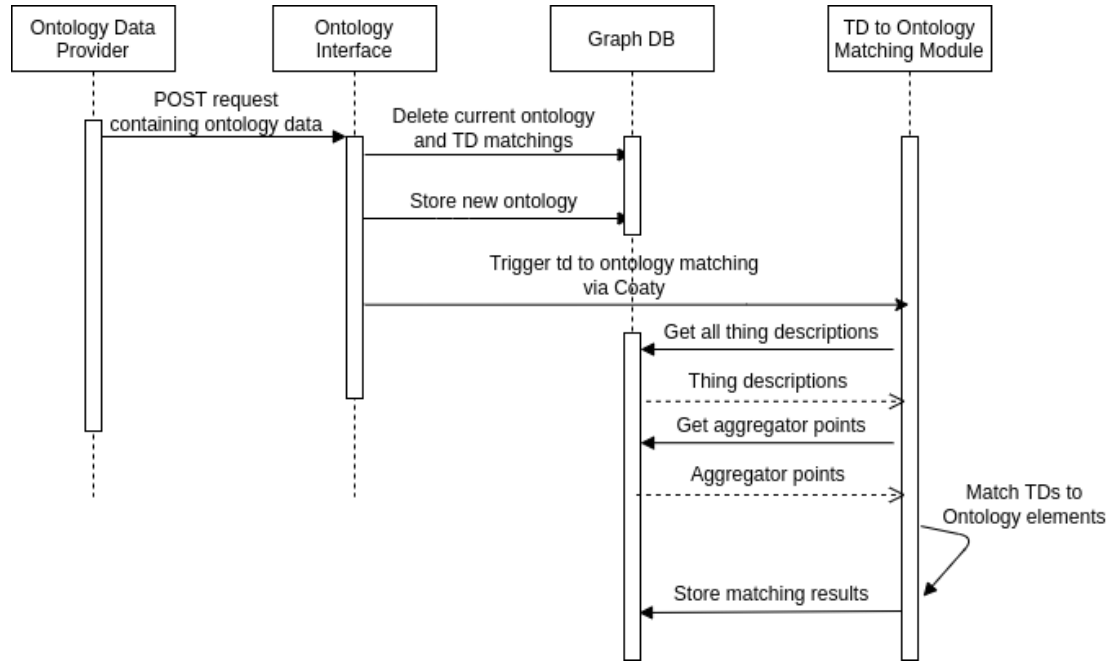


Figure 5.1.: A diagram showing the messages exchanged during a new ontology import.

ontology related nodes.

Figure 5.3 illustrates the sequence of actions after an IoT service request is received. A request can be sent via TSMATCH mobile client, REST client or directly on MQTT level. In order to capture the service requests, the data aggregation module subscribes to "fortiss-org.TSMATCH.NDATA.SERVICE_REQUEST" topic. As shown in Figure 4.6, each service request contains ontology elements. After receiving a request, it queries database to find IoT things that are matched to the given ontology elements. The list of IoT things is then returned to the client.

The aggregation module then subscribes to measurement data from those IoT devices and aggregates the data using a mean function. If the client initially sent the request via the REST interface, then it receives web socket connection details where the aggregated sensor observation data is continuously published. Otherwise the client receives data on the "fortiss-org.TSMATCH.NDATA.OBSERVATION" topic.

5. Implementation

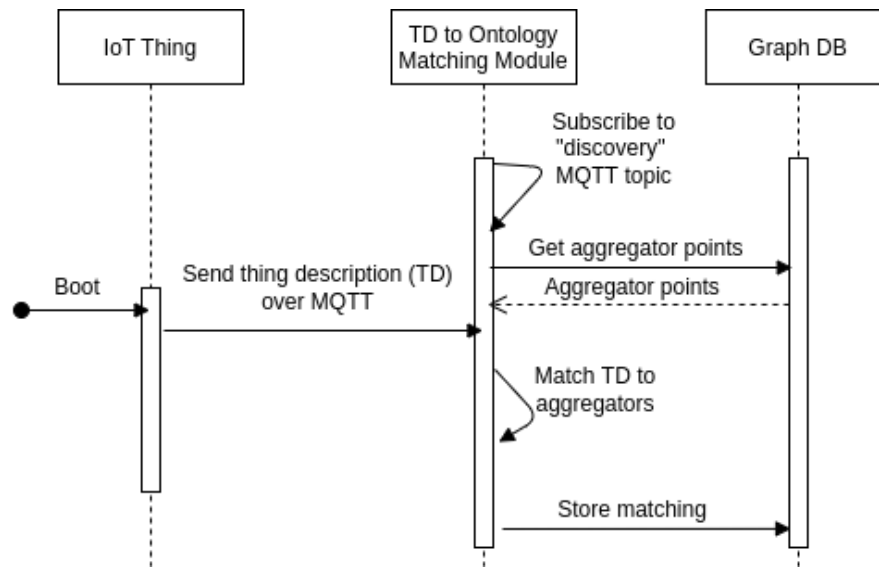


Figure 5.2.: A diagram showing the messages exchanged when a new IoT device is discovered.

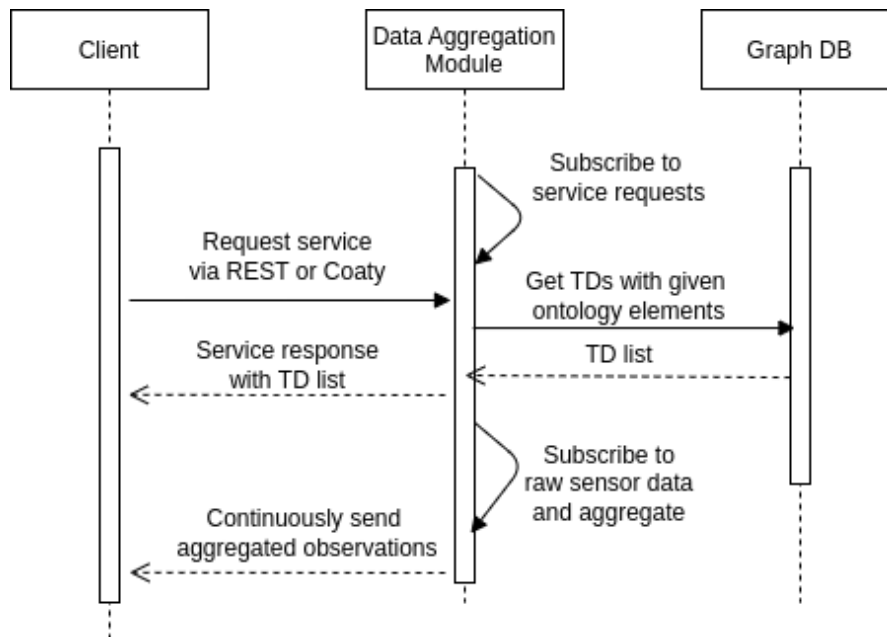


Figure 5.3.: A diagram showing the messages exchanged when a new service request arrived.

6. Performance Evaluation

6.1. Evaluation Plan and Experimental Settings

The aim of the performance evaluation is to analyse different semantic matchmaking algorithms, lexical, word2vec word embeddings and k-Means.

