Reality Sensing, Mining and Augmentation
for Mobile CitizenGovernment Dialogue
FP7-288815

D1.2 - Mobile Sensor App with Mining Functionality

Dissemination level:	PU - Public
Contractual date of delivery:	Month 27, March 2014
Actual date of delivery:	Month 26, February 2014
Workpackage:	WP1 - Reality Sensing and Mining
Task:	T1.1, T1.2, T1.3
Type:	Prototype
Approval Status:	Draft
Version:	Beta 0.7
Number of pages:	38
Filename:	D1-2.tex

Abstract

HH: Add abstract

The information in this document reflects only the author's views and the European Community is not liable for any use that may be made of the information contained therein. The information in this document is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

This work was supported by the EU 7th Framework Programme under grant number IST-FP7-288815 in project Live+Gov (www.liveandgov.eu)

History

Version	Date	Reason	Revised by
0.1	2013-10-14	Outline	Heinrich Hartmann
0.2	2014-01-06	Added section on topic modeling	Christoph Kling
0.3	2014-01-07	Revised Outline	Heinrich Hartmann
0.4	2014-01-07	Alpha Version	Heinrich Hartmann
0.5	2014-01-21	Added section on Service Line Detection	Christoph Schaefer
0.6	2014-01-22	Added section on Distributed Geo Matching	Daniel Janke
0.7	2014-01-24	Added description of Sensor Collector	Heinrich Hartmann
0.8	2014-01-31	Added Traffic Jam Module Description	Laura Niittylä
0.9	2014-02-04	Added SVM Classifier Description	Elisavet Chatzilari

Author list

Organization	Name	Contact Information
UKob	Heinrich Hartmann	Phone: +49 261 287 2759 Fax: +49 261 287 100 2759 E-mail: hartmann@uni-koblenz.de
UKob	Christoph Schaefer	Phone: +49 261 287 2786 Fax: +49 261 287 100 2786 E-mail: chrisschaefer@uni-koblenz.de
UKob	Christoph Kling	Phone: +49 261 287 2702 Fax: +49 261 287 100 2702 E-mail: ckling@uni-koblenz.de
UKob	Daniel Janke	Phone: +49 261 287 2747 Fax: +49 261 287 100 2747 E-mail: danijank@uni-koblenz.de
MTS	Laura Niittylä	E-mail: Laura.Niittyla@mattersoft.fi
CERTH	Elisavet Chatzilari	E-mail: ehatzi@iti.gr

Executive Summary

HH: Add Summary

Abbreviations and Acronyms

API	Application Programming Interface
GPS	Global Positioning System
GSM	Global System for Mobile Communications
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
REST	Representational State Transfer
SVM	Support Vector Machine
URL	Uniform Resource Locator
UUID	Universal Unique Device Identifier
WP	Work Package
WIFI	Wireless Fidelity (IEEE 802.11), WLAN
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

Table of Contents

List of Figures

List of Tables

1 Introduction

HH: Write Introduction

1.1 Description of Deliverable D1.2

Sensor Data App with Mining Functionality: This deliverable will provide the extended version of the data capturing prototype for mobile devices, with implemented reality mining methods and optimized communication interfaces for result transmissions to the application server.

1.2 Task Descriptions

T1.1 **Sensor and user input data capturing**
This task will first define a taxonomy of classifications that are useful to determine for contextualization of citizens uploads including activity profiles (e.g. walking, cycling, riding train, riding bus,...) and coarse issue classifications (street, people, traffic,...). Research investigations will determine how these categories can be captured and classified economically using limited battery power (e.g. trading off between power draining sensors like GPS or audio by less expensive ones like accelerator monitoring) and limited bandwidth to the Live+Gov backbone server. The task also comprises the contextualized capturing of citizen text, deictic or voice input while accounting for limitations of mobile hardware and communication channels.

T1.2 **Smartphone based reality mining**
In this task we will develop memory and energy aware mining mechanisms for exploiting data collected in T1.1 for various kinds of initial processing on mobile devices - including classification, filtering, abstraction, and personalization. The key concern of our development will be on wide usability of reality mining methods and their flexible applicability for dynamic user contextualization in project scenarios.

T1.3 **Server based reality mining**
Based on limited analysis of data in T1.2 and based on citizens preferences and explicit consent, (possibly) abstracted sensor data will be uploaded to the Live+Gov backbone where further mining will occur. We will develop mining methods which are needed for the application, but cannot be run on individual smartphones (e.g. because input from multiple users is needed). In particular, this will comprise the detection of common patterns delivered from many citizens, but also outlier detection and treatment, which is required to discover spoofing or important emergencies. The methods resulting from this task will be particularly relevant for supporting augmented reality applications (WP3) and contextualization of data mining results in eGovernance scenarios in line with policy models (WP2).

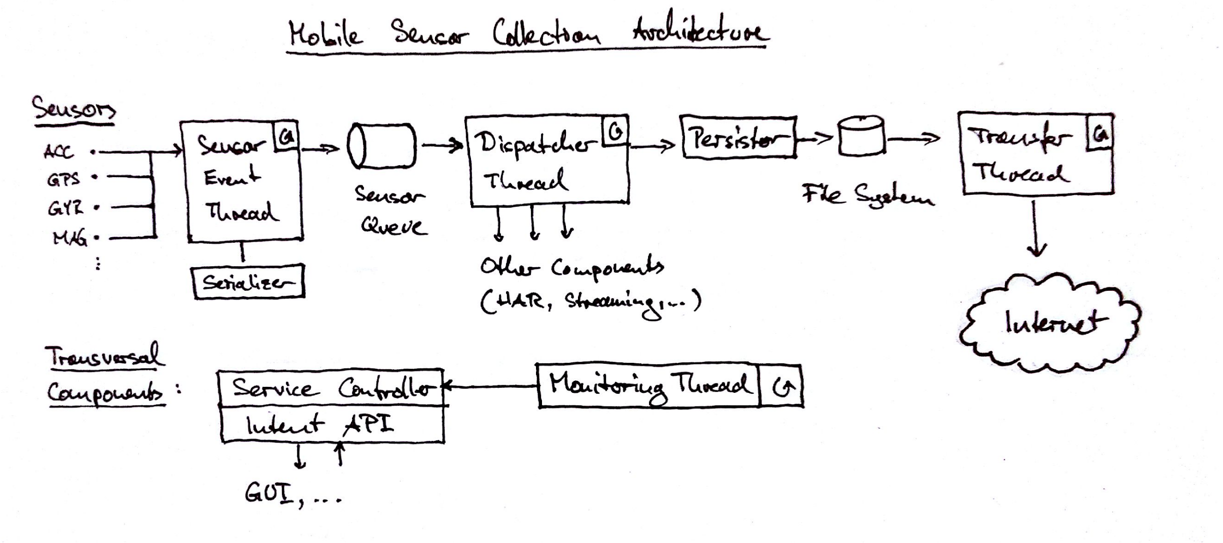


Figure 1: Sensor Collection Architecture

2 Sensor Collection Service

2.1 Mobile Sensor Collector

In this deliverable we present the a revised sensor collection component. It offers the following improvements over the last version presented in D1.1:

- High performance recordings on Low-End devices. During the field trial some low-end devices showed difficulties to record the sensor samples with sufficient frequency. We streamlined the architecture in order to reduced the overhead caused by the processing of the samples. In this process we removed dependency on external SDCF library, and reimplemented core parts of the component.
- Improved battery awareness. Power consumption in the sensor collection process is largely driven the GPS sensor. In May 2013 Google released a new location API, that takes advantage of multiple location provider and reduces the power consumption significantly. The revised component makes use of this new API, and offers a fallback in case the service is not supported by the device.
- Revised export format. We have improved the data exchange formats used for publishing samples to other components and storage on the central server. In section 2.1.3 we specify the ssf-Sensor Stream Format, which is inspired by the Common Log Format used by many webservers. It allows easy human inspection and processing by standard (UNIX) tools like “grep” and “sed” while being reasonably memory efficient.
- Streaming API. The streaming API is a new communication channel between the mobile sensor collector and the central server. It allows to transfer the incoming sensor data directly to the server using the ssf format.
- Integration into Live+Gov Service Center. The revised sensor collector and sensor storage service is compatible with the Live+Gov service center. This feature allows central user management as well as health monitoring and centralized log aggregation of the services.
- Seamless Extendability. The revised architecture can be easily extended by further stream

processing components. This is realized by a dispatcher thread that distributes sensor samples to registered components over a simple interface (again using ssf). Implemented extensions include the Human Activity Recognition component (c.f. section 3.1) and a GPS-sample publisher component that is used by the Service Line Detection service (cf. section 3.2).

The new architecture is sketched in Figure 1. It consists of the following components:

- **Sensor Event Thread**. The sensor event thread registers callback listeners for all configured sensors. When sensor events occur the received values are serialized in the ssf format (cf. 2.1.3) and pushed onto the SensorQueue.
- **SensorQueue**. The sensor queue is a synchronized queue object that stores sensor values as strings.
- **DispatcherThread**. The dispatcher thread reads sensor values from the sensor queue and passes them to several services that can be registered. Included services are the persistor service and the activity recognition component, the streaming service and GPS publication pipeline (for use in the Service Line Detection API).
- **Persistor**. The persistor service is executed by the DispatcherThread and appends the received sensor sample into a predefined file. Also a variant with zip-file compression is included.
- **TransferThread**. The transfer thread is an independent thread that transfers the persisted samples to the Live+Gov Servers.
- **Monitoring Thread**. The monitoring thread polls the different components and gathers run-time information like numbers of processed samples or transfer state.
- **Service Controller**. The service controller starts and stops all running threads, and takes care of proper configuration of all services. It listens to Intents specified in the API and changes the state of the service accordingly.

The following table summarizes all supported sensors:

Sensor	Description
GPS	Global position in latitude and longitude. If available, the GPS samples are gathered via Google's new Play Services location provider[1]. If the Play Services are not available, the service falls back to accessing the GPS samples directly.
Accelerometer	Measures the acceleration force in m/s^2 that is applied to a device on all three physical axes. Also the low-pass and high-pass filtered variants 'Linear Acceleration' and 'Gravity' offered by the Google API can be captured.
Rotation Vector	Measures the orientation of a device by providing the three elements of the device's rotation vector.
Gyroscope	Measures a device's rate of rotation in rad/s around each of the three physical axes.
Magnetic field	Measures the ambient geomagnetic field for all three physical axes in μT.
WLAN	A list of all available wireless local area networks in the transmission range.
Bluetooth	A list of all available bluetooth clients in the transmission range.
GSM	Cellular network operator and the radio cell the mobile phone is connected with.
Google Activity	Google recently released an Activity Recognition Library. The sensor collector is able to record activities as sensor values as well.

