From Weak to Strong Geo Object Classification

Jörg Roth

Nuremberg Institute of Technology, Germany, Joerg.Roth@th-nuernberg.de

Abstract

Many geo sources are based on a weak classification to identify types of objects. Contributors can enter their knowledge about an object using tags of key-value pairs. Even though tags have no limits regarding expressiveness, automatic processing is difficult. We identified seven major problems where applications are not able to undoubtedly identify certain object types, have certain problems with subclasses or cannot detect object properties. As a solution, we propose the strong classification that strictly identifies classes, subclasses and properties. The major challenge is to map existing weakly classified objects to our strong classification. We introduce a mapping environment that uses a rule-based approach and provides tools to analyse the data source, rule set and mapping results.

Key Words: *geo data, geo object classification, strong classification, object tagging*

1. Introduction

Geo data is the 'raw material' for all types of location-based applications. Nowadays, geo data is available from a variety of geo data sources. The open geo data source *Open Street Map* (*OSM*) is currently very popular and collected approx. 26 million objects (as at Aug. 2013) only in the area of Germany.

The classification is besides the geometric shape the most important information about a geo object. It enables applications to distinguish, e.g., roads, shops, lakes or trees. For a map renderer, it controls how an object is painted (symbol icon, colour or line properties). For route-planning, the classification of a road defines the average and maximum speed, whether a certain means of transportation is allowed to drive and identifies one-way directions. For look-up-services, it allows a user to focus on certain object categories, e.g. hotels or restaurants.

Even though the classification is extremely important, many geo data sources only use a *weak* classification based on tags. User-defined tags are often difficult to understand by applications. In contrast, a *strong* classification allows an application to undoubtedly identify object classes. We often find strong classifications in data of land survey offices. The *ATKIS* schema of the German land survey office [1, 2], e.g., uses a four digit number to distinguish geo object classes. *Tiger/line* (US Census) [3] distinguishes approx. 800 object classes. However, even strong, these classes are not prepared to represent the huge variety of different object types we usually find in geo data sources nowadays. They are very restricted regarding subclassing and object properties. Most important, they do not support a mapping from widely available weakly classified data sources to their own representation.

2. Weak Geo Object Classification

A weak classification assigns a list of *tags* to each object; each contains a *key* and *value*. The first row in tab. 1, e.g., defines an Italian restaurant with certain daily opening hours. This classification is easy to read for people. Keys such as **`cuisine`** and **`opening_hour`** are predefined, but contributors have the flexibility to define new keys if existing keys are not suitable to describe an object. Usually, weak classifications are subject of community discussions and suitable key-value combinations are only loosely defined by guidelines. The drawback: the non-formal notation is difficult the process by software and keys do not have an obvious meaning for applications. We discuss the problems of weak classifications using the example of Open Street Map.

2.1 Open Street Map

The Open Street Map project follows the community idea to collect geo data. Consequently, the weak classification approach is ideal for typical contributors. As many OSM users contribute at irregular intervals, an easy classification schema is important:

- Web-based guidelines [4] provide typical classification patters. Guidelines often look like 'for a certain tag s_1=v_1, there should be a tag s_2=…'. E.g. 'for **`amenity=restaurant`**, there should be **`cuisine=`**… to identify the type of restaurant'.
- During the online-contribution process, the user interface can suggest a list of keys for a specific object. For this, a user can inspect similar objects in the database.
- Users may ignore suggestions and guidelines and can enter arbitrary new tag combinations. The system accepts such contributions. Maybe they will be changed by other users.

The last point is both advantage and disadvantage: on the one hand, a community system must be able to accept new object types, thus must be extendable. On the other hand, new keys or tag combinations have to be learnt, both by users and software.

Tab. 1: OSM classification examples

Tags	Meaning
`amenity=restaurant, cuisine=italian, opening_hours=17:00-24:00`	an Italian restaurant with certain opening hours
`highway=motorway, lanes=2, layer=1, lit=no, maxspeed=80, oneway=yes, surface=concrete`	a motorway with some properties
`amenity=cinema, phone=+49911…, wheelchair=yes, name=CineABC`	a cinema with name and phone; accessible by wheelchairs
`amenity=post_box, collection_times=Mo-Fr 16:00; Sa 13:45, operator=Post`	a post box with collection time and operator

Tab. 1 shows some classification examples. Looking deeper into the meaning of these tag combinations, we can identify three different tag roles:

- a *main class* (e.g. **`amenity=..., highway=...`**), represented by a certain key assigns a top-level classification;
- *subclass* tags (e.g. **`amenity=restaurant`** in combination with **`cuisine=italian`**) specify the object class more specific;
- object *properties* (e.g. **`opening_hours=17:00-24:00`**, **`wheelchair=yes`**, **`lanes=2`**) specify further attributes of the object within a certain object class.