For this purpose, the following performance evaluation parameters have been considered:

- **accuracy**, defined as the semantic matchmaking accuracy of Things to service matching. Percentage of correct matchings to all matchings.
- **time to process** of a matchmaking, corresponding to the required running time measured on the device where TSMATCH resides, for matching TD to ontology elements
- **peak memory usage** during TD to ontology matching for all of the proposed approaches

The evaluation plan is to first create training and testing datasets. Then the selected semantic matchmaking algorithms will be trained using the training dataset, except lexical db-based approach since it doesn't require a training process. A baseline will be extracted by manually matching the testing dataset to ontology elements. Each of the semantic matchmaking algorithms will match the testing dataset to ontology elements and accuracy of the matchmaking will be calculated using the manually created baseline. This last step will be repeated on various devices in order to measure time to process and peak memory usage parameters.

6.2. Datasets

6.2.1. TD Dataset

The matchmaking requires datasets comprising heterogeneous TDs. For the purpose of analysis a large TD dataset has been created, after a vast search (see section 2). This dataset has been created based on the following 2 datasets:

- WoT Dataset composed of heterogeneous TDs¹, corresponding to sensor TD. This dataset comprises 327 files, each of which holds one or more than one sensor or actuator. There are 2191 sensor TDs in total[25].
- OneDM SDF TD dataset², composed of 200 sensor TDs[19].

Therefore, in total a sensor TD dataset (DAT1) comprising 2391 heterogeneous sensor TDs is created.

6.2.2. Ontology Dataset

As described in section 2.1, we have used the FIESTA-IoT Ontology as an example of an ontology, to perform the semantic matchmaking. FIESTA-IoT comprises 483 different class of entities including 178 category elements for "QuantityKind", 92 category elements for "Unit", 109 category elements for "SensingDevice" and 11 category elements for "DomainOfInterest" class.

6.2.3. Training and Testing Datasets

Out of DAT1, 2 testing and 1 training datasets have been built.

A first testing dataset, TESTING1³, consists of 200 sensor descriptions, roughly 10%, out of DAT1, that have been randomly selected based on a uniform distribution using the package from NumPy⁴. As described in section 6.2.4, the testing dataset is manually labeled based on the FIESTA-IoT ontology. Therefore, due to time constraints, instead of trying different testing-training split percentages, only 1 split (10%) option is considered. The remaining %90 percent of DAT1 dataset is used as the training dataset, TRAINING⁵, for Word2Vec and K-Means algorithms.

A second testing dataset (C-TESTING⁶) has been considered, still relying on 200 sensor descriptions, but removing the sections of the TD that are not applicable to the matching towards an ontology. For instance, the WoT TD attribute "created" or "modified" defines when the TD is created and modified. Therefore, it doesn't contribute to the semantic meaning while understanding which ontology element

¹<https://github.com/w3c/wot-testing/tree/main/events/2022.03.Online/TD>

²<https://github.com/one-data-model/playground/tree/master/sdfObject>

³https://git.fortiss.org/iiot/demonstrator2/-/tree/erkan/matching_improvement/td_to_ontology_matching/dataset/testing

⁴<https://numpy.org/>

⁵https://git.fortiss.org/iiot/demonstrator2/-/tree/erkan/matching_improvement/td_to_ontology_matching/dataset/training

⁶https://git.fortiss.org/iiot/demonstrator2/-/tree/erkan/matching_improvement/td_to_ontology_matching/dataset/testing_cleaned

does this description relates to. In C-TESTING, only the sections of the WoT TD that are applicable to the semantic matchmaking process are considered. The aim of this specific dataset is to evaluate potential improvements in terms of node usage, e.g., CPU, memory, and eventually, running time.

6.2.4. Baseline Testing Dataset

The performance evaluation that has been carried out considers the testing datasets (TESTING1, C-TESTING) as baseline⁷ which has been manually labelled. This dataset has been built based on human assessment of the categorisation of each sensor on the testing datasets against the aggregation points of the FIESTA ontology. Baseline dataset contains sparsity for "domain of interest" category due to very few TDs having domain related information. Therefore the "domain of interest" category is excluded from all of the accuracy measurements that are described in the next sections.

To exemplify how the labelling was done, consider a TD that has "humidity" as its title, a description section which mentions that this sensor is measuring relative humidity in a room and another field named "unit" and its value is "%". By looking at the first 2 fields, and to the QuantityKind category in FIESTA-IoT Ontology, one can infer that the TD can be linked to the FIESTA aggregation point "RelativeHumidity". Among the sensing device category elements, this description refers to a "HumiditySensor". By looking at the unit field, one can also infer that this sensor provides data in percentages and hence it corresponds to "Percent" category element for the unit category.

6.2.5. Word Vector Training Dataset

This dataset is used to derive worst-case and best-case accuracy thresholds, that can be used to apply on the semantic matchmaking process. A worst-case and best-case assessment assists in defining limits that are relevant in particular when considering that the matchmaking process occurs on the Edge, eventually being applicable to far Edge devices (constrained devices).

As mentioned in section 4.4.2, the word vectors published by Google includes 3 million vectors with 300 dimensions each that takes around 6 GBs of space in a text file. Therefore a subset of the word vectors will be used. In order to analyse different word vector subsets, an experiment that shows the effect of subset size on the accuracy is carried out. Table 6.1 shows how different subset of word vectors

⁷https://git.fortiss.org/iiot/demonstrator2/-/tree/erkan/matching_improvement/td_to_ontology_matching/dataset/testing/ground_truth.txt

perform. The word vectors are ordered based on the frequency of the words in descending order. Therefore each subset with size x refers to the most popular x words. Subset size in MBs refers to how much space does a subset requires when stored in a text file. Accuracy is calculated using the best performing key and value thresholds, based on the results obtained after evaluating different key and value thresholds (see section 6.3. While the subset size is linearly correlated with the size of each file, the increase in accuracy gets lower as the subset size increases.

Table 6.1.: Size and accuracy comparison for word vector subsets of different sizes.

Subset size	Size in MBs	Accuracy
1,000	3.48	49.16
2,000	7.96	50.33
5,000	17.4	68.83
10,000	34.8	62.66
20,000	69.6	64.83
50,000	174	70.16
100,000	348	73
200,000	796	73.83
300,000	1194	74.66

For the accuracy comparison, the Word2Vec-based and k-Means-based semantic matchmaking algorithms are trained using both the best performing and the worst performing, smallest and the biggest set of vectors.

6.3. Results, Performance Comparison of Similarity Approaches

This section includes a performance comparison of the selected similarity approaches. Table 6.2 shows the settings for each of the similarity approach. The abbreviations will be used for the rest of the chapter instead of the full name for convenience.

6.3.1. Similarity Threshold Impact on W2VEC

There are two threshold values that need to be set manually before running the semantic matchmaking algorithm. As show in algorithm 2, these thresholds are the *key* threshold which refers to the value where a key in a TD is accepted as similar to an aggregation point, and the *value* threshold, where a value in a TD is accepted as

Table 6.2.: Settings for each tested algorithm.