2.1.1 Intent API

The communication with other Live+Gov toolkit components is facilitated through the exchange intent messages[2]. The API was slightly improved since deliverable D1.1. New intents are marked with an asterisk (*).

The following API-calls are implemented.

- **Start/Stop Service.** Starts and stops the sensor collector service. Samples can only be recorded, or transmitted when the service is running.
- **Start/Stop Recording.** Starts and stops recording of sensor samples.
- **Send Annotation.** This intent allows users to annotate their recording by a string value that will be recorded by a "tag-sensor".
- **Enable/Disable sample storage.** Samples are stored in a local database, of variable size. If Disabled samples can still be broadcasted to the system.
- **Enable/Disable continues sample transfer.** Controls the sample transfer state machine. When enabled and a network connection is available sensor samples are transfered in batches to the server backend.
- **Transfer samples.** Triggers the sample transfer.

[2] http://developer.android.com/guide/components/intents-filters.html

- **Enable/Disable sample broadcast.** When enabled all recorded sensor samples are broadcasted in the form of intents into the system. This allows other components, e.g. feature extraction and context mining, to make use of the recorded sensor data.
- **Request Status Report.** When this intent is received a status report is broadcasted to the system.
- **Tag sample.** Annotations by users are realized as “Tag sample”. The content of this intents is included as “tag”-sensor in the recording.

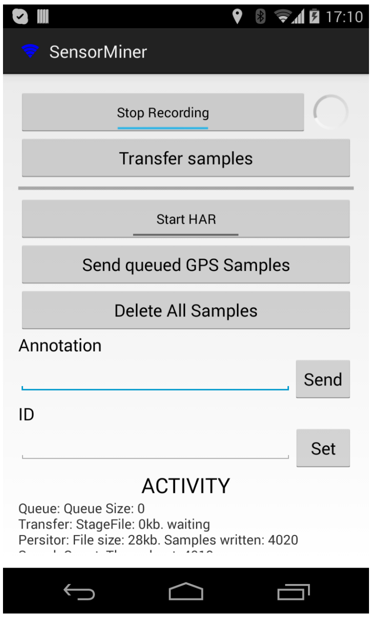
When a control intent is received by the component, the appropriate action is taken. After execution has finished a status report intent is broadcasted to the system which provides information about whether the requested method call was successful (cf. Table 1). Further technical description of the component and the API are included in D4.1.

Key	Type	Description
running	boolean	True if service is running.
recording	boolean	True if sensor samples are recorded.
storage	boolean	True if samples are stored in the database.
transfer	boolean	True if continues transfer of samples is enabled.
broadcast	boolean	True if samples are broadcasted.

Table 1: Structure of status intents

The sensor sample broadcast intent contains a unique device identifier, a timestamp and serialized sample data in XML format. Figure **??** shows two examples for GPS and accelerometer samples.

2.1.2 Testing GUI



```
Queue: Queue Size: 0
Transfer: StageFile: 0kb. waiting
Persitor: File size: 28kb. Samples written: 4020
```

Figure 2: Sensor Collection Architecture

2.1.3 Sensor Stream Format – "SSF"

This document describes the sensor stream format (*ssf*) which is used for transfering sensor values from a mobile device (or another sensor source) to the Live+Gov data-base. The file format is inspired by the Common Log Format used by many webservers. We view incoming sensor values as events and record them as a simple stream of CSV-rows. Each sensor has an individual prefix but writes into the same file.

The key advantages of this format as opposed to the previously used XML format are:

- Human readability. The format is easy to understand by humans.
- Statelessness. Every line can be interpreted without the context of the file. Therefore ssf-files can be arbitrarily sliced, concatenated and filtered using UNIX tools like *grep, cat, sed, awk*.
- Streaming. Lines can be individually transferred over tcp sockets or messaging systems. Allowing immediate inspection of the recorded samples on a remote system.

Format Description

The rows of the ssf-format have the following structure:

```
SENSOR_PREFIX,TIME_STAMP, USER_ID, SENSOR_VALUES
```

The authorative source for the field descriptions is the source code.

- `SENSOR PREFIX`. Identifies the sensor producing the sample. The following sensor prefixes are supported:
  `GPS` (GPS sensor), `ACC` (Accelerometer), `LAC` (Linear Acceleration), `GRA` (Gravity), `GYR` (Gyroscope), `MAG` (Magnetometer), `WIFI` (Wifi networks), `BLT` (Bluetooth), `GSM` (GSM cells), `ACT` (Google Play Services Activity), `ERR` (Error Value), `TAG` (Annotations entered by the user).
- `TIME STAMP` UNIX timestamp in milliseconds e.g. `1377609577214`
- `USER ID` ID that identifies records from the same user. The default value is the device-id that is provided by Android.
- `SENSOR VALUES` Sensor values in individual formats. In order to adhere to the CSV standard no `,`-symbol and new-lines may be used in this field.
  - `ACC/GYR/MAG/LAC/GRA` X,Y,Z-values separated by space characters.
  - `GPS` lat,lon,alt-values separated by space characters
  - `ACT` Activity Name, Confidence separated by space characters.
  - `WIFI` List of access-points, where each visible accesspoint is separated by a `;` and written as

    Escaped SSID String/Escaped BSSID String/Frequency in MHz/RSSI in dBm
  - `BLT` List of devices where each visible device is separated by a `;` and written as Escaped Address/Device Major Class/Device Class/Bond State/Optional Escaped Name/Optional RSSI

- `GSM` The state of the device written as Service State/Roaming State/Manual Carrier Selection State/Escaped Carrier Name/Escaped Signal Strength

  followed by `:`, followed by a possibly empty list of cells, where each cell is separated by a `;` and written as Escaped Cell Identity/Cell Type/RSSI in dBm

Example rows may look as follows:

```
ACC,1377605748123,5,0.9813749 0.0021324 0.0142523
GPS,1377605748156,5,50.32124 25.2453 136.5335
WIFI,1341244415,wifiUser,"WiFi AP"/"00:12:42"/2412/-45; "Another WiFi AP"/"33:13
TAG,1378114981049,anotherUser,"test tag"
BLT,1385988380374,bluetoothUser,"C8:F7:33:B7:B5:B4"/computer/computer laptop/bon
```

2.1.4 Streaming Service

The revised version of the sensor collector includes a streaming service, that transfers recorded samples directly to a remote machine.

The streaming service uses the ZeroMQ networking library[3] to transfer single ssf lines to the Live+Gov machine running a streaming server. The streaming server appends the received samples to a file (zmq_stream.ssf).

This simple service allows direct monitoring of the recorded samples, via simple UNIX command line tools, e.g.

```
tail -f zmq_stream.ssf
```

prints the recoded samples to the console.

```
tail -f zmq_stream.ssf | grep ^GPS
```

filters out gps samples. And finally

```
tail -f zmq_stream.ssh | cut --delimiter=',' --field=1 | logtop
```

shows the frequencies of the individual incoming sensor values.

2.2 Sensor Storage Service Architecture

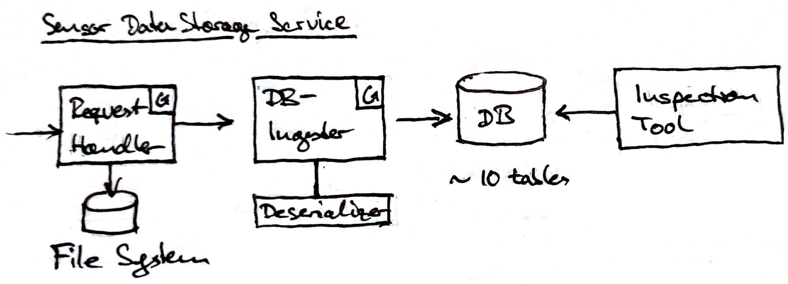


Figure 3: Sensor Collection Architecture

[3] http://www.zeromq.org

2.2.1 Database Scheme

The uploaded samples are stored in a PostgreSQL[4] database. We have switched from MySQL to this technology, because of the advanced query fuctionality that is offered by the PostGIS[5] plugin.

The database has the following schema:

```
sensor_gps (trip_id INT, ts BIGINT, lonlat GEOGRAPHY(Point));
sensor_accelerometer (trip_id INT, ts BIGINT, x FLOAT, y FLOAT, z FLOAT);
sensor_linear_acceleration (trip_id INT, ts BIGINT, x FLOAT, y FLOAT, z FLOAT);
sensor_gravity (trip_id INT, ts BIGINT, x FLOAT, y FLOAT, z FLOAT);
sensor_tags (trip_id INT, ts BIGINT, tag TEXT);
sensor_google_activity (trip_id INT, ts BIGINT, activity TEXT);
har_annotation (trip_id INT, ts BIGINT, tag TEXT);
```

2.2.2 Inspection Front End

2.2.3 Integration of Logging and Heartbeats

2.2.4 Cross-Platform Strategy

Explain problems with Titanium framework.

Android component runs on Blackberry.

Implement HAR as SAAS. Write client app for iOS.

[4] http://www.postgresql.org/

[5] http://postgis.net/

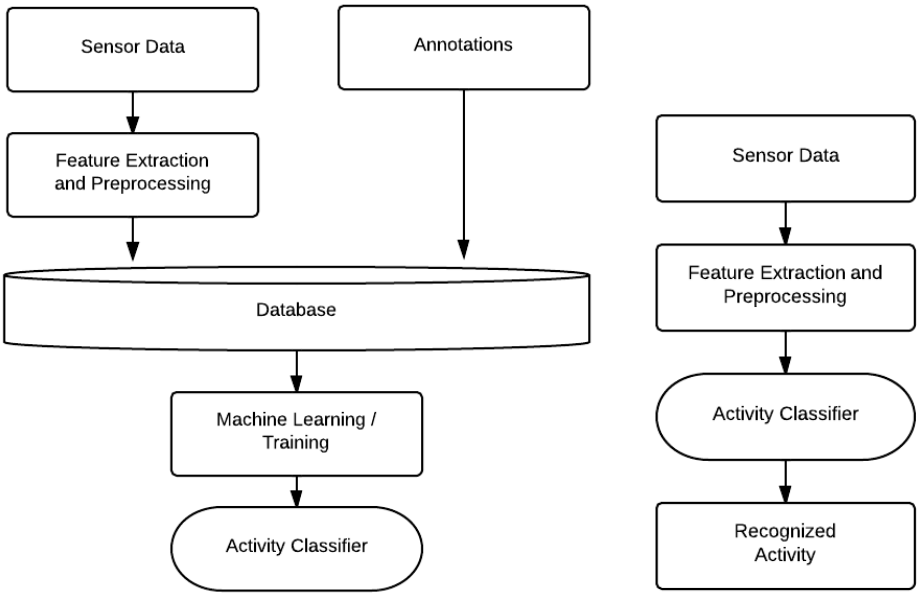
3 Reality Mining Methods

3.1 Human Activity Recognition

In this section we describe the implementation of the human acitivity recognition ('HAR') component on the mobile device. The algorithmic foundation of this data mining task have been described in detail in deliverable D2.2 in section 3.2 “Mobile Sensing and activity recognition”. We include a brief summary here (3.1.1) for the sake of completeness. Subsection 3.1.2 contains the descriptions of the implementation. In subsection 3.1.5 we evaluate our method to the on two data sets and compare it to the state of the art approaches.