The classification can roughly be compared to object-oriented software components, with the notion of classes, subclasses and attributes. Note that there is a smooth transition between properties and subclasses, also for geo objects. As an example: we could also introduce a two-lane highway as a certain subclass, avoiding the number of lanes as property. As in object-oriented development, the role of properties is usually obvious for a certain usage scenario.

2.2 Weak Classification Problems

The weak classification has certain drawbacks, especially, if applications should automatically understand and process classifications. The major drawbacks are:

Tag ambiguity

For a certain tag inside a classification, it is difficult to detect its role (main class, subclass, or property). Object tags are not ordered. We, e.g., cannot expect the first tag to indicate the main class. We also cannot build a dictionary of keys and their role, because many keys play different roles in different combinations. An example is the **`building`** key: it may indicate a main class, if no other tags follow. It can be a property of objects that both can exist with or without a building. Finally, it may indicate a certain subclass of a general building.

Non-uniqueness

Contributors may assign different sets of tags to express the same meaning. This can be either because the guidelines changed after a contribution has been stored or there is a certain overlap of classifications. E.g.:

- for coffee bars we find the tags **`amenity=cafe`** or **`shop=coffee`** as well as further variations;
- combined foot and bicycle ways can either be **`highway=cycleway, foot=yes`** (actually a cycleway that also allows pedestrians) or **`highway=footway, bicycle=yes`** (actually a foot way that also allows bicycles).

As a result, it is difficult to define equality of classes.

Over-classification

Objects often contain redundant tags that do not provide additional information as a certain class implicitly defines some properties. E.g. for motorways, the tags **`bicycle=no`**, **`foot=no`**, **`horse=no`** are not required as these means of transportation are not allowed on motorways. But nevertheless, they are often stored. Another example: even though there exists a default speed limit in cities (e.g. 50 km/h in Germany), some road objects explicitly state this speed limit. Such redundant information is a problem for automatic processing as these tags have to automatically be removed.

Polymorphic objects

Some geo objects have more than one facet, as they actually represent multiple objects by a single geometry. E.g. some stationary shops offer a post office counter, many hotels are with a restaurant, or touristic viewpoints may reside on a mountain peak. Contributors can deal with such objects in two ways: they may create two objects at the same place (i.e. with the same geometric shape) or they may create one object that holds tags of both object facets. Even though we can find both ways in typical sources, the second way reflects much more the reality, as it respects the object identity. If, e.g. the stationary shop offers a post office counter, changing and removing the database entry should always affect both facets, as they actually belong to a single object. For the weak classification polymorphic objects may cause a problem, as it is not obvious which tag belongs to which facet. This is a problem in particular for object properties.

Obscure subclasses

Sometimes it is easy to check, whether a classification describes an upper or subclass of another class. E.g. **`amenity=restaurant`** without a cuisine tag describes any restaurant, thus can be considered as upper class of *Italian restaurant*. Unfortunately, upper classes often use completely different tag combinations, thus cannot easily be discovered. As an example:

- a park area is indicated by **`leisure=park`**;
- a barbecuing area is indicated by **`amenity=bbq`**, **`fireplace=yes`**;
- a picnic area is indicated by **`tourism=picnic_site`**.

Even though we could consider barbecuing and picnic places as subclasses of park areas, their tags have nothing in common; even worse, their main class keys are different. As a result, there is no effective way the discover subclass objects.

String-related issues

Storing tags means always to store strings. Strings have certain drawbacks:

- They require a high amount of space. This is a problem, if we want to use small devices such as smart phones to store geo data. This also may be a problem for communication.
- They are more difficult to search in databases. Consider an SQL statement that looks up all Italian restaurants that are accessible by wheelchair. The **`WHERE`** clause would contain complex **`LIKE`** conditions, and may require programmatically filtering the database results. In addition, strings are only partly supported by column indexes, thus such queries would execute very slowly.

We could integrate textual search engines and compression techniques on top of a database. However, these would not solve all problems.