Similarity approach	Dataset	Abbreviation
LEX-DB	WordNet[18] lexical db	LEX-DB
W2VEC	TRAINING	W2VEC-TD
W2VEC	300,000 vectors for most popular words published by Google	W2VEC-300k
W2VEC	1,000 vectors for most popular words published by Google	W2VEC-1k
K-MEANS	TRAINING	K-MEANS-TD
K-MEANS	300,000 vectors for most popular words published by Google	K-MEANS-300k
K-MEANS	1,000 vectors for most popular words published by Google	K-MEANS-1k

similar to child nodes of an aggregation point. In order to find which key and value thresholds provide better accuracy, different combinations of the two thresholds have been evaluated, for all of the proposed approaches. The sentence similarity approach performed worse than the other 2 approaches in terms of accuracy with using different key-value thresholds. The results for 2 best performing similarity approaches, word2vec word embeddings and k-Means, are shown in this and the following section.

Threshold Analysis for a 300,000 Vectors Dataset

The experiment has been carried out by considering the worst-case and best-case word vectors from table 6.1.

Table 6.3 provide the results when considering 300,000 vectors.

Both key and value similarity is ranging between 0.5 and 0.9. Accuracy is at the highest level when the key similarity threshold is 0.5, 0.7 and 0.8, and the value similarity threshold is 0.7. When looking at the rows where value similarity is 0.7, changing the key similarity impacts the accuracy in less than 1 percent. A

Table 6.3.: Impact, key and value threshold, accuracy of W2VEC-300k.

Key similarity threshold	Value similarity threshold	Accuracy
0.5	0.5	64.66
0.5	0.6	69.16
0.5	0.7	74.33
0.5	0.8	65.83
0.5	0.9	58.66
0.6	0.5	66
0.6	0.6	70.66
0.6	0.7	74.16
0.6	0.8	65.66
0.6	0.9	58.33
0.7	0.5	66.16
0.7	0.6	70.83
0.7	0.7	74.33
0.7	0.8	65.83
0.7	0.9	58.33
0.8	0.5	66.16
0.8	0.6	70.83
0.8	0.7	74.33
0.8	0.8	65.83
0.8	0.9	58.33
0.9	0.5	65.83
0.9	0.6	70.16
0.9	0.7	73.66
0.9	0.8	65.16
0.9	0.9	58.33

roughly similar situation holds for all of the value similarity thresholds. On the other hand, if the key similarity is kept fixed and the value similarity is changed, then the accuracy varies more than 10 percent. As an example, when the key similarity threshold is 0.5 and value similarity changes between 0.5 and 0.9 then the lowest accuracy value becomes 58.66 while the highest accuracy is 74.33 percent. This leads to the conclusion that value similarity threshold has more relevancy than the key threshold on accuracy. This is expected, given that the value threshold

6. Performance Evaluation

relates with the matching to a child node on an ontology, so provides a finer-grained matchmaking.

Since the accuracy is at the highest when the value similarity threshold is 0.7 and the key similarity has a lot less relevancy than the value similarity threshold, another experiment is carried out by ranging the value similarity threshold between 0.65 and 0.75 and keeping the key similarity threshold fixed at 0.5, 0.7 and 0.8. The purpose of this experiment is to see how much the accuracy changes when a more finer-grained increase on the value threshold is applied.

Results are presented in Table 6.4, which shows that accuracy is higher when the value threshold is set to 0.67, 0.68 and 0.69. Setting the key similarity threshold to 0.5, 0.7 or 0.8 produces the exact same results. Increasing the value similarity threshold above 0.69 reduces the accuracy of the algorithm.

Table 6.4.: Impact of key and value threshold finer.grain values on the accuracy of W2VEC-300k.

Key similarity threshold	Value similarity threshold	Accuracy
0.5, 0.7 or 0.8	0.65	74
	0.66	74.16
	0.67	74.66
	0.68	74.66
	0.69	74.66
	0.7	74.33
	0.71	74.33
	0.72	71.66
	0.73	71.66
	0.74	71
	0.75	71.5

Threshold Impact Analysis for a 1,000 Vectors Dataset

A larger vector dataset intuitively provides the best performance. However, it also brings the trade-off of having to handle large storage and large processing times. Therefore, a second experiment to calibrate the key and value thresholds is carried out, by considering a small dataset, using only the 1,000 vectors for most popular words from Google's dataset.

Table 6.5 provides the results achieved when the key and similarity thresholds varies between 0.5 and 0.9. The highest accuracy obtained from this experiment is

Table 6.5.: Impact, key and value thresholds, W2VEC-1k.

Key similarity threshold	Value similarity threshold	Accuracy
0.5	0.5	45.16
0.5	0.6	48.16
0.5	0.7	49.16
0.5	0.8	50
0.5	0.9	49.5
0.6	0.5	45.16
0.6	0.6	48.16
0.6	0.7	49.16
0.6	0.8	50
0.6	0.9	49.5
0.7	0.5	45.16
0.7	0.6	48.16
0.7	0.7	49.16
0.7	0.8	50
0.7	0.9	49.5
0.8	0.5	45.16
0.8	0.6	48.16
0.8	0.7	49.16
0.8	0.8	50
0.8	0.9	49.5
0.9	0.5	45.16
0.9	0.6	48.16
0.9	0.7	49.16
0.9	0.8	50
0.9	0.9	49.5

50. This experiment shows that changing the key similarity threshold doesn't impact the accuracy result when 1,000 vectors are used. Accuracy values change only when the value similarity change. We hypothesize that for sparse datasets, matchmaking to aggregation points may not be enough to perform well.

Next, another experiment is performed to see how finer-grained changes in value similarity threshold effects the accuracy. Results are shown in table 6.6. Since the key similarity threshold didn't have significant impact in the accuracy in the previous

Table 6.6.: Impact, key and value thresholds, finer-grained approach, W2VEC-1k.

Key similarity threshold	Value similarity threshold	Accuracy
0.8	0.75	49.33
	0.76	49.5
	0.77	49.16
	0.78	49.16
	0.79	50
	0.8	50
	0.81	50
	0.82	50
	0.83	50
	0.84	50
	0.85	49.5

experiment, a key threshold is trivially picked for this experiment. The results show that the highest accuracy is 50 when the value threshold is 0.79, 0.8, 0.81, 0.82, 0.83 or 0.84. Increasing the threshold further reduces the accuracy.

6.3.2. Similarity Threshold Impact on K-Means

Threshold Impact Analysis for the 300,000 Vectors Dataset

The same experiments are repeated for the K-Means approach when different key and value thresholds ranging between 0.5 and 0.9 are applied. Results are presented in Table 6.7. In this experiment, changing the key threshold doesn't impact the accuracy. The best performing value threshold is 0.7 with an accuracy of 58.

Table 6.7.: Impact, key and value thresholds, K-MEANS-300k accuracy.