3.1.1 HAR Method Summary

The process of activity recognition uses a pipeline of signal processing and machine learning techniques. It consists of two phases: the “training phase” and the “integration phase”.


Sensor Data
Annotations
Feature Extraction and Preprocessing
Database
Machine Learning / Training
Activity Classifier
Sensor Data
Feature Extraction and Preprocessing
Activity Classifier
Recognized Activity

(a) HAR Training Phase (b) HAR Integration Phase

Figure 4: Overview Human Activity Recognition

In the training phase (cf. Figure 4(a)) a group of volunteers is asked to perform the targeted activities for a certain amount of time, while recording sensor samples with the device in their pocket. The gained training data stored in a database and used to train a classifier of the activities.

In the integration phase (cf. 4(b)), the trained classifier is embedded into the mobile device.

Both phases rely on the preprocessing steps of ”windowing” and “feature generation”. The stream of incoming sensor data is divided into time windows of fixed size (typically 1-10 sec.) and for each window a set of features is computed. This features are filtering out relevant information from the raw signal. Typical features include mean values and standard deviations as well as frequency domain features like Fourier modes.

Deliverable D2.2 contains detailed lists of all sensors and features and data mining methods that are used in the literature as well as discussions of quality of the recognition results.

3.1.2 Component Description

Architecture Description.

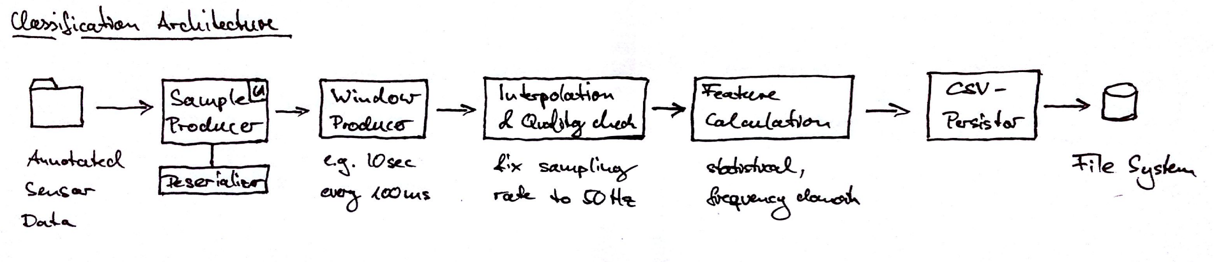


Figure 5: Classification Architecture

- SAMPLE PRODUCER The samples chosen to be used to train the classifier are present in a CSV file. Because of reusability the file has to be imported line by line so it is possible to reuse the whole streaming pipeline, which is already in place for the mobile device.
- WINDOW PRODUCER Since the classification of the current activity uses a timeframe, the window producer groups them together. It allows different window sizes and overlaps.
- INTERPOLATION & QUALITY CHECK To remove frequency disparity and ignore timeframes below a set sample frequency a quality check is in place. After the check, all windows get standardized to a constant frequency using interpolation between the data points.
- FEATURE CALCULATION The classification runs on different features extracted from the windows (cf. 3.1.3).
- CSV-PERSISTOR For further processing of the calculated data it is required to be store. This is done using a simple CSV data format, to ensure cross program importability.
- SENSOR EVENT THREAD The sensor collector is connected to nearly all sensors available on the device. The samples get pushed into a queue to keep the calculation time needed for each sensor event at a minimum.
- DISPATCHER THREAD This thread distributes the sensor samples to each connected component like the persistor or the human activity recognition.
- HAR PIPELINE The pipeline reuses most parts described in 5. The difference here is the fully trained classifier, which tries to identify the current activity.
- INTENT BROADCASTER After the classifier recognised an activity, it gets broadcasted using an intent.

3.1.3 Features

- Mean of each individual axis

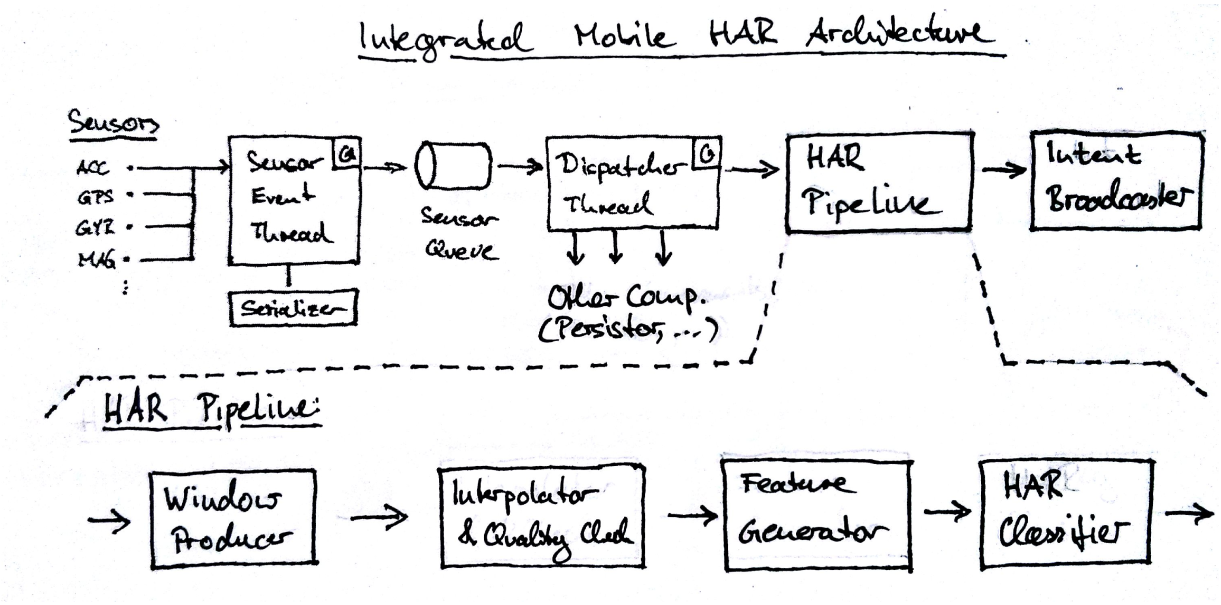


Figure 6: Integrated HAR Architecture

- Variance of each individual axis
- Mean of standard deviation
- Variance of standard deviation
- Bin distribution of standard deviation
- Bin distribution of frequency domain
- Device tilting angle
- Energy
- Kurtosis

3.1.4 Classifier Training

The first step is separating the data set into a train and test set. Here we used data from different users for each activity. After both data sets got processed by the classification architecture (cf. Figure 5) the training and test set get imported into *Weka*[6]. Weka comes with build in machine learning algorithms to choose from. Here the J48 implementation of the C4.5 algorithm is used to create a decision tree out of the provided training set. The test set is used to validate the created tree and calculate the accuracy and confusion matrix. Since Weka allows to export the created tree as Java class, it is possible to integrate the output directly into the HAR Pipeline.

SVM based classifier

A popular algorithm to train a binary classification model for mapping features to activities is the Support Vector Machines (SVMs). SVMs are known for their ability in smoothly generalizing

[6] http://www.cs.waikato.ac.nz/ml/weka/

and coping efficiently with high-dimensionality pattern recognition problems. They define a hypothesis space that includes all the possible linear separations of the data (Fig. 7) and they choose the one that maximizes the margin between the two classes (Fig. 8).

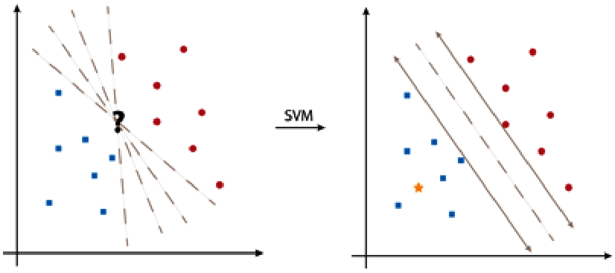


Figure 7: Left: Hypothesis space including all linear separations. Right: The selected hypothesis maximizes the margin.

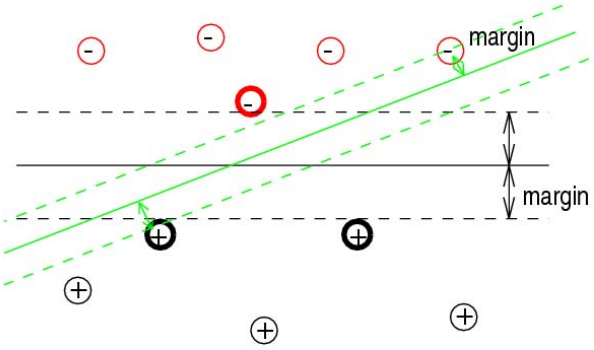


Figure 8: Maximization of the margin

The hyperplane that optimally separates the positive and the negative class (i.e. maximizes the margin) can be described by $\mathbf{w} \cdot \mathbf{x} + \mathbf{b} = \mathbf{0}$, where $\mathbf{w}$ is normal to the hyperplane and $\frac{b}{\|\mathbf{w}\|}$ is the perpendicular distance from the hyperplane to the origin (Fig. 9). The optimal hyperplane can be obtained by solving the following Quadratic Programming optimization problem:

$$\min \frac{1}{2} \parallel \mathbf{w} \parallel \quad \text{s.t.} \quad y_i(\mathbf{w} \cdot x_i + b) - 1 \geq 0 \quad \forall i \tag{3.1}$$

In order to relax the constraints of Eq. 3.1 and allow for some misclassified points, a slack variable $\xi_i, i = 1, \ldots, L$ is introduced which transforms Eq. 3.1 into:

$$\min \frac{1}{2} \parallel \mathbf{w} \parallel + C \sum_{i=1}^{L} \xi_i \quad \text{s.t.} \quad y_i(\mathbf{w} \cdot x_i + b) - 1 + \xi_i \geq 0 \quad \forall i \tag{3.2}$$

where the parameter C controls the trade-off between the slack variable penalty and the size of the margin. In the testing phase, in order to classify an unseen example x_t, its distance to the hyperplane is calculated using the formula $\mathbf{w} \cdot x_t + b$. This distance indicates the classifier's confidence that the unseen example x_t belongs to the examined class.