Contributor-related issues

The weak classification shifts most of the responsibility for correct categorization to contributors. As guidelines are only informal without the ability for a formal check, a contributor should carefully read and respect the guidelines to formulate tags. As a certain classification cannot be automatically be checked, many tags are wrong or misleading. Even worse: as the guidelines evolve over time, a contributor should study these guidelines again and again for any new object and should correct old contributions, if they do not meet new guidelines anymore.

Note that contributors, even willing, are not always able to spend much time whenever they enter new geo objects. As a consequence, the data source does not always reflect all guidelines. A typical snapshot contains objects of different classification 'ages' and even out-dated guidelines left their foot prints in the data set.

3. Strong Geo Object Classification with HomeRun

As a result of the problems above, equality of classes and subclass relations are difficult to check, and it is difficult to identify object properties. String-based tags are not suitable for efficient processing and storing. Finally, tagging guidelines often are not respected or out-dated. To solve these problems, we suggest a strong classification approach as used in the *HomeRun* project [7, 8]. It includes a model to classify objects and to build up class hierarchies, rules to control the mapping of weak to strong classification, a tool environment to analyse classifications in the data source and to check the mapping results. Finally, it provides a software library to execute the mapping and to access class properties at runtime.

3.1 The Classification Model

The first improvement is: our model actually *has* classes. The huge number of possible tag permutations in the weak classification contradicts the idea of classes. In contrast, our strong classification model propagates *classes* with *subclasses* and *properties*.

HomeRun geo objects are based on *single inheritance* and allow *multiple classes per instance*. Note that the approach of multiple classes per instance does not have an analogy in object-oriented software classes, especially should not be confused with *multiple inheritance* (multiple upper classes per class).

We assign multiple classes to address the problem of polymorphic geo objects. In principle, we could create complex, combined upper classes for such objects, e.g. 'Hotel/Restaurant' or 'Stationary shop/ Post office'. But in real geo databases, polymorphic objects appear in huge numbers of different combinations, thus the amount of complex upper classes would be very high. Even though multiple classes per instance increase lookup costs, this approach achieves the required expressiveness.

We use integer numbers to identify classes. Equality of classes is mapped to equality of numbers. A class c_{sub} is subclass of c (denoted $c_{sub} \lhd c$) iff there exists i>0 where c mod $10^i = 0$ AND $0<c_{sub}$-c$<10^i$. As an example: 20501257 (*Italian Restaurant*) ◁ 20501250 (*European Restaurant*) ◁ 20501200 (*Restaurant*) ◁ 20501000 (*Gastronomy*). Fig. 1 shows another example – the subtree to model roads.