Key similarity threshold	Value similarity threshold	Accuracy
0.5	0.5	45.83
0.5	0.6	47.83
0.5	0.7	58
0.5	0.8	54.33
0.5	0.9	55
0.6	0.5	45.83
0.6	0.6	47.83
0.6	0.7	58
0.6	0.8	54.33
0.6	0.9	55
0.7	0.5	45.83
0.7	0.6	47.83
0.7	0.7	58
0.7	0.8	54.33
0.7	0.9	55
0.8	0.5	45.83
0.8	0.6	47.83
0.8	0.7	58
0.8	0.8	54.33
0.8	0.9	55
0.9	0.5	45.83
0.9	0.6	47.83
0.9	0.7	58
0.9	0.8	54.33
0.9	0.9	55

6. Performance Evaluation

A second experiment is conducted and results are shown in Table 6.8. The value similarity ranges between 0.65 and 0.75 since the best performing value threshold in the previous experiment was 0.7. In this case, 0.69 value similarity threshold lead to the highest accuracy of 60.66.

Table 6.8.: Impact, key and value thresholds finer-grained values, K-MEANS-300k accuracy.

Key similarity threshold	Value similarity threshold	Accuracy
0.8	0.65	53.16
	0.66	56
	0.67	56.65
	0.68	60
	0.69	60.66
	0.7	58
	0.71	60.33
	0.72	59.83
	0.73	59.16
	0.74	59.16
	0.75	58.5

Threshold Impact Analysis for the 1,000 Vectors Dataset

We have run a similar experiment for 1,000 vectors using different key and value thresholds and results are shown in table 6.9. Again, changing the key similarity threshold for this experiment didn't impact the algorithm accuracy. The best performing value similarity threshold is 0.9 with the highest accuracy of 49.33 percent.

Table 6.9.: Impact, key and value thresholds, K-MEANS-1k accuracy.

Key similarity threshold	Value similarity threshold	Accuracy
0.5	0.5	33.5
0.5	0.6	37.16
0.5	0.7	39.83
0.5	0.8	41
0.5	0.9	49.33
0.6	0.5	33.5
0.6	0.6	37.16
0.6	0.7	39.83
0.6	0.8	41
0.6	0.9	49.33
0.7	0.5	33.5
0.7	0.6	37.16
0.7	0.7	39.83
0.7	0.8	41
0.7	0.9	49.33
0.8	0.5	33.5
0.8	0.6	37.16
0.8	0.7	39.83
0.8	0.8	41
0.8	0.9	49.33
0.9	0.5	33.5
0.9	0.6	37.16
0.9	0.7	39.83
0.9	0.8	41
0.9	0.9	49.33

Table 6.10 shows the effect of finer-grained value similarity threshold change on the accuracy. In this time, value threshold ranges between 0.85 and 0.95 since the

Table 6.10.: Impact, key and value thresholds, finer-grained values, K-MEANS-1k accuracy.

Key	Value	Accuracy
0.8	0.85	49.16
	0.86	49.33
	0.87	49.33
	0.88	49.33
	0.89	49.33
	0.9	49.33
	0.91	49.5
	0.92	49.5
	0.93	49.5
	0.94	49.5
	0.95	49.33

previous experiment showed 0.9 as the best performing threshold. The accuracy value is at the highest 49.6 level when the value threshold is 0.91, 0.92, 0.93 or 0.94.

6.3.3. Threshold Impact Analysis Summary

A summary for the threshold analysis is provided in Table 6.11. In regards to W2VEC-300K, the threshold analysis shows that the best performing key threshold values are 0.5, 0.7 and 0.8 for key and one of 0.67, 0.68 or 0.69 for value. When only 1,000 word vectors (W2VEC-1K) are used, the key similarity threshold does not impact significantly the accuracy and the best performing value thresholds are: 0.79, 0.8, 0.81, 0.82, 0.83 or 0.84. In regards to K-MEANS-300K, the key similarity threshold

Table 6.11.: Summary, best performing key and value thresholds.

Approach	Key	Value
W2VEC-300k	0.5, 0.7 and 0.8	0.67, 0.68 or 0.69
W2VEC-1k	No impact	0.79, 0.8, 0.81, 0.82, 0.83 or 0.84
K-MEANS-300K	No impact	0.69
K-MEANS-1K	No impact	0.91, 0.92, 0.93 or 0.94

changes did not impact significantly the algorithm accuracy, and the best performing

value threshold is 0.69. When 1,000 word vectors are considered (K-MEANS-1K) the key similarity threshold again didn't have a significant impact on the accuracy and the best performing value threshold was one of 0.91, 0.92, 0.93 or 0.94.

6.3.4. Algorithm Accuracy Analysis

A second set of experiments has been carried out to assess the accuracy of the proposed approaches, for the best and worst-case threshold scenarios summarised in Table 6.11.

The results are compared both in terms of the total matchmaking accuracy and also accuracy per category used in the FIESTA-IoT ontology. As previously stated in section 2.1, there are 4 categorizations published for quantity kind (QK), unit, sensing device (SD) and domain of interest ontology elements. As mentioned in section 6.2.4, the baseline testing dataset has a high sparsity when it comes to the domain of interest category. Hence it has been excluded in the accuracy analysis.

A match occurs when the selected semantic matchmaking algorithm matches a given TD to the same ontology element as the one that is in the baseline dataset. A mismatch occurs when the algorithms matches the TD to a different ontology element. Lastly, if the algorithm can't find a match for a given TD even though there is a match described in the baseline dataset, then we call it an undetected occurrence.

Tables 6.12, 6.13, 6.14, 6.15, 6.16, 6.17 and 6.18 provide results for the number of matched, mismatched and undetected ontology elements where the applied algorithm couldn't detect the match, accuracy values per category and also total values for all of proposed approaches.

Table 6.12.: Accuracy of LEX-DB.

	QK	Unit	SD	Total
Match	117	135	101	353
Mismatch	10	0	13	23
Undetected	73	65	86	224
Accuracy	58.5	67.5	50.5	58.83

Summary of Accuracy results

To better explain the results, Figure 6.1 presents the accuracy values achieved for all algorithms when performing a matchmaking of IoT TDs to the FIESTA-IoT category QK. Results shown that W2VEC achieves overall the best accuracy which significantly decreases (as expected) when sparser datasets are used. K-Means is the second best

6. Performance Evaluation

Table 6.13.: Accuracy of W2VEC.

	QK	Unit	SD	Total
Match	55	76	48	179
Mismatch	137	110	144	391
Undetected	8	14	8	30
Accuracy	27.5	38	24	29.83

Table 6.14.: Accuracy of W2VEC-300k.

	QK	Unit	SD	Total
Match	151	156	141	448
Mismatch	21	12	8	41
Undetected	28	32	51	111
Accuracy	75.5	78	70.5	74.66

Table 6.15.: Accuracy of W2VEC-1K.

	QK	Unit	SD	Total
Match	100	118	82	300
Mismatch	43	11	47	101
Undetected	57	71	71	199
Accuracy	50	59	41	50

Table 6.16.: Accuracy of K-MEANS-TD.