The previous consider linear separation of the data. However, this is rarely the case for most real world scenarios and for this reason the kernel trick has been introduced. Applying the kernel

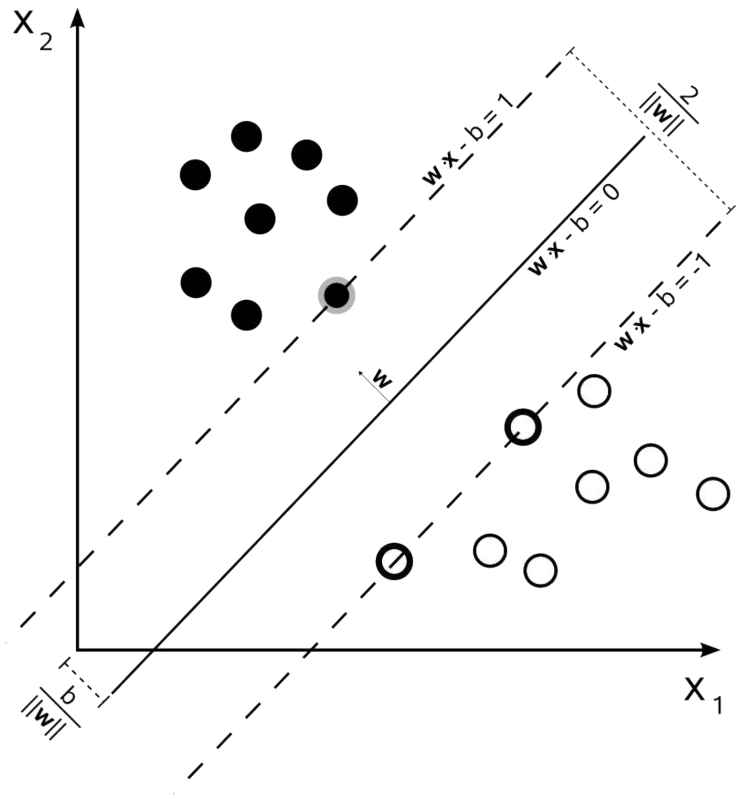


Figure 9: The resulting hyperplane after training an SVM.

trick to the cases where the classes are not linearly separable in the input feature space, we manage to map the features to a higher dimension (Fig. 10) where they can be linearly separated. For example, in the case of an RBF kernel the otherwise linear hyperplane is transformed to a hypersphere (Fig. 11). For any kernel $K(x,y)$, the classification model can be represented by a vector $\mathbf{w}$ (i.e. the model parameters), a bias scalar b and the support vectors $\mathbf{SV}_j, j = 1, \ldots, N_{SV}$. For an unseen example x_t, a confidence score is extracted by computing its distance to the hyperplane of the model:

$$confidence = \mathbf{w} * \sum_{j=1}^{N_{SV}} K(\mathbf{SV}_j, \mathbf{x}_t) + b \tag{3.3}$$

In our case, an SVM model is trained on the previously extracted features in order to learn the properties that define the examined activity. The models are trained using the one versus all (OVA) technique, i.e. all positive examples of the specific activity versus all negative examples (i.e. the examples of all other activities). The distance of a vector from the hyperplane indicates our confidence that during the analysed window, the user performs the examined activity. High positive values of this score increase our confidence that this window belongs to the positive class while high negative values provide strong confidence that the performed activity is not the examined one.

3.1.5 Evaluation

We have evaluated our classifier on two different datasets.

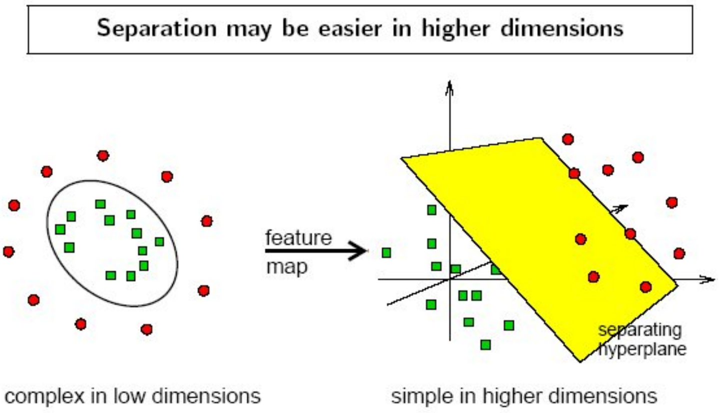

Separation may be easier in higher dimensions
feature
map
separating
hyperplane
complex in low dimensions
simple in higher dimensions

Figure 10: The data are not separable in the input space by a linear hyperplane. Using the kernel function, the data are mapped into a higher dimensional space where they can be linearly separated.

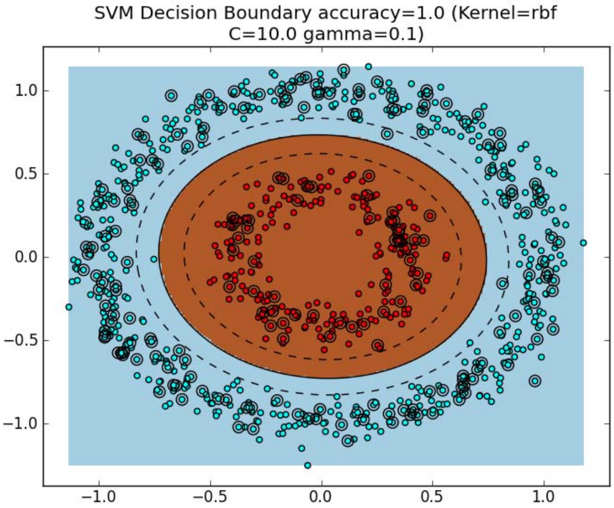


Figure 11: Non-linear separation of the data using the RBF kernel

The first dataset was gathered on the University Campus in Koblenz in December 2013. It contains a total of around $900K$ samples collected by 10 volunteers. The volunteers were instructed to perform the activities ”walking”, ”running”, ”stairs” and ”cycling” on predefined routes on the university campus (cf. Figure 13). The total time effort per volunteer was about 20-25minutes and a financial reward was offered as an incentive. After the recording the samples have been inspected using our inspection tool and the beginning and ending of the activities were manually stripped in order to avoid noise from holding the device in the hand.

The other dataset was obtained from the *UCI Machine Learning Repository*[7] and was gathered by Davide Anguita, et. al. [2] in 2012. It contains around $700K$ collected by 30 volunteers.

The number of samples per activity of both datasets are summarized in Figure 12. Both datasets contain only accelerometer samples, and have been preprocessed that have been sampled at a fixed rate of $50Hz$.

[7] http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones

Activity	UCI Dataset	UKOB Dataset
sitting	113.728	80.951
standing	121.984	320.737
walking	110.208	292.024
running	0	31.916
cycling	0	436.106
stairs	188.800	30.086
lying	124.416	0
Totals	659.136	903.156

Figure 12: Number of accelerometer samples by activity and dataset.

3.2 Service Line Detection

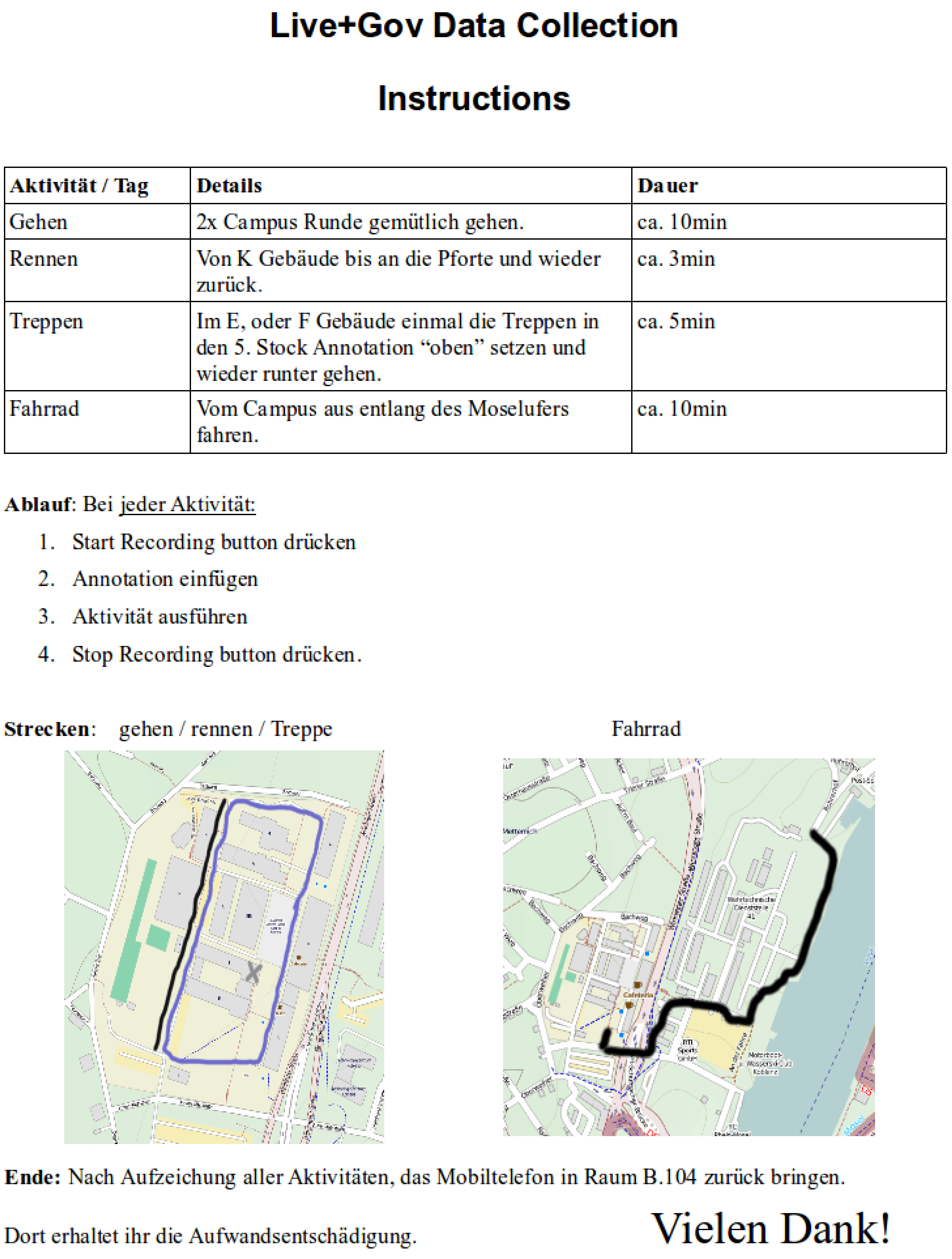
One part of the user contextualization is the service line detection. The aim of this component is to recognize if a citizen uses public transport within the HSL area in Helsinki. If such a usage is detected, the right service line id with its direction and the further itinerary will be determined. Based on this information it is possible to solve higher level tasks like traffic jam detection, connecting train recommendation, or network utilization analysis.