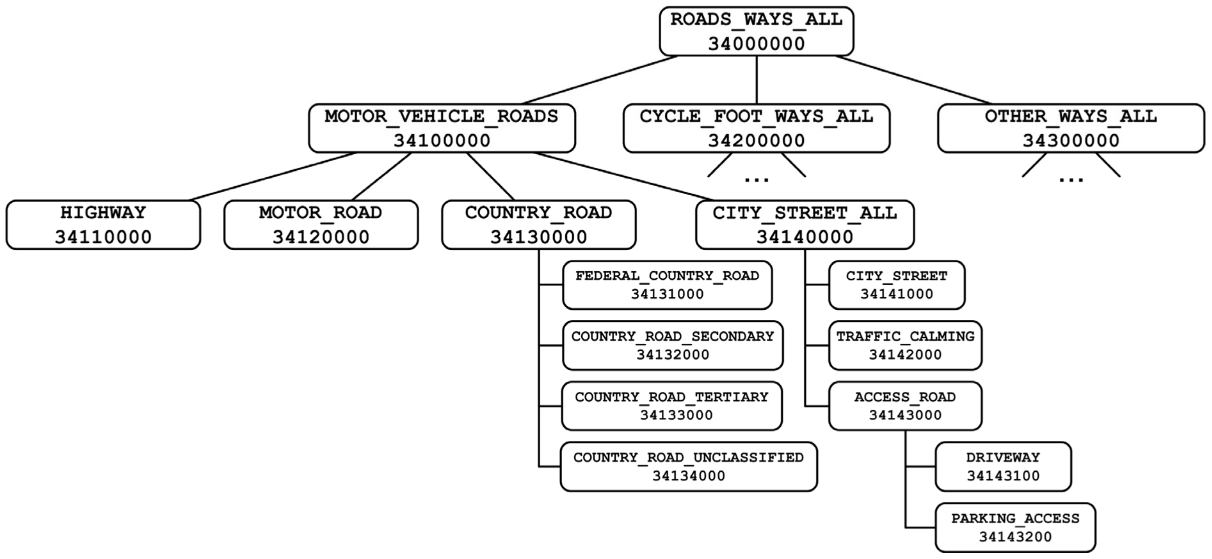

Fig. 1: Part of the class tree

Note that the number of direct subclasses is not limited to 9, as we can also reserve more than one digit for subtrees. Also note that the maximum height of the tree is limited by the maximum number of digits in the number representation (currently 19 digits for long integers). However, current trees are far from reaching this limit. Important benefits of this approach are:

- Equality of classes can be checked by number comparison.
- Subtrees can be represented by number intervals. All city roads: [34140000, 34149999], all roads for motor vehicles: [34100000, 34199999] and all road: [34000000, 34999999].
- Once mapped to our numbers, we solved the obscure subclass problem.
- Numbers are more appropriate than strings regarding memory and communication.
- Numbers can benefit from efficient index mechanisms, e.g. in SQL tables.

Even though numbers in object records fully represent a class membership, developers can use symbolic constants in their programs (e.g. **COUNTRY_ROAD** instead of **34130000**). These constants are automatically generated by the tool environment and improve readability. Note that constants do not affect the efficiency as they are internally are represented as numbers. As another benefit: software is more independent from changes of the class tree.

3.2 Mapping Rules

Mapping rules transfer a weak classification based on tags into our strong classes including properties. Rules follow the pattern *Conditions* → *Assignments* as shown in fig. 2.

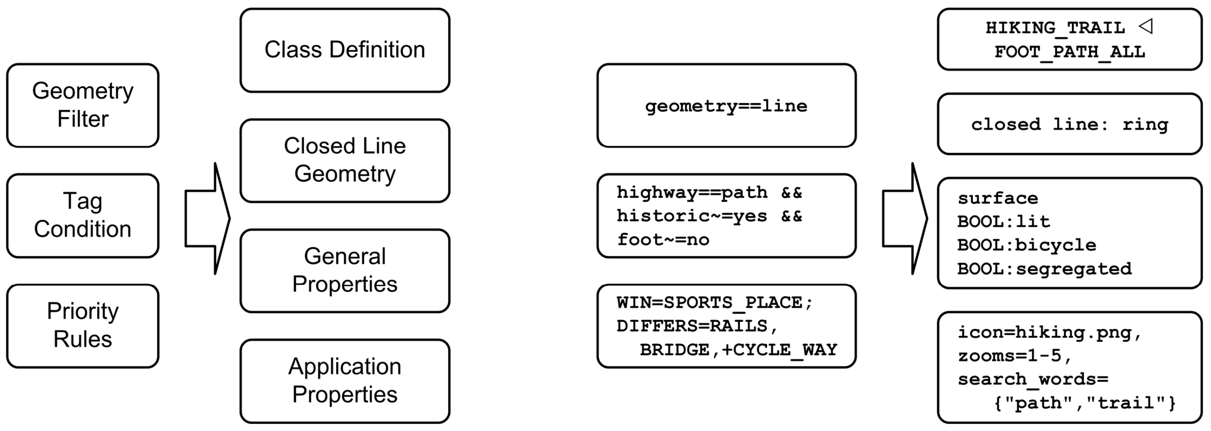
Geometry Filter
Tag Condition
Priority Rules
Class Definition
Closed Line Geometry
General Properties
Application Properties
geometry==line
highway==path && historic~=yes && foot~=no
WIN=SPORTS_PLACE; DIFFERS=RAILS, BRIDGE,+CYCLE_WAY
HIKING_TRAIL ◁ FOOT_PATH_ALL
closed line: ring
surface BOOL:lit BOOL:bicycle BOOL:segregated
icon=hiking.png, zooms=1-5, search_words= {"path","trail"}

Fig. 2: Mapping rules pattern (left), rule example (right)

The first condition is the *Geometric Filter*. Classes may only be applied to certain geometries. E.g. lakes are areas (not lines), roads do not have point geometries and traffic signs are represented by single points. OSM defines four geometry types: points, lines (open and closed) and multi-geometries (called *relations*). The geometric filter can be any non-empty subset of these four types.