	QK	Unit	SD	Total
Match	25	24	30	79
Mismatch	175	176	170	521
Undetected	0	0	0	0
Accuracy	12.5	12	15	13.16

performing algorithm overall, while LEX-DB also shows good performance (however, LEX-DB has been trained with Word2NET). Figure 6.4 provides accuracy results for all algorithms, when TDs are matched to the FIESTA-IoT category Unit. In this case, W2VEC and K-MEANS achieve a similar performance, with one exception, when the smaller and sparser dataset is considered. Overall, both algorithms achieve a good accuracy percentage both for 300k vectors and for 1k vectors. LEX-DB is presented

6. Performance Evaluation

Table 6.17.: Accuracy of K-MEANS-300K.

	QK	Unit	SD	Total
Match	123	154	87	364
Mismatch	25	8	32	65
Undetected	52	38	81	171
Accuracy	62.5	77	43.5	60.66

Table 6.18.: Accuracy of K-MEANS-1K.

	QK	Unit	SD	Total
Match	92	115	90	297
Mismatch	33	13	13	59
Undetected	75	72	97	244
Accuracy	46	57.5	45	49.5

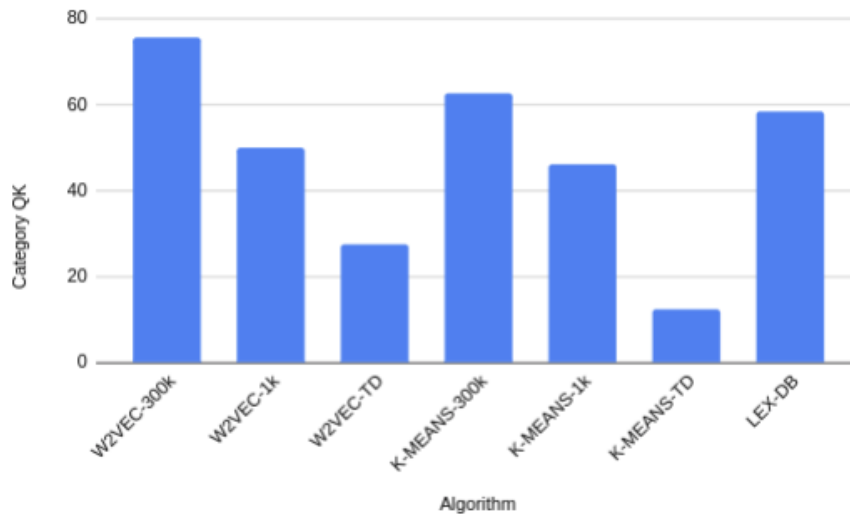


Figure 6.1.: Accuracy for category QK.

here, for reference. As explained, LEX-DB has been trained against WordNet and therefore, the accuracy should not change.

Figure 6.3 provides accuracy results for all algorithms, when TDs are matched to the FIESTA-IoT category SD. W2VEC is the algorithm that provides the best accuracy for this case. K-MEANS exhibits a different result pattern, where the accuracy is

6. Performance Evaluation

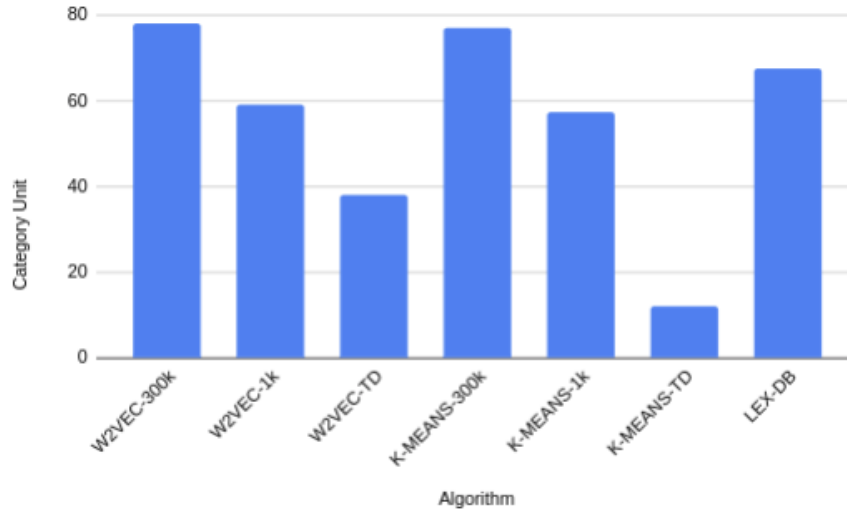


Figure 6.2.: Accuracy for category Unit.

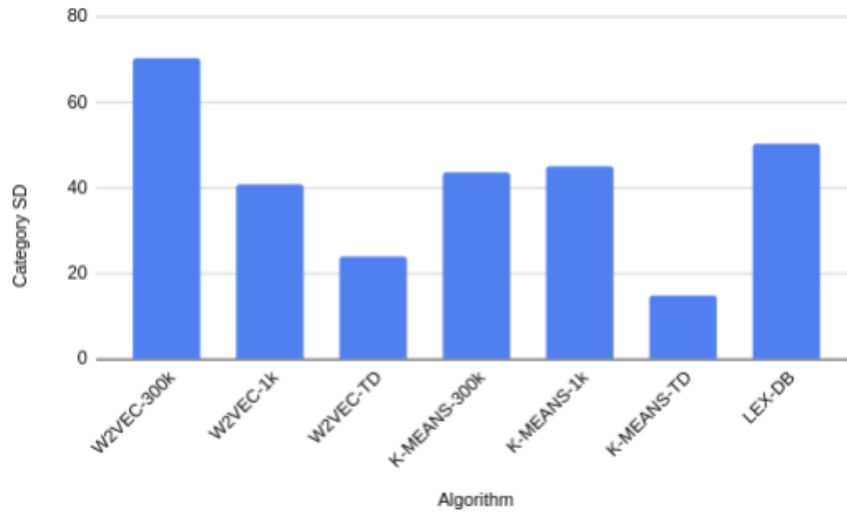


Figure 6.3.: Accuracy for category SD.

slightly higher for the sparser testing dataset (45 instead of 43). The LEX-DB value is now lower for the SD category, which implies that the type of wording may not adequately cover the SD category.

Figure 6.4 shows the total averaged accuracy per algorithm and case run. W2VEC

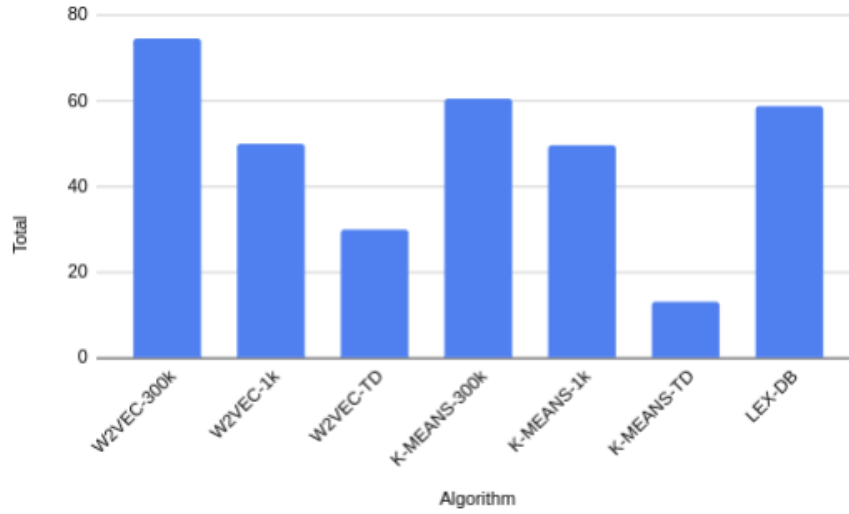


Figure 6.4.: Total accuracy.

has a particular good performance when 300k vectors are used, but it significantly reduces if 1k vectors are used. On the other hand, K-MEANS achieves less accuracy than W2VEC when 300k vectors are considered, but seems more stable when 1k vectors are applied.