The service line detection is implemented as a server side mining component which provides a REST API for answering user queries in real time. For long term evaluations all API calls are recorded and send to the L+G Service Center on a daily bases. Due to the seamless integration into the L+G ecosystem, where each user gets an universal unique id, all received tracks are personalized and can be combined with additional information coming from other components. In a later analysis, the service line detection results can be augmented with the low level human activity recognition data of the same user to gain a better insight into the users behavior.

Assigning a trajectory to a specific service line requires the access to suitable background knowledge. In the Helsinki case it is two fold: First of all, the system is fed with all HSL public transportation schedules and associated geographic information of stops and travel paths. By means of this dataset the theoretical position of every vehicle at any point in time can be calculated. A drawback of only including static time tables is that actual schedule deviations stay disregarded. This gap is closed by the HSL Live API, which is the second data source the service line detection relies on. With this API, the actual positions of some vehicles can be determined. Especially trams are tracked in realtime and can be accessed via this interface. Unfortunately, the vast majority of vehicles, such as buses, have not yet been integrated. Lacking a complete realtime coverage is why still some uncertainty remains in the overall system. Therefore, the main challenge is to combine both kinds of data to archive the best possible performance.

Just as the background data, the classification algorithm is also two-stage. Given an user trajectory containing a small set of GPS coordinates with timestamps (Figure 14(a)), in a first step the algorithm checks if the latest given timestamp is near the actual time. If so, the HSL Live API is called in order to find all vehicles which are currently around the user (Figure 14(b)). Service lines found below a certain distance to the user are scored with a high probability to be the true line belonging to the given track.

In a second step, each coordinate of the given track is compared to the static route data (Figure 14(c)) obtained from the time tables and paths. It should be noted that the public transportation schedules only contain exact arrival times for each stop, but don't provide any



Live+Gov Data Collection

Instructions

Aktivität / Tag	Details	Dauer
Gehen	2x Campus Runde gemütlich gehen.	ca. 10min
Rennen	Von K Gebäude bis an die Pforte und wieder zurück.	ca. 3min
Treppen	Im E, oder F Gebäude einmal die Treppen in den 5. Stock Annotation “oben” setzen und wieder runter gehen.	ca. 5min
Fahrrad	Vom Campus aus entlang des Moselufers fahren.	ca. 10min

Ablauf: Bei jeder Aktivität:

1. Start Recording button drücken
2. Annotation einfügen
3. Aktivität ausführen
4. Stop Recording button drücken.

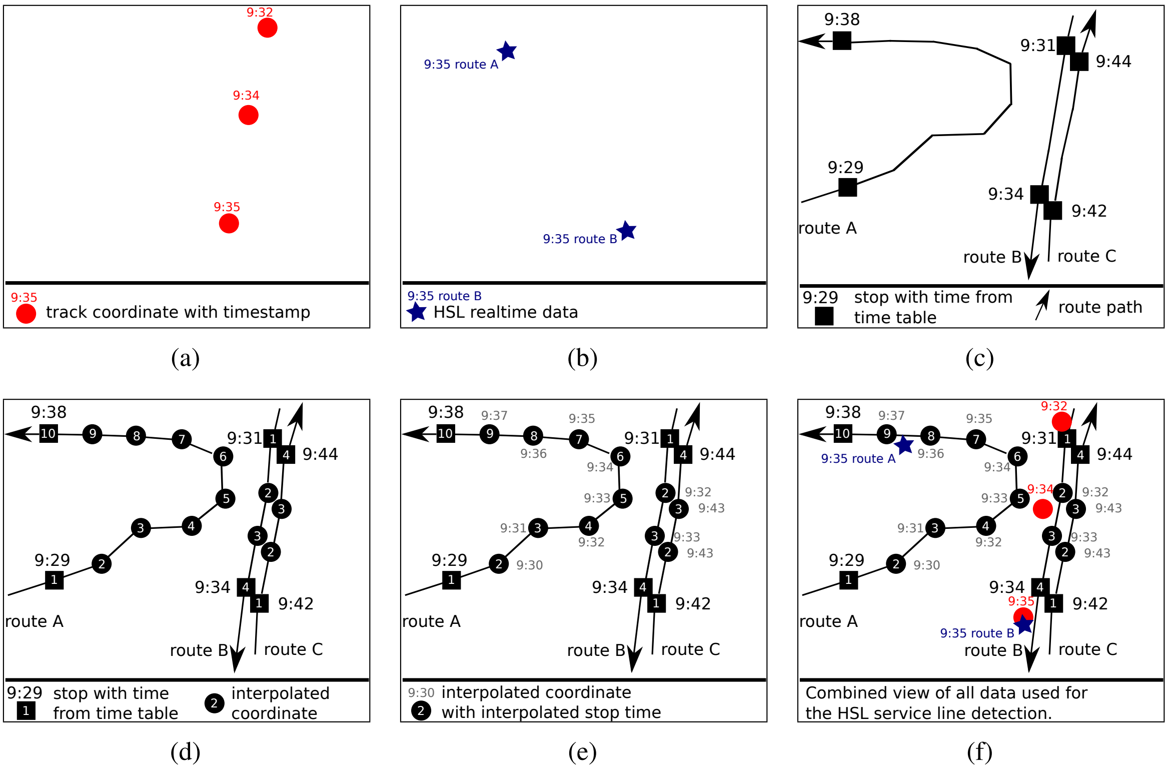
Strecken: gehen / rennen / Treppe Fahrrad

Ende: Nach Aufzeichung aller Aktivitäten, das Mobiltelefon in Raum B.104 zurück bringen.

Dort erhaltet ihr die Aufwandsentschädigung.

Vielen Dank!

Figure 13: Instructions for data collection in German language


9:32
9:34
9:35
9:35 track coordinate with timestamp
9:35 route A
9:35 route B
9:35 route B HSL realtime data
9:38
9:31
9:44
9:29
9:34
9:42
route A
route B
route C
9:29 stop with time from time table
route path
9:29 stop with time from time table
interpolated coordinate
9:37
9:35
9:36
9:33
9:32
9:43
9:30
9:30 interpolated coordinate with interpolated stop time
Combined view of all data used for the HSL service line detection.

(a) (b) (c)

(d) (e) (f)

Figure 14: The various input data for the HSL service line detection.

time data for the pathes between them. Since this data is too sparse to perform reliable service line detection on it, all paths are rasterized evenly (Figure 14(d)) and the corresponding time for each intermediate part is interpolated (Figure 14(e)). This ensures that no route section falls through the cracks when executing a simple distance query in time and space. Figure 14(f) shows the combined view of all data used for the HSL service line detection.

The result of the algorithm is the one service line which has the highest probability due to the Live API and/or the smallest distance to the given set of coordinates. During a testing phase, the optimal values of all involved parameters like distance thresholds, time and space raster sizes, and the individual scoring weights have to be learned. As one outcome of the Helsinki field trial, we expect to find a good heuristic how to setup the various ingredients of the algorithm.

3.2.1 Implementation details

As a starting point, HSL provided us with all static service line data like arrival times, stop positions, route meta data, and path files in the well known General Transit Feed Specification (GTFS)[8] format. The original data set had a size of 400 MB and was imported into a PostgreSQL database. All paths and stop positions were stored as native geospatial values using PostGIS, a spatial database extender for PostgreSQL. After the import, we found 587 distinct routes, 7534 stops, and a total number of 206,153 trips in the database. In general, a service line like the bus 934 from Myyrmäki to Luhtaanmäki, has two directions, travels along its route path and stops at its stops. If a service line runs every ten minutes, each trip has its own trip id and is uniquely identifiable.

[8] https://developers.google.com/transit/gtfs/

While these data are too sparse for distance queries, we interpolate the possible positions of a vehicle along its path to ensure that the distance between two consecutive trip positions is always smaller than ten meter. On the one hand, this simplifies a query like ”find all trips in a distance of 20 meters around the user which arrive in plus minus one minute from now” a lot. On the other hand, the interpolation and denormalization inflates the data. Resulting in 324,440,457 trip positions annotated with arrival time and trip id, we enlarged the original data to a size of 38 GB.

This size leads the PostGIS index to its limit and prevents response times below 30 seconds for a service line detection query. Therefore, all data was partitionated horizontally in time dimension. The reason is, that all time tables repeat itself every 7 days and an usual query covers only a very small period of time at a specific weekday. Against this background, we divided the data into 24 parts for every day, which results in 7 times 24 subtables in the database. Finally, this setting archives average response times of less than one second for a whole service line detection query, containing of up to 200 single distance queries.

3.3 Traffic Jam Detection

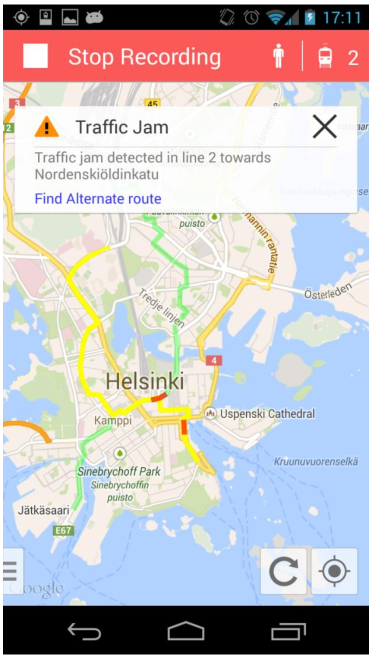
3.3.1 Related work

Traffic fluency monitoring and jam detection is part of traffic control in all major cities and the real time information about road network fluency is of interest to all network users and traffic related authorities. Over the last years, several different methods for detecting jams have been created. In many cities traffic is monitored via road side cameras detecting fluency in the most important channels of the network and sends the image to authorities and web services. Another increasingly popular method for detecting jams is by traffic message channels (TMC) that collect information about road network status from roadside sensors and vehicles and transmits the information via radio signals to the end user. TMC information is most commonly used in navigators, both on inbuilt vehicle navigating systems and separate navigator devices.[9] Over the past few years also different smartphone applications, such as Waze or Inrix have emerged on the market. In these applications information gathering is commonly done either with the vehicles connected to the system or by the users who report their journey conditions around them. Different applications provide different type of information, usually including one or several of the following information types: jams, weather, road work and accidents. What is common to all known methods for detecting jams, no service is provided specifically to the public transport users as most services are targeted for private vehicle drivers and the authorities. Based on our surveys no service is being used specifically to public transport nor uses the delay information of public transport vehicles, which makes the traffic jam detection module an unique tool.