The *Tag Condition* is a boolean expressions *key↔value(s)* where ↔∈{==, **!=**, **^=**, **~=**} (see tab. 2). They can be combined by logical **&&** and **||** as used in programming languages. The right side can be a single value *v*, a list of values {v_1,…v_n} or the wildcard *. Examples:

Bus stop:

```
(bus==yes && public_transport=={stop_position;platform} ||
amenity==bus_station || highway==bus_stop) && bus^=yes &&
public_transport^={platform;stop_position}
```

Cycle track:

```
(highway==cycleway || cycleway==* || highway==path &&
bicycle=={yes;designated} && foot==no) && foot~=yes
```

Besides the well-know conditions in programming languages == and **!=**, we request two more as shown in tab. 2. These in particular address over-classification issues and create much more condensed and readable expressions.

Tab. 2: Different comparisons in the tag condition

	Test on equal		Test on unequal	
Key must exist	`==`	e.g. `amenity==restaurant` – the class expects this combination	`!=`	e.g. `amenity!=restaurant` – the key `amenity` must appear and must be different to `restaurant`
wildcard	`==*`	the key must appear	`!=*`	not allowed
Key does not have to exist	`^=`	e.g. `amenity^=restaurant` – the key `amenity` does not have to appear, but if, the value must be `restaurant`	`~=`	e.g. `amenity!=restaurant` – the key `amenity` does not have to appear, but if, the value must not be `restaurant`
wildcard	`^=*`	is always true, used to consume a tag, e.g. `foot^=*` marks any `foot` tag as 'used' for this rule	`~=*`	key must not appear, e.g. `foot~=*` means: no `foot` tag is allowed for this class

A weak classification may fulfil more than one rule. The mapping system has then to clarify, whether multiple hits are a result of polymorphic objects, over-classification or if the rule conditions unintentionally overlap. *Priority Rules* can express in a fine granular manner for pairs of two hit classes:

- one class *wins* against another, i.e. of two hits one will removed from the result; or
- the two classes intentionally *differ*, i.e. both classes are furthermore considered as result classes (usually for polymorphic objects).

The corresponding **`WIN`** or **`DIFFERS`** expressions can be applied to certain classes or entire subtrees (indicated by **`+CLASS`** in fig. 2 right). Based on these three conditions, rules are processed as follows:

- Compute the set of classes for which the geo object fulfils the geometric filter and tag condition.
- Eliminate all classes of the set that are super classes of another class in the set. This is because in a subtree the lowermost class is most meaningful.
- Eliminate all classes of the set that lose against another class because of priority rules.

Usually, only one class should remain unless we have a polymorphic object. If a rule remains in the result set, the *assignments* are performed (fig. 2):

- *Class Definition*: the actual class number and symbolic constant (e.g. **`HIKING_TRAIL`**). The mapping mechanism automatically computes a free class number.
- *Closed Line*: as a special problem of OSM geometries, areas and linear rings have the same representation. A contributor may clarify (using the tag **`area=yes`**) but is not forced to do it. This assignment tells: a closed line is meant as ring (e.g. loop roads) or as area (e.g. a lake, estate or place).
- *General Properties*: this is a list of all keys that are meant as object properties. The values may be restricted to e.g. boolean values, numbers, speeds in km/h, distances in meters, or weights in tons. Currently we support 25 property types.
- *Application Properties*: further application-dependent properties are defined for map rendering (e.g. icons), search tools (e.g. class terms and synonyms) and route planning (e.g. road properties).

After an object undergoes this procedure, an application can access the defined values for own computations. A small software library (even available for smart phones) provides access to all results.

3.3 The Tool Environment

The tool environment has three goals: first, it analyses the weak classified data source, second, it checks the mapping rule set and third, it analyses the strong classified target data. Checking the rule set means to perform basic consistency tests that e.g. detect circular win rules or detect tag conditions that never can be true. We focus on the other two goals.

A tool analyses real data and produces so-called *co-occurrence trees*. In a first step, key-value pairs of all geo objects in the source are collected. Keys are then classified by occurred values. If, e.g., mainly yes/no values appear for a certain key, it seems to be a boolean property tag. If a small list of constants appears for a key, it usually is a main class or subclass key. If a huge number of different values appear, it usually is a text property (e.g. name or description).

For each class and subclass tag, a co-occurrence tree is displayed as shown in fig. 3. Co-occurrence trees indicate common tag combinations for typical weak classifications at a single glance.