Overall, W2VEC seems to provide better results when large, dense datasets are considered. For the 1K case, W2VEC accuracy lowers and is similar to the one of K-Means. However, for the majority of cases tested, W2VEC achieves the overall best performance.

LEX-DB accuracy, which has been trained against WordNet, can only be compared with the cases run for 300k vectors. In such case, LEX-DB achieves lower performance than either W2VEC or K-MEANS across all use-cases.

For the case of the smaller and sparser testing dataset (TD cases), both W2VEC and K-MEANS is very low.

Overall, the accuracy achieved for all algorithms and ran cases is between 60% and 80%. Ideally, such accuracy should reach a 90% level, in order for an adequate automated process to run better. We believe this can be improved by relying on multiple ontologies, e.g., a cross-domain approach.

6.4. Node Usage Analysis

A last batch of experiments has been run to assess node usage of each algorithm, having in mind future deployments in Edge devices. For far Edge devices, we have selected three different types of equipment representing different types of Edge devices, and Table 6.19 shows CPU, memory, disk and operating systems for each selected device:

- A Lenovo ThingPad T460p, standing for a regular end-user equipment device.
- A Raspberry Pi 4B, standing for an embedded Edge controller device.
- An Intel NUC 10, standing for an example of a IoT gateway device.

Table 6.19.: Hardware details and operating system of each testing device.

Device	CPU	Memory	Disk (Total / Free)	OS
Lenovo ThingPad T460p	Intel® Core™ i7-6700HQ CPU @ 2.60GHz × 8	Samsung M471A1K43-BB0-CPB 16 GiB	Samsung SSD 850 (465 GiB / 245 GiB)	Ubuntu 20.04.4 LTS
Raspberry Pi 4B	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz	4GB RAM	32 GB / 16.8 GB SD Card	Raspbian GNU/Linux 10 (buster)
Intel® NUC 10 Performance kit NUC10i7FNH	Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz	Kingston SODIMM DDR4 Synchronous 2667 MHz 16 GiB	Samsung SSD 970 EVO Plus (500 GBs / 404 GBs)	Ubuntu 20.04.3 LTS

The node usage analysis considers average time required to perform a match (Matching Duration in milliseconds, MD), and peak memory in MBytes (PM, MB) use for each selected algorithm. To obtain the running time, the semantic matchmaking algorithm is executed with a given similarity algorithm for 200 times and average

6. Performance Evaluation

time required to match a TD to an ontology element is calculated. Memory usage of the process that runs the matchmaking algorithm is also tracked during the whole execution and highest memory usage is observed. Table 6.20 presents results for TESTING1.

Table 6.20.: Node usage analysis results for TESTING1.

Device Type	Device	MD (ms)	PM (MB)	Algorithm
End-user device	Lenovo ThinkPad T460p	156189	258	LEX-DB
		774	3894	W2VEC-300k
		2025	3910	K-MEANS-300k
		179	249	W2VEC-1k
		609	249	K-MEANS-1k
Edge controller	Raspberry Pi 4B	> 10 minutes	142	LEX-DB
		2783	2401	W2VEC-300k
		3899	2402	K-MEANS-300k
		715	121	W2VEC-1k
		2531	122	K-MEANS-1k
IoT Gateway	Intel® NUC 10 Performance NUC10i7FNH	60979	256	LEX-DB
		585	3889	W2VEC-300k
		1440	3890	K-MEANS-300k
		145	240	W2VEC-1k
		556	240	K-MEANS-1k

In terms of MD, the best algorithm is W2VEC across all devices. LEX-DB is by far the algorithm performing worse across all devices. In regards to memory, the algorithm that shows better performance is LEX-DB overall, while W2VEC and K-MEANS show a similar performance. We have repeated the experiment for the C-TESTING set, to understand if a deeper cleaning process of the dataset may in the future significantly impact node usage results. Results are provided in Table 6.21. Results show that a thorough cleaning process can improve results in terms of MD, but we highlight that there is not a significant impact. For instance, W2VEC-300k has a performance improvement from 774ms to 708ms (0.08%) for the Lenovo equipment; a reduction from 2783 to 2184 ms (0.2%) for the Raspberry Pi 4B. The improvement range is similar when considering the 1k cases (0.1%). In terms of memory usage, there is also not a significant improvement.

6. Performance Evaluation

Adding a deeper cleaning process will also impact the overall memory and matching time. The improvements observed hint that such integration may not pay up in terms of node usage improvement.

Table 6.21.: Node usage analysis for C-TESTING.

Device Type	Device	MD (ms)	PM (MB)	Algorithm
End-user device	Lenovo ThinkPad T460p	148911	257	LEX-DB
		708	3892	W2VEC-300k
		1266	3908	K-MEANS-300k
		160	249	W2VEC-1k
		404	249	K-MEANS-1k
Edge controller	Raspberry Pi 4B	> 10 minutes	141	LEX-DB
		2184	2400	W2VEC-300k
		3671	2401	K-MEANS-300k
		626	121	W2VEC-1k
		1665	121	K-MEANS-1k
IoT Gateway	Intel® NUC 10 Performance NUC10i7FNH	56031	255	LEX-DB
		515	3888	W2VEC-300k
		910	3891	K-MEANS-300k
		124	240	W2VEC-1k
		376	240	K-MEANS-1k

7. Key Findings

The dissertation proposed to address 4 key research questions as detailed in section 1.2.

Concerning RQ1 *"Which functional blocks are required to support a semi-automated match-making process between IoT TDs and service descriptions?"*, we have proposed an architecture (rf. to Chapter 4) that comprises data pre-processing, interfacing to ontologies, semantic matchmaking and data aggregation.

These 4 functional blocks serve the semantic matchmaking purpose. Moreover, we have also analysed if a deeper cleaning process would assist the impact in terms of node usage metrics. The experiments carried out in Chapter 6 show that the cleaning does not seem to be a required artifact.

Concerning RQ2 *"How to define similarity thresholds that are adequate for the semantic matchmaking process?"* we have carried out an extensive evaluation to calibrate threshold values that can assist in a finer-grained matchmaking. The achieved results show that the value threshold is more relevant than the key threshold to assist a finer-grained semantic matchmaking. The value threshold assists in performing a match to the child nodes of an aggregation point of the ontology.

IN regards RQ3 *"Which approaches can be employed to support an ontology-based semantic matchmaking process, and can ML improve the semantic matchmaking?"*, we have implemented and evaluated three approaches (LEX-DB, W2VEC, K-MEANS) in terms of accuracy and node usage.