3.3.2 Component Description

The jam detection module frequently polls vehicle location compared to the scheduled location from the Helsinki public transport service. The module analyses the vehicle situation and detects where there are traffic jams in the certain transportation authoritys transportation area. The

[9] http://www.tisa.org/technologies/tmc/



Stop Recording

Traffic Jam

Traffic jam detected in line 2 towards Nordenskiöldinkatu

Find Alternate route

Figure 15: Visualization of the current traffic situation in Helsinki based on Jam detection

interface returns always the current jam situation, so client does not need to keep the jams in memory.

The module detects the jams based the tram location, taking into notice the delay and how fast it increases. Also, the jam is only detected if there are several vehicles filling the same criteria within the same area.

The module provides information about the nearest stop to the detected jam and also the previous stop from where the affected vehicles have come from. Based on this information current traffic situation is visualized on the application. All of Helsinki city tram network can be drawn on the map, but only lines that are currently affected by trams are shown on the map at each time. If no jams are detected in between two stops, this interval is shown with green colored line. If a jam is detected between stops, the interval is shown with red colored line. Image 15 shows a screenshot of the application on how the jams are presented in Helsinki city center during the trial.

Architecture Description

The JamDetector module simplified architecture is visualized in the Figure 16. The public transportation systems generally differ in each city or area in multiple ways in the available data and data formats. The different public transportation systems are isolated in the module and configurable using the Spring Framework.

Live+Gov SaaS solution compatibility

The JamDetector follows Live+Gov SaaS architecture. This means that all connecting clients are authorized. Also the JamDetector provides the heart beat (health check) information to the SaaS service and sends the application log to the service.

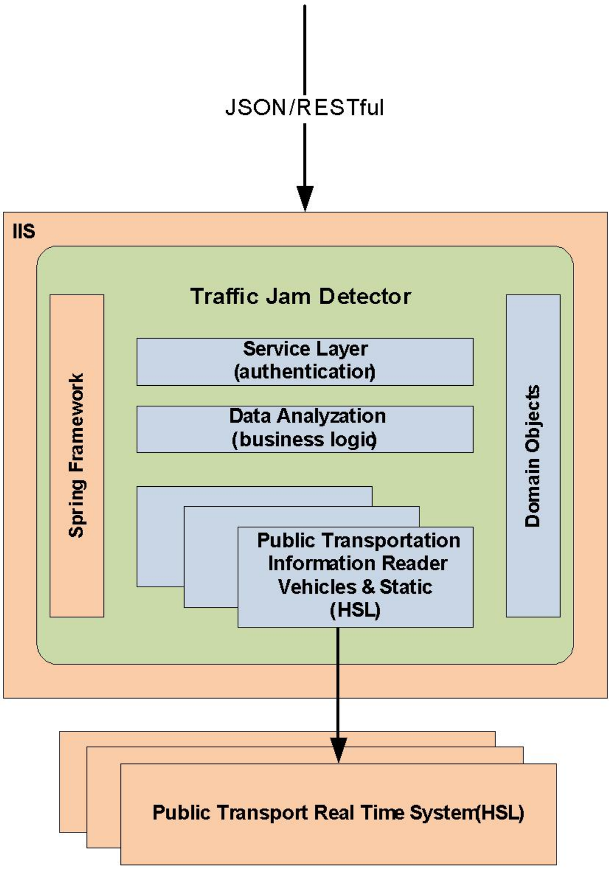

JSON/RESTful
IIS
Traffic Jam Detector
Spring Framework
Service Layer (authenticatior)
Data Analyzation (business logic)
Public Transportation Information Reader Vehicles & Static (HSL)
Domain Objects
Public Transport Real Time System(HSL)

Figure 16: Traffic Jam Module Architecture

Address	213.157.92.101:80
Path	/jamdetector/JamService.svc/jams/authority
Protocol	HTTP GET
Data format	JSON

Figure 17: Jam Detection Module Details

API description

The API lists the jams detected in the certain public transportation area in JSON format. The jam object contains information about the vehicles participating the detected jam and vehicles nearest stop/station. This information can be used to visualize the jam location for the users.

Data structure

The data structure of the JSON-message is described in figure 18.

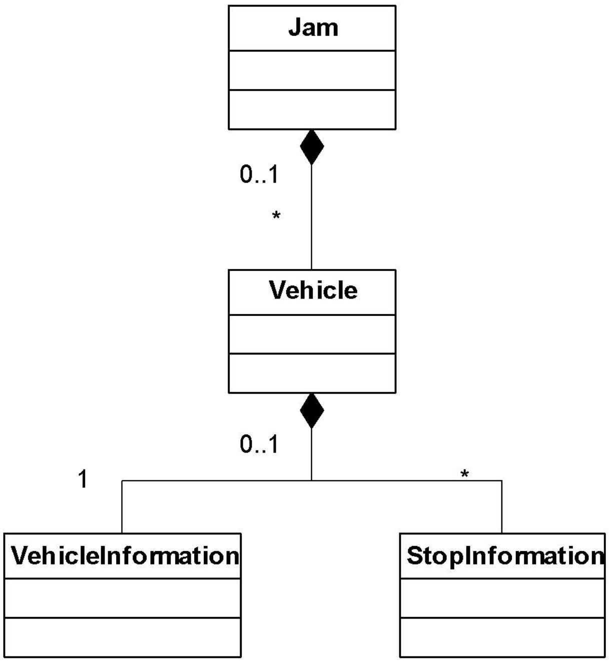


Figure 18: JSON data structure

Example of a given message:

```
URL /jamdetector/JamService.svc/jams/hsl

[ { "Id" : 0,
    "IsJam" : true,
    "SlowVehicles" : [ { "CumulativeDelay" : 13,
          "NextStop" : { "Code" : "1301453",
              "Latitude" : 60.197929999999999,
              "Longitude" : 24.876580000000001,
              "Name" : "Laajalahden aukio"
            },
          "PreviousStop" : { "Code" : "1301455",
              "Latitude" : 60.195390000000003,
              "Longitude" : 24.873429999999999,
              "Name" : "Tiilimki"
            },
          "Stop" : { "Code" : "1301455",
              "Latitude" : 60.195390000000003,
              "Longitude" : 24.873429999999999,
              "Name" : "Tiilimki"
            },
          "Vehicle" : { "Delay" : 13,
              "Id" : "RHKL00076",
              "IsOnStop" : false,
              "Latitude" : 24.873405999999999,
              "LineDirection" : 2,
              "LineId" : "1004",
              "Longitude" : 60.195535999999997,
              "NextStopIndex" : 2,
              "Time" : "20140116-102700",
              "Timestamp" : "/Date(1389860820413+0200)/"
            }
        },
        { "CumulativeDelay" : 72,
          "NextStop" : { "Code" : "1240419",
              "Latitude" : 60.203020000000002,
              "Longitude" : 24.96584,
              "Name" : "Kylsaarenkatu"
            },
          "PreviousStop" : { "Code" : "1230410",
              "Latitude" : 60.204529999999998,
              "Longitude" : 24.96998,
              "Name" : "Toukoniitty"
            },
          "Stop" : { "Code" : "1230410",
              "Latitude" : 60.204529999999998,
              "Longitude" : 24.96998,
              "Name" : "Toukoniitty"
            },
          "Vehicle" : { "Delay" : 82,
              "Id" : "RHKL00065",
              "IsOnStop" : false,
              "Latitude" : 24.968769999999999,
              "LineDirection" : 2,
              "LineId" : "1006",
              "Longitude" : 60.203980999999999,
              "NextStopIndex" : 3,
              "Time" : "20140116-102700",
              "Timestamp" : "/Date(1389860820405+0200)/"
            }
        }
      ],
    "SlowVehiclesInJamCount" : 2
  } ]
]
```

In this example the JamDetector has detected one jam and there are two vehicles in the jam. In addition to the vehicle information the next, previous and nearest stop information is returned. The jam location can be approximated either from the participating vehicle locations or stop locations. The interface contains more information than the current clients require. This is for debugging and possible visualization purposes. For example vehicles line information is such information.

If there are no jams currently detected, the interface returns:

```
[]
```

3.3.3 Evaluation

The module development begun with evaluating different methods for creating the detection algorithms and creating a way to detect jams reliably. The tram location data from Helsinki provides reliable information about the delay of a vehicle in real time and was considered the most efficient and reliable source of data for the jam detection.

A comprehensive and reliable detection algorithm was created through experimenting in the early stages of the module development. The parameters to be used were searched the most suitable values through thorough consideration in order to generate reasonable and accurate alerts that would correspond with the logical situation in traffic and would fit in the definition of a jam. After some weeks of monitoring the detections, the defined values were also re-evaluated and re-defined before the first pilot based on the previous observations.

Based on the first trial results, the amount of detected jams is seen relatively high, on average 3 jams were received on each time the API was called. This would indicate the parameters needing to be set tighter in the future if only greater jams are wanted to be detected. When evaluating the accuracy it was quickly discovered that validating the detection is rather challenging when not having the possibility to verify the situation on site. User questionnaires showed that no users were travelling in the areas where alerts were given at the time of the alert and therefore they could not reliably state their opinion about the accuracy of alerts. Also, no known major jams took place during the trial in the pilot area and therefore manual validation could not be done during the trial. Manual comparison to other known services providing jam alerts was done. In all cases the detected jams were of a short duration, mainly less than a minute and therefore no reliable statement about accuracy could be done at this stage. In the comparison we used two popular services: Waze[10], which is a mobile application providing traffic data in over 30 countries and V-traffic[11] , a national service in Finland for providing traffic data in major cities. Based on the comparison to other services the results seem promising as alerts were given in the same areas at the same time, only with minor differences.

When comparing the module to existing services it is obvious that there are existing systems that provide information with more geographical coverage and collect the data from greater number of sources. Thus, these services also provide more alerts as they also cover roads that are not included in the current module coverage, the Helsinki tram network. What needs to be addressed when comparing the module to existing services is that the other services do not focus on public transport and using only these services would not help to meet the requirements of the mobility

[10] http://www.inrixtraffic.com/

[11] http://www.v-traffic.fi/

use case in full. However, the possibility to support the jam detection module with existing services needs to be further studied.