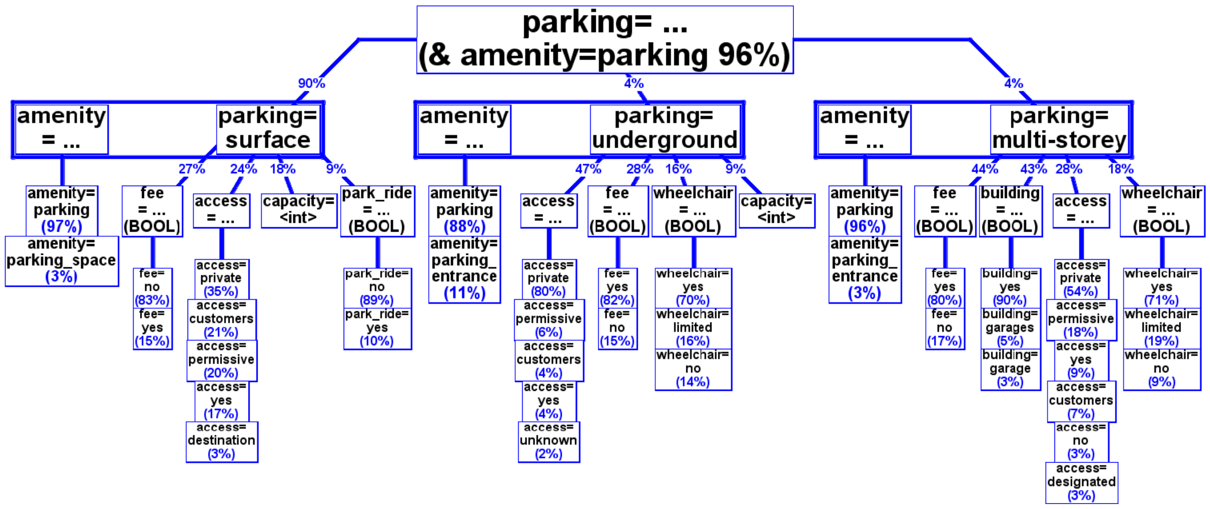

Fig. 3: The co-occurrence tree for the key 'parking'

The tree in fig. 3 inspects the key **parking**. It usually appears in combination with **amenity=parking** (96%) and most often has the value **surface** (90%). This indicates **parking** to be the subclass tag of **amenity**. Typical further tags are **fee=yes/no**, **capacity=<integer>** and **park_ride=yes/no**. Besides **surface**, we find **underground** and **multi-story** with different related tags. Based on these observations we could create a class subtree for 'parking':

- We assign a top level class **PARKING**, if only **amenity=parking** appears in the source.
- This class has three subclasses **SURFACE_PARKING**, **UNDERGROUND_PARKING** and **MULTISTOREY_PARKING**.
- All classes have the properties **fee** (boolean), **access** (list of constants) and **capacity** (int); **SURFACE_PARKING** additionally **park_ride** (boolean); **UNDERGROUND_PARKING** and **MULTISTOREY_PARKING** additionally **wheelchair** (boolean) and **building** (boolean).

Note that we are free to model other trees. E.g., we could consider the 3% of tags **amenity=parking_space** and model more classes. We also could merge underground and multi-storey parking (each only 4% of tags) to a class **NONSURFACE_PARKING**.

As the OSM classification quickly evolves, the number of different classifications dramatically increases. The co-occurrence tree is important to always capture the actual state of how guidelines are considered by contributors. We could, e.g., easily discover, if new important subclasses or properties were introduced.

Currently the rule set contains approx. 800 rules with some thousand subconditions. A second tool evaluates the mapping results. Analyses provide insight on the effectiveness of the mapping, especially, if there is the need to extend the set of classes or to adapt the conditions. Tab. 3 summarizes the analyses and activities to improve the rule set.

Tab. 3: Strong classification target analyses and activities

Analyses	**Objective**	**Activity**
Detect classes with many or few instances	Some classes may not be appropriately split into different subclasses.	Re-arrange the class tree.
Detect instances with many assigned classes	Usually, polymorphic objects do not have more than four assigned classes. Too many hits may complicate further processing.	Improve priority rules to reduce multiple class hits or create new classes for polymorphic objects.
Statistics on unused tags of objects	Unused tags of objects may be properties or subclass tags; a contributor usually wants all tags to be considered.	Improve tag conditions or property lists.
Statistics on property type conflicts	Properties are typed as boolean or integer, but some contributor may use other values.	Adapt property types or ignore malformed values.
Detect tag combinations with no hit classes at all	For some tag combinations no rule condition may be true. Either conditions are too restrictive, these objects are not represented as strong classes or contributors created illegal tags.	Introduce new classes, relax some of the conditions or ignore these tags.