The results show that W2VEC (NLP with a neural network model) achieved the best performance for the majority of cases run, both in terms of accuracy and running time. The clustering approach based on K-MEANS exhibits good performance for the different sizes of training sets, but its performance is lower than the one achieved with W2VEC. LEX-DB (cosine similarity approach) has shown the worse performance.

A list of additional key findings obtained is given below:

- The 2 ML-based approaches W2VEC and K-MEANS can extract semantic meaning more accurately than LEX-DB as the latter does not include a learning process. Overall, W2VEC achieved best performance than K-MEANS.
- W2VEC and K-MEANS have a better performance in terms of node usage than LEX-DB.

7. Key Findings

- The word vectors published by Google represents generic meaning of words rather than IoT related meaning since it is trained on Google News dataset. This leads to inaccurate matchings or no matching at all. For example, the word "temperature" in the Google News dataset most likely refers to weather temperature rather than soil temperature. Therefore in a vector space it is placed nearer to weather and weather related words.
- The better an IoT device is described, the better the matching is. Better refers to how detailed a thing description is, e.g., does it include a measurement unit, where does the device located?
- Performance of the semantic matchmaking also depends on how comprehensive a given ontology is. For instance, if soil temperature is not listed among measurement aspects in an ontology data, then a service that requires soil temperature can never receive sensor observations.

8. Conclusions and Future Work

This dissertation is focused on the development of semi-automated matchmaking processes that can assist IoT interoperability in large-scale IoT networks. Initially, we made the following assumptions: i) each IoT Thing has a semantic description (TD); ii) every IoT service can be described semantically based on an ontology. Based on these assumptions, we proposed an architectural solution to match IoT TDs to service descriptions. The dissertation then assesses different semantic matchmaking approaches, derived from a selection based on an analysis of related work. Three approaches have been selected (LEX-DB, W2VEC, K-MEANS) and evaluated in terms of accuracy and node usage impact. During the dissertation, the proposed architectural solution has been implemented on a realistic testbed (fortiss IIoT Lab, demonstrator TSMatch) and open-source code is available.

Results achieved show that the best performing solution is W2VEC (NLP based on a neural network model), being LEX-DB the worse performing solution. The results achieved allowed also to detect gaps that can be addressed in future work. In order for further improvements on the proposed approach, relevant aspects for future work are:

- IoT TDs datasets need to be created and enriched. The created dataset is made available via git¹, and shall be enriched in future work, via the fortiss TSMatch development.
- W2VEC and K-MEANS should be further assessed with experiments that consider different percentages of training and testing sets.
- A research on how sensor data together with the metadata (thing descriptions) can help semantic matchmaking process should be conducted.
- Smarter ways for sensor data fusion for a given IoT service are important to be developed.

¹https://git.fortiss.org/iiot/demonstrator2/-/tree/erkan/matching_improvement/td_to_ontology_matching/dataset

A. Bibliography

- [1] R. Agarwal, D. G. Fernandez, T. Elsaleh, A. Gyrard, J. Lanza, L. Sanchez, N. Georgantas, and V. Issarny. "Unified IoT ontology to enable interoperability and federation of testbeds." In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. IEEE. 2016, pp. 70–75.
- [2] R. Agarwal, D. Gomez, T. Elsaleh, L. Sanchez, J. Lanza, and G. Amelie. "m3-lite Taxonomy." In: (July 2015). DOI: 10.5281/zenodo.1193303.
- [3] R. Agarwal, D. Gomez, T. Elsaleh, L. Sanchez, J. Lanza, and A. Gyrard. "FIESTA-IoT Ontology." In: (Apr. 2016). DOI: 10.5281/zenodo.1193299.
- [4] M. et al. *word2vec*. URL: <https://code.google.com/archive/p/word2vec/>. (accessed: 01.03.2022).
- [5] S. Bird, E. Klein, and E. Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.
- [6] N. Bnouhanna, E. Karabulut, R. C. Sofia, E. E. Seder, G. Scivoletto, and G. Insolvibile. "An Evaluation of a Semantic Thing To Service Matching Approach in Industrial IoT Environments." In: *inProc IEEE Percom IoT-Prod 2022 Workshop*. 2022.
- [7] G. Cassar, P. Barnaghi, W. Wang, and K. Moessner. "A hybrid semantic match-maker for IoT services." In: *2012 IEEE International Conference on Green Computing and Communications*. IEEE. 2012, pp. 210–216.
- [8] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. "Ontology matching: A machine learning approach." In: *Handbook on ontologies*. Springer, 2004, pp. 385–403.
- [9] M. Fazel-Zarandi and M. S. Fox. "Semantic matchmaking for job recruitment: an ontology-based hybrid approach." In: *Proceedings of the 8th International Semantic Web Conference*. Vol. 525. 01. 2009, p. 2009.
- [10] C. S. Guntupalli. *Open-source implementation of Sentence similarity based on Semantic nets and Corpus Statistics paper*. <https://github.com/chanddu/Sentence-similarity-based-on-Semantic-nets-and-Corpus-Statistics->. 2016.

- [11] M. Lagally and M. McCool. "IoT Interoperability with W3C Web of Things." In: *2022 IEEE 19th Annual Consumer Communications Networking Conference (CCNC)*. 2022, pp. 1–5. DOI: 10.1109/CCNC49033.2022.9700546.
- [12] Y. Li, D. McLean, Z. A. Bandar, J. D. O'shea, and K. Crockett. "Sentence similarity based on semantic nets and corpus statistics." In: *IEEE transactions on knowledge and data engineering* 18.8 (2006), pp. 1138–1150.
- [13] S. Lohmann, S. Negru, F. Haag, and T. Ertl. "Visualizing Ontologies with VOWL." In: *Semantic Web* 7.4 (2016), pp. 399–419. DOI: 10.3233/SW-150200.
- [14] C. Luo, X. He, J. Zhan, L. Wang, W. Gao, and J. Dai. "Comparison and benchmarking of ai models and frameworks on mobile devices." In: *arXiv preprint arXiv:2005.05085* (2020).
- [15] C. Malewski, A. Bröring, P. Maué, and K. Janowicz. "Semantic matchmaking & mediation for sensors on the sensor web." In: *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 7.3 (2013), pp. 929–934.
- [16] T. Mikolov, K. Chen, G. Corrado, and J. Dean. "Efficient estimation of word representations in vector space." In: *arXiv preprint arXiv:1301.3781* (2013).
- [17] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. "Distributed representations of words and phrases and their compositionality." In: *Advances in neural information processing systems* 26 (2013).
- [18] G. A. Miller. "WordNet: a lexical database for English." In: *Communications of the ACM* 38.11 (1995), pp. 39–41.
- [19] O. D. Model. *OneDM SDF Playground*. <https://github.com/one-data-model/playground/>.
- [20] Z. Peng, G. Xin, Y. Wei, W. Wang, B. Wang, and L. Wang. "Short Text Clustering Enhanced by Semantic Matching Model." In: *2019 2nd International Conference on Information Systems and Computer Aided Education (ICISCAE)*. IEEE. 2019, pp. 480–484.
- [21] R. Řehůřek and P. Sojka. "Software Framework for Topic Modelling with Large Corpora." English. In: *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50.
- [22] M. Ruta, F. Scioscia, G. Loseto, A. Pinto, and E. Di Sciascio. "Machine learning in the Internet of Things: A semantic-enhanced approach." In: *Semantic Web* 10.1 (2019), pp. 183–204.