We believe that when expanding the data to also cover other public transport vehicles, the coverage in the module will increase and the quality and the usefulness of the module improves. For example in the Helsinki region, all public transport busses, trams, trains and the ferry will be covered in the new public transport information system to be implemented in 2016. Then the module will provide a much better service with greater geographical coverage and expanded fleet. Unfortunately at the time of development, location data is not yet available for other vehicles than trams but the possibility to include these vehicles in the jam detection has been taken into notice from the start.

As no reliable evaluation is possible at this stage we will continue validating the module, analyzing the results and developing the algorithms. This process will continue until the final trial where improvements and reliable accuracy of the module are hoped to have been achieved.

3.4 Distributed Geo-Matching

Nowadays, the means of public transportation become equipped with physical sensors like GPS sensors or light barriers. Those sensors are used to keep track of the current vehicle position or to estimate the number of passengers. But those sensors do not detect, for instance, emergencies, accidents or the passengers' opinions on bus lines. To overcome these limitations, another kind of sensors is required.

In the last decade, many social networks or micro-blogging services like Twitter$^{\text{TM}}$arose. They are frequently used by humans to publish their opinions, observations, etc. which then can be requested by a web API. Therefore, the users of those services can be seen as social sensors. The increasing number of mobile phones leads to a situation in which more and more passengers become social sensors by publishing statements about crowded buses or rioting passengers.

The evaluation of the social sensor data requires that the bus or train can be identified to which a message is related. This task is simplified by the fact that most mobile phones have integrated GPS sensors. They measure the current geographical position, which is added to the published message together with a timestamp. This position information can be used to identify the nearest train or bus. A more detailed description of this matching problem will be given in the next section.

The amount of data produced by the different types of sensors can exceed the processing capabilities of a single computer. Therefore, a distributed approach is required, which will be explained in section 3.4.2. Finally, the results of an experiment with the current implementation are presented in section 3.4.3.

3.4.1 Problem of Matching Physical and Social Sensor Data

In order to find the bus or train a message is related to, the data measured by the physical sensors have to be processed first. In this context, the relevant data consists of the longitude and latitude ($\mathbb{R} \times \mathbb{R}$) measured by the GPS sensors of the vehicles. Before transmitting these data, they have to be extended by the unique id (V_{id}) of the current vehicle and a timestamp (T) to keep track of the chronological order. The set of all possible position is defined as *Pos* in equation (3.4).

$$Pos \quad := \quad V_{id} \times T \times \mathbb{R} \times \mathbb{R} \tag{3.4}$$

The relevant data of the social sensors are the messages published by the passengers. Each message contains a timestamp, i.e., the publishing time and the GPS position of the mobile phone from which the message was sent. The set of all possible messages *Mes* is defined in equation (3.5) where C is the set of all possible message contents.

$$Mes \quad := \quad T \times \mathbb{R} \times \mathbb{R} \times C \tag{3.5}$$

The data of the different sensors are transmitted as a not necessarily finite stream of data as defined in the equations (3.6) and (3.7).

$$posStream: \quad \mathbb{N} \rightarrow Pos \tag{3.6}$$

$$mesStream: \quad \mathbb{N} \rightarrow Mes \tag{3.7}$$

The range of *posStream*, i.e., the set of vehicle positions received by the data stream *posStream* are the trajectories of the different vehicles. A trajectory is defined in equation (3.8) as a discrete function mapping some point in time to a geographical position.

$$traj: \quad T \mapsto \mathbb{R} \times \mathbb{R} \tag{3.8}$$

But this definition is not adequate because the timestamps of messages may vary from the timestamps contained in the position data of the vehicles. Therefore, a continuous trajectory function $\widehat{traj}$ is required. This is reached by, for instance, a linear interpolation of $traj$.

Equation (3.9) shows the function *allTraj* which maps the unique identifier of a vehicle on its corresponding continuous trajectory.

$$allTraj: \quad V_{id} \mapsto (T \mapsto \mathbb{R} \times \mathbb{R}) \tag{3.9}$$

In order to relate a message *m* to a bus or train, the vehicle must be determined which has the smallest euclidean distance *dist* to the position of the sender at the point in time when *m* was sent. Furthermore, a vehicle has a maximum length *maxDist* and all vehicles with a greater distance from *m* can be ignored. Thus, each message can be seen as a range nearest-neighbour query *rnn* as defined in equation (3.10).

$$\begin{aligned}
&rnn: Mes \rightarrow Pos \cup \{\texttt{null}\} \text{ with} \\
&rnn((t, lon_m, lat_m, cont)) := (v_{Id}, t, lon_v, lat_v) \\
&\qquad \text{if } allTraj(v_{Id})(t) = (lon_v, lat_v) \wedge dist((lon_m, lat_m), (lon_v, lat_v)) \leq maxDist \wedge \\
&\qquad \nexists v'_{Id} \in V_{id} : dist((lon_m, lat_m), allTraj(v'_{Id})(t)) < dist((lon_m, lat_m), allTraj(v_{Id})(t)) \\
&rnn((t, lon_m, lat_m, cont)) := \texttt{null} \\
&\qquad \text{otherwise}
\end{aligned} \tag{3.10}$$

Putting it all together, the problem of finding the nearest vehicle for each message is defined as followed:

Problem Finding range nearest vehicle for each message
Input: *posStream*, *mesStream*, *maxDist*
Output: $rnnStream: \mathbb{N} \rightarrow Mes \times Pos \bullet rnnStream(i) := (mesStream(i), rnn(mesStream(i)))$

An implementation which solves this problem has to deal with the fact, that the data received by different sensors are not equally delayed. For instance, the data measured from physical sensors may be faster transmitted then messages received from social networks.

Another difficulty is, that the amount of data received by the different input streams may exceed the processing capabilities of a single computer. One solution for this problem would be to discard incoming data if the load of the single machine would be to high. This could lead to the loss of valuable data. In order to avoid this, a distributed approach is required which scales horizontally.

Some approaches like PLACE* [9] distribute the incoming data according to a static mapping of computers on geographical regions. The disadvantage of this static mapping can be seen, for instance, if there exists a sport stadium in some region. If a sport event takes place, then there are many vehicles and passengers producing a huge amount of data. Therefore, the corresponding computer can only deal with a relatively small region. But if no event takes place, this computer has almost nothing to do. In order to improve this, the mapping has to be dynamic, i.e., a dynamic load balancing is required.

3.4.2 Distributed Geo-Matching Approach

The schema of an intuitive stream-processing system which solves the problem described above can be seen in Figure 20(a). It consists of a single component (C) which receives the incoming data streams of vehicle positions and messages. It caches the current positions and processes the range nearest-neighbour queries initiated by the received messages. The answers are transmitted via the outgoing data stream $rnnStream$.

In order to answer range nearest-neighbour queries, C has to cache the current position of all vehicles. The problem of the delayed sensor data requires the caching of more vehicle positions than just the latest. Thus, a sliding window approach is implemented with a window size exceeding all regular delay times. This is realized by deleting all vehicle positions outside the window whenever a new position with a timestamp later then the timestamp of the latest cached position is received.

The data structure used as cache should answer range nearest-neighbour queries efficiently. In spatio-temporal databases R*-trees [5] are used for this purpose. Similar to a B-tree, the inserted elements are stored only in the leafs. Each node except the root has a minimal and a maximal number of children or elements. If an insertion is performed on a node already containing the maximal number of elements, then it is split and the resulting inner node is inserted in the parent node. In contrast to B-trees, each node of an R*-tree represents the minimal bounding rectangle (MBR) of all elements contained in its subtree as illustrated in Figure 19. The presented tree consists of a root node (dark gray), seven leafs (light gray) and several vehicle positions (black squares).

During the processing of a range nearest-neighbour query, only the children of a node are visited whose MBR intersects with the queried range. Thus, the performance of the query processing depends on the number of subtrees to be traversed. In order to reduce this number, the MBRs of the child nodes should be disjoint. As shown in [5] the algorithms used for insertion and deletion are designed to ensure a minimal overlap of MBRs. Another advantage is, that the MBRs are dynamically adjusted according to the currently contained elements.

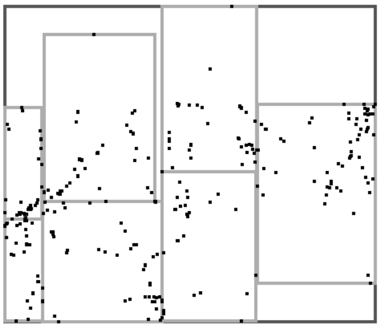
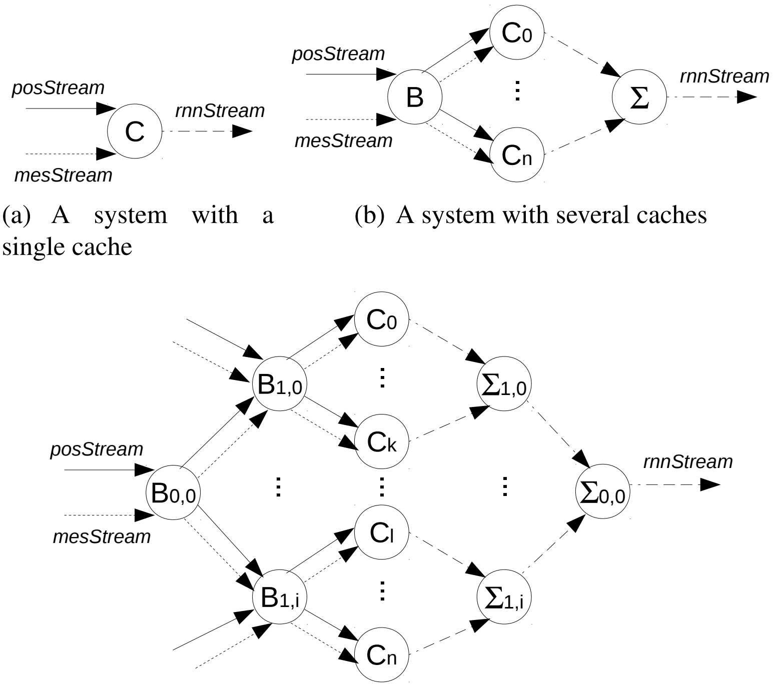


Figure 19: An examplary R*-tree



(a) A system with a single cache

(b) A system with several caches

(c) A distributed R*-tree

Figure 20: Distributing a geo-matching system

The problem of the system shown in Figure 20(a) is that its processing capabilities are quite limited. Therefore, a distributed approach is required which spreads the load on several computers. Figure 20(b) shows such a system. It consists of several caches C_0 to C_n each of them owning an own R*-tree for caching current vehicle positions and responding to queries.