4. Discussion and Results

Our approach is a development of older classification approaches [5, 6] and solves some drawbacks related to multiple classifications and rule inconsistencies. Once set up, the mapping mechanism automatically processes several million of weak classifications during an import and produces only a small report of critical cases. Of 26.5 million objects (Germany, Aug. 2013) 24.6 million are mapped to at least one strong class. Of the non-mapped objects 1.7 million are not tagged at all in the source. For only 248 thousand tagged objects (1%) our system failed to map, but many of these objects were incompletely or inconsistently tagged by contributors. 317 thousand objects were identified as polymorphic. A majority of 9.2 million objects are general buildings, 6.9 million are different types of roads. More detailed objects followed far behind, e.g. 393 thousand power poles, 319 thousand forests, 305 thousand single trees.

The strong classification forms the basis for the map renderer *dorenda*, the route-planning service *donavio* and the search tool *HomeRun Explorer* [7, 8]. Our classification significantly simplifies the development of these tools and services. Former implementations that directly based on the weak classification turned out to be too inflexible and complex. We especially want to shift the knowledge about the rapidly changing geo data source away from applications to a single rule set.

Besides the mentioned tools, we used the strong classification in our annual course '*Location-based services and applications*' (Nuremberg Institute of Technology, CS department), where students have to implement location-based applications in three-month projects. As a major benefit: the students could quickly focus on application functions (e.g. solve the geometric or topologic problems) and did not have to deal with the interpretation of classification tags.

Even though our class mapping robustly works, there still is a problem with values of property fields. We can identify two problems. First, some property values formulate complex structures by strings. For e.g. **`opening_hours`**, the contributor wants to express daily opening hours with breaks, different opening hours for weekdays or exceptions for holidays. The results may be very complex such as

```
Mo, Tu, Th, Fr 06:00-13:00, 16:00-18:00; We 08:00-13:00;
     Sa sunrise-sunset; Dec 25 off; week 2-20 10:00+
```

Applications usually not only display such strings, but want to process them (e.g. want to decide whether a shop currently is open). As there is a huge variety of different properties and formats, our platform should support an application to process such properties.

A second problem: for unstructured fields (e.g. names, descriptions), users tend to use different strings to indicate the same value (e.g. variations of upper/lowercase, abbreviations, usage of '-' or blanks). An application thus has difficulties to identify same values. However, this problem can often only properly be solved by the data source.

5. References

[1] Arbeitsgemeinschaft der Vermessungsverwaltungen der Länder der Bundesrepublik Deutschland (AdV): ATKIS Homepage, http://www.atkis.de/

[2] NÖV – Nachrichten aus dem öffentlichen Vermessungsdienst NRW, Vol. 2-3, 1987

[3] Tiger – Topologically Integrated Geographic Encoding and Referencing, US Census, http://www.census.gov/geo/maps-data/data/tiger.html

[4] Open Street Map, MapFeatures, http://wiki.openstreetmap.org/wiki/Map_Features

[5] Roth, J.: Modelling Geo Data for Location-based Services, 3. GI/ITG KuVS Fachgespräch "Ortsbezogene Anwendungen und Dienste", 7.-8. Sept. 2006, Berlin

[6] Roth J.: Übernahme von Geodatenbeständen aus Open Street Map und Bereitstellung einer effizienten Zugriffsmöglichkeit für ortsbezogene Dienste, Praxis der Informationsverarbeitung und Kommunikation (PIK), Vol. 13, No. 4, 2010, 268-277

[7] Roth J.: Modularisierte Routenplanung mit der donavio-Umgebung, in Werner M., Haustein M. (Eds.): 9. GI/ITG KuVS Fachgespräch "Ortsbezogene Anwendungen und Dienste", Sept. 13-14 2012, TU Chemnitz, Universitätsverlag Chemnitz (Germany), 2013, 119-131

[8] Roth J.: Combining Symbolic and Spatial Exploratory Search – the Homerun Explorer, Innovative Internet Computing Systems (I^2CS), Hagen (Germany), June 19-21, 2013, in press