- [23] I. B. G. Sarasvananda, R. Wardoyo, and A. K. Sari. "The K-Means Clustering Algorithm With Semantic Similarity To Estimate The Cost of Hospitalization." In: *IJCCS (Indonesian Journal of Computing and Cybernetics Systems)* 13.4 (2019), pp. 313–322.
- [24] A. Tzavaras and E. G. Petrakis. "Web of Things Functionality in IoT: A Service Oriented Perspective." In: *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*. IEEE. 2021, pp. 1–8.
- [25] W3C. *Web of Things (WoT) Testing*. <https://github.com/w3c/wot-testing/>.
- [26] R. Zhang and N. El-Gohary. "A machine-learning approach for semantic matching of building codes and building information models (BIMs) for supporting automated code checking." In: *International Congress and Exhibition "Sustainable Civil Infrastructures"*. Springer. 2019, pp. 64–73.
- [27] X. Zhang, Y. Wang, and W. Shi. "{pCAMP}: Performance Comparison of Machine Learning Packages on the Edges." In: *USENIX workshop on hot topics in edge computing (HotEdge 18)*. 2018.
- [28] H. Zhao. "Semantic matching across heterogeneous data sources." In: *Communications of the ACM* 50.1 (2007), pp. 45–50.

B. TD to Ontology Element Matching Algorithm

Algorithm 1 TD to ontology element matching algorithm

```
1: procedure td_to_ontology_matching(thing_description)
2:   short_td = "name", "description", "title" fields in thing_description
3:   sensor_description = thing_description["sensor"]
4:   delete "sensor" in thing_description
5:   delete "name", "description", "title" fields in thing_description
6:   ap = get_aggregation_points()
7:   matching_dict = new Dictionary()
8:   for ap_name in ap do
9:     matching = match(short_td, ap["taxonomy"], ap["name"])
10:    if matching == None then
11:      matching = match(sensor_description, ap["taxonomy"], ap["name"])
12:      if matching == None then
13:        matching = match(thing_description, ap["taxonomy"], ap["name"])
14:      end if
15:    end if
16:    matching_dict[ap_name] = matching
17:  end for
18:  return matching_dict
19: end procedure
```

Algorithm 2 Find a match for a given thing description in a given category

```
1: procedure match(thing_description, category, ap_name)
2:   highest_score = 0
3:   matched_key = None
4:   for key in thing_description do
5:     score = similarity(ap_name, key)
6:     if score > highest_score then
7:       highest_score = score
8:       matched_key = key
9:     end if
10:  end for
11:  if highest_score > KEY_SIMILARITY_THRESHOLD then
12:    highest_score = 0
13:    matched_value = None
14:    for node_name in category do
15:      score = similarity(node_name, thing_description[matched_key])
16:      if score > highest_score then
17:        highest_score = score
18:        matched_value = thing_description[matched_key]
19:      end if
20:    end for
21:    if highest_score > VALUE_SIMILARITY_THRESHOLD then
22:      return matched_value
23:    end if
24:  end if
25:  highest_score = 0
26:  matched_value = None
27:  for key in thing_description do
28:    for node_name in category do
29:      score = similarity(node_name, thing_description[key])
30:      if score > highest_score then
31:        highest_score = score
32:        matched_value = thing_description[key]
33:      end if
34:    end for
35:  end for
36:  if highest_score > VALUE_SIMILARITY_THRESHOLD then
37:    return matched_value
38:  end if
39: end procedure
```

C. Source Code Documentation

This section includes file structure for the 3 TSMATCH modules that are developed in the scope of this dissertations.

```
data_aggregation/                                // data aggregation module
  start_data_aggregation.sh                       // runs the module
  .env                                             // environment variables
  Dockerfile
  requirements.txt                               // required libraries
  src/
    main.py                                     // starting point
    repository/                                // db interactions
      OntologyRepository.py                   // ontology related db operations
      ServiceRequestRepository.py            // service requests related db
                                              operations
      SensorRepository.py                   // sensors related db operations
      BaseRepository.py                     // common db operations, e.g.
                                              connect, disconnect

    service/
      ServiceRequest.py                     // active service requests
      MQTTClient.py                         // mqtt client
    handler/
      ObservationEventHandler.py             // handle an incoming observation
                                              event
      ServiceRequestEventHandler.py          // handle an incoming service
                                              request event

    util/
      Neo4jUtil.py                          // db related helper functions

ontology_interface/                              // ontology interface module
  manage.py                                    // django starting point
  .env                                         // environment variables
  Dockerfile
  start_ontology_interface.sh                 // runs the module
  requirements.txt
  ontology/                                   // sample ontology files
    m3-lite.owl
    ontology.json
```

C. Source Code Documentation

```
ontology.txt
m3-lite.json
converter/                                // owl to json converter
    owl2vowl.jar
    test.json
app/                                      // django app
    tests.py
    models.py
    admin.py
    apps.py
    views.py                            // views for each url mappings
    urls.py                             // url mappings
    repository/                         // db interactions
        OntologyRepository.py          // ontology related db operations
    service/
        MQTTClient.py                 // mqtt client
        OntologyService.py            // includes ontology import
                                        // operations
    util/
        StringUtil.py                 // string related helper functions
web/
    asgi.py                             // initialize django asgi app
    settings.py                         // django settings
    urls.py                             // url mappings
    wsgi.py                             // initialize django wsgi app

td_to_ontology_matching/                // things description to ontology
                                        // matching module
.env                                    // environment variables
Dockerfile
requirements.txt                        // required libraries
start_data_enrichment.sh               // runs the module
src/
    main.py                            // module starting point
    evaluate.py                        // performance evaluation
    algorithm/
        Clustering.py                 // k-means clustering
        Word2Vec.py                  // word2vec
        SentenceSimilarity.py         // lexical-db based sentence
                                        // similarity
    repository/                        // db interactions
        OntologyRepository.py         // ontology related db operations
        ThingRepository.py            // things related db operations
        SensorRepository.py           // sensor related db operations
```

C. Source Code Documentation

BaseRepository.py	// common db operations
service/	
TDToOntologyMatching.py	// match thing descriptions to ontology elements
MQTTClient.py	// mqtt client
preprocessing/	
Word2vec.py	// word2vec related pre-processing operations
StringPreprocessing.py	// string pre-processing
util/	
StringUtil.py	// string helper functions
Neo4jUtil.py	// graph db helper functions
JSONUtil.py	// json helper functions
dataset/	
worst_final_clusters.json	// clusters using 1k word vectors
best_final_clusters.json	// clusters using 300k word vectors
testing/	// testing dataset
testing_cleaned/	// cleaned testing dataset
training/	// training dataset
word2vec/	// word vectors
google/	// word vectors by Google
training/	// trained word vectors using td training dataset