Furthermore, this system consists of balancer component B. It receives the incoming streams of vehicle positions and messages and decides to which of the different caches each single datum is sent. Similar to the root of an R*-tree, it keeps track of the MBRs of the different caches. A new vehicle position is basically sent to the cache with the nearest MBR. If B receives a new message, it is only forwarded to the caches whose MBR intersects with the queried range. In the best case this is only one cache. But it could be several caches as well. Thus a special combiner component (Σ) is required which receives the nearest vehicle of each cache involved in the query processing and returns the overall nearest neighbour.

A further responsibility of B is the load balancing, i.e., the number of cached vehicle positions and queries in progress should be similar for all caches. Therefore, B counts the number of vehicle positions and messages forwarded to the different caches. The load balancing is done by distributing the received vehicle positions according to special target ranges for the different caches. In the case of a balanced system, these regions are equal to the MBRs. But if the load of a cache C_i exceeds the load of a cache C_j with a neighboured MBR, than the target range of C_i is reduced and the range of C_j increased. During the progress of the sliding window the MBRs of the caches will approach their target ranges.

In regular intervals, B sends special requests to all caches to send the information about their current loads and MBRs back (not shown in Figure 20(b)). If B receives the responses of all caches, it updates its calculated values. These update requests are performed asynchronously, i.e., the distribution of the incoming data is not blocked. This leads to a situation where the information received by the caches are not up to date any more. Therefore, B caches all forwarding decisions from the time the updated request is sent until all caches have responded. Then these decisions are applied on the received load information and MBRs.

The disadvantage of this system is, that the single balancer component B and combiner component Σ can become bottlenecks. A future solution could be found when thinking of B as the root of a distributed R*-tree whose children are the trees stored in the different caches. If it is close to its processing limits, it creates child balancer components which are responsible for subregions of the overall MBR. Each of the children get an input stream for all the data of its MBR. Data for regions which are not covered by the children are still received by the root. Whenever a balancer component is split, the combiner components are split analogously. Such a system is illustrated in Figure 20(c).

3.4.3 Experiment

3.5 Geographic Topic Analysis

The BuitenBeter dataset consists of 13,811 geo-tagged issues reported to the office of public order in the Netherlands between July and September 2012. Each issue is assigned to one of 16 categories such as ”poor road conditions”, ”Dirt on the street” or ”Graffiti”.

Geographical co-occurrences of arbitrary categorical observations - such as public order issues of the BuitenBeter dataset - can be used to discover latent underlying factors that explain observations which co-occur more frequently than expected by chance. One method for detecting these latent factors is probabilistic topic modelling, where grouped observations are modelled as being sampled from mixtures of latent topics, which are probability distributions over the set of distinct observations.

For the BuitenBeter dataset, we e.g. might expect that areas where graffities are reported will see many reports of the category ”Dirt on the street” since both categories are related to urban environments. We model those environments as hidden ”topics”.

Existing approaches for geographical topic modelling adopt topic models such as latent Dirichlet allocation [6] and extend the models by assigning distributions over locations to topics, or by introducing latent geographical regions. In models which extend topics for spatial distributions (such as two-dimensional normal distributions) [8], topics with a complex (i.e. non-Gaussian) spatial distribution cannot be detected. In models with latent, Gaussian distributed regions [10], documents within a complex shaped topic area do not influence the topic distribution of distant documents within the same area. Therefore, topics with a complex spatial distribution such as topics distributed along coastlines, rivers or country borders are harder to detect by such methods. More elaborate models introduce artificial assumptions about the structure of geographical distributions, e.g. by introducing hierarchical structures [1] in advance.

We introduce a novel geographical topic model which captures dependencies between geographical regions to support the detection of topics with complex, e.g. non-Gaussian distributed spatial structures [7].

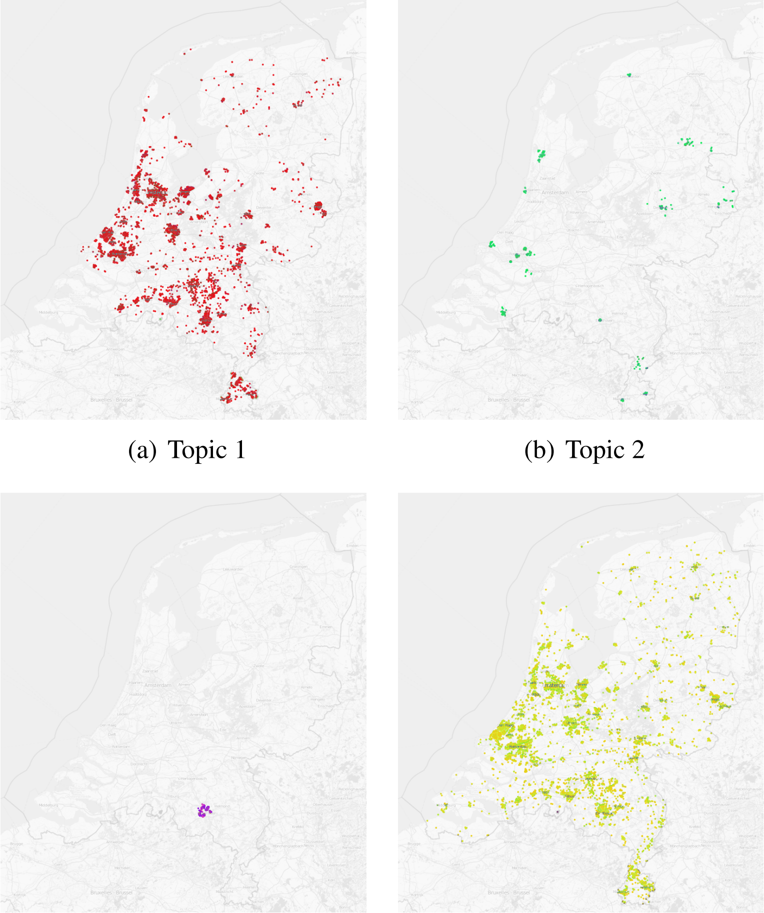
Our model differs from existing approaches in several aspects:

We model locations and words separately, as the separation of spatial clusters and document semantics allows us to define meaningful neighbour relations between spatially adjacent clusters. Our topic model takes a set of geographical clusters as input.

We also expect that geographical clusters adjacent in space exhibit similar topic distributions: Most geographical topics cannot be approximated by a simple spatial probability distribution such as a Gaussian distribution and for these complex topic areas, coherent sets of multiple spatial distributions are a reasonable approximation. Therefore we detect and smooth the topic distribution of these adjacent regions to increase the probability of detecting coherent topic areas.

The detection of geographical clusters is straight-forward: We use a mixture of a fixed number of Fisher distributions – distributions on a three dimensional unit sphere similar to isotropic Gaussian distributions on a plane. The parameters are fit using the approximation given by Banerjee et al. [4] in an expectation-maximisation algorithm. In our model, each cluster is associated with a distribution over the set of topics, sampled from a Dirichlet process (DP) with a common base measure which is itself drawn from a DP with Dirichlet distributed multinomial distributions over the set of categories as base measure. For the smoothing of topic distributions of adjacent clusters, we first define the adjacency relation using the Delaunay triangulation [3] of the cluster centroids. Each public order issue report draws an own topic distribution from a DP with a weighted mixture of the topic distribution of its own geographical cluster and its neighbour clusters as base measure. Finally, each observed report category is drawn from its document-specific topic distribution. For a detailed description of the model and its parameters, we refer to our paper [7].

To train our model on the BuitenBeter dataset, we use 300 clusters and the initial settings for the parameters $\beta = 0.5$, $\gamma = 1.0$, $\alpha_0 = 1.0$, $A = 99999$, $\delta = 1.0$. By setting A to such a high



(a) Topic 1

(b) Topic 2

(c) Topic 3

(d) Topic 4

Figure 21: Document positions of reports with above-average probabilities for Topic 1-4 on the map of the Netherlands.

value, we give up document-specific topic distributions and sample the categories directly from a mixture of region-specific topic distributions. All other parameters are set to the values used in the paper [7], except for the hyperparameters for A, which is fixed.

For creating a comprehensible report, the detected topics can be characterised by their most-likely categories:

Topic 1: *Dirt on the street*, *Other*, *Graffities*
Topic 2: *Damaged street light*, *Bad road*
Topic 3: *Obstacle by trees*, *Weed*, *Loose paving stones*, *Bad road*, *Idea/wish*
Topic 4: *Other*, *Weed*, *Loose paving stones*, *Bad road*, *Damaged street light*, *Idea/wish*

The position of documents with an above-average probability for each topic are shown in Figure 21 We see that, as expected, Topic 1 is only observed in the area of larger cities, whilst Topic 4 is present across the whole country. Topic 2 occurs mostly within city centres. Topic 3 is observed in the area of Eindhoven for which different categories were available in the BuitenBeter application.

4 References

[1] Amr Ahmed, Liangjie Hong, and Alex Smola. Hierarchical geographical modeling of user locations from social media posts. In *WWW*, 2013.

[2] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and JorgeL. Reyes-Ortiz. Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine. In Jos Bravo, Ramn Hervs, and Marcela Rodrguez, editors, *Ambient Assisted Living and Home Care*, volume 7657 of *Lecture Notes in Computer Science*, pages 216–223. Springer Berlin Heidelberg, 2012.

[3] Franz Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991.

[4] Arindam Banerjee, Inderjit S. Dhillon, Joydeep Ghosh, and Suvrit Sra. Clustering on the unit hypersphere using von Mises–Fisher distributions. *JMLR*, 6:1345–1382, 2005.

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM.

[6] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet allocation. *JMLR*, 3:993–1022, March 2003.

[7] Christoph C. Kling, Jerome Kunegis, Sergej Sizov, and Steffen Staab. Detecting non-gaussian geographical topics in tagged photo collections. In *WSDM*, 2014.

[8] Sergej Sizov. GeoFolk: latent spatial semantics in Web 2.0 social media. In *WSDM*, pages 281–290, 2010.

[9] Xiaopeng Xiong, H.G. Elmongui, Xiaoyong Chai, and W.G. Aref. Place: A distributed spatio-temporal data stream management system for moving objects. In *Mobile Data Management, 2007 International Conference on*, pages 44–51, 2007.

[10] Zhijun Yin, Liangliang Cao, Jiawei Han, Chengxiang Zhai, and Thomas S. Huang. Geographical topic discovery and comparison. In *WWW*, pages 247–256, 2011.