Antoine Kalmbach

Fleet Inference

Importing Vehicle Routing Problems Using Machine Learning

Master's Thesis in Information Technology

June 12, 2014

University of Jyväskylä

Department of Mathematical Information Technology

Author: Antoine Kalmbach

Contact information: ane@iki.fi

Supervisors: Tuukka Puranen, and Jussi Rasku

Title: Fleet Inference Importing Vehicle Routing Problems Using Machine Learning

Työn nimi: Kalustonpäättelyn hyödyntäminen reitinoptimoinnissa

Project: Master's Thesis

Study line: Software Engineering

Page count: 111+0

Abstract: This thesis studies the use of automated reasoning in speeding up the process of converting vehicle routing problem data into data that is understood by a system that optimises them. The vehicle routing problem is a combinatorial optimisation problem, and we call the optimising system a *solver* for short. In this thesis, we consider a solver a program that functions using the software-as-a-service paradigm: problem descriptions are entered into the system, and the solver produces an optimised version of the problem.

Traditionally, solvers require the problem descriptions to be in a particular data format. Such data usually exists in other formats, and a great effort must be put in converting them to the accepted format. This is usually done manually by operations researchers, and such conversion can be onerous and time-consuming. In light of this, we study the use of machine learning in creating a system that can understand a variety of input data formats and convert the source data into one target format, letting operations researchers shift their focus away from demanding data processing tasks.

To this end, we implement such a framework, titled *fleet inference*, using machine learning. The former finds links between data files, usually column oriented CSV or Excel® files, and the latter pairs source data entities into target entities.

This thesis implements fleet inference using two separate modules—join inference and attribute classification. The framework consists of an automated classifier that is shown how optimisation problem data is structured, after this training the classifier can be used to understand structure in an otherwise seemingly unstructured data set. After a structure in these files has been obtained, we try to match data in them to data a vehicle routing problem solver needs—e.g., the capacities of vehicles available in the problem.

This system was implemented using a variety of classification techniques, and we present careful evaluations and introduce readers to the concepts of classification and data integration, all the while showing the apparent benefits of what automated reasoning can produce when faced with onerous data processing scenarios.

Keywords: fleet inference, join inference, data integration, machine learning, vehicle routing problem, data exchange, attribute classification, operations research

Suomenkielinen tiivistelmä: Tämä pro gradu -työ tutkii automaattisen päättelyn hyödyntämistä reitinoptimointiongelmien ratkaisemisessa. Reitinoptimointiongelma on kombinatorinen optimointiongelma, jonka ratkaiseminen edellyttää nk. *ratkaisujärjestelmän* luontia. Ratkaisujärjestelmä toimii ratkaisupalveluna, johon syötetään ongelman tiedot ja järjestelmä tuottaa ongelmasta optimoidun version.

Tämä toimintaketju alkaa ongelman tietojen tulkitsemisella. Tässä työssä esitellään menetelmä tämän askeleen nopeuttamiseksi. Koneoppimisella luodaan järjestelmä, jolle opetetaan esimerkkejä näyttäen miltä reitinoptimointiongelman data näyttää. Menetelmä on kaksiosainen: datasta etsitään rakenne sisäisten viittauksien ymmärtämiseksi ja kun datan rakenne on tulkittu, yhdistetään datassa löytyvä tieto vastaamaan varsinaisen optimointiongelman tietoja.

Aiemmin tämä askel on sisältänyt paljon käsityötä. Lisäksi optimointiympäristöt ovat edellyttäneet, että optimointiongelmat syötetään ratkaisijoihin tietyssä ja vain tietyssä muodossa. Datan muuntaminen tähän muotoon on vaivalloista. Siksi tässä gradussa esitellään tapa, joka automaatiota käyttäen säästää aikaa ja vaivaa operaatiotutkijalta.

Tämän ratkaisemiseksi gradussa tutkitaan *kalustopäättelyä* koneoppimista käyttäen. Kalustopäättely koostuu liitospäättelystä ja attribuuttiluokittelusta. Liitospäättely analysoi hajautetussa muodossa olevan datan, esimerkiksi useassa Excel® tai CSV-tiedostossa sijaitsevan datan, keskinäiset viitteet ja muodostaa näistä rakenteen. Rakenteen muodostamisen jälkeen datasta löydetään se tarvittava tieto, jota optimointiin edellytetään—esimerkiksi datasta tarvitaan kalustoon kuuluvien autojen kapasiteetit, jotta ajoneuvot voidaan järjestellä oikein optimoinnissa.

Ratkaisu koostuu pitkälti menetelmästä, jossa algoritmia opetetaan näyttämällä esimerkkejä siitä, miten liitospäättelyssä liitokset muodostuvat ja miltä kohdeattribuutit näyttävät attribuuttiluokittelussa. Toisin sanoen, algoritmi opetetaan ymmärtämään miten datan sisäiset viitteet toimivat ja miten nämä kuvautuvat reaalimaailmaan eli lopputulokseen.

Esitelty ratkaisu on toteutettu erilaisin koneoppimisen menetelmin. Tässä työssä käymme läpi ratkaisun ymmärtämäisen vaadittavan teorian sekä testaamme kalustonpäättelyä konseptina läpikotaisesti. Tutkimme ensisijaisesti sitä, miten automaattisella datan käsittelyllä voidaan helpottaa vaativien optimointiongelmien ratkaisemista ja miten sellainen järjestelmä toteutetaan.

Avainsanat: kalustopäättely, liitospäättely, dataintegraatio, datansiirto, koneoppiminen, reitinoptimointiongelma, attribuuttiluokittelu, operaatiotutkimus

Preface

This thesis wouldn't have been possible without the help of many people. First and foremost, my supervisors, Jussi Rasku and Tuukka Puranen, deserve my fullest gratitude for their guidance and advice. Without them, this thesis would not even exist: this thesis is a product of their brainstorming session where I was present first as a curious observer and then as an eager volunteer shortly thereafter.

I would also like to thank Jukka Kemppainen and Antti Sieppi for their helpful comments.

The software engineering part of this thesis would not have been possible to do if not for the existence of many free high-quality scientific software. I wish to express my gratitude to the authors of the following tools: the pandas data analysis library (McKinney 2011); the Scikit-learn machine learning library (Pedregosa et al. 2011); SciPy and Numpy (Jones, Oliphant, and Peterson 2001; Oliphant 2007): the IPython enviroment (Perez and Granger 2007). All graphical plots were made using Matplotlib (Hunter 2007).

Lastly, I would like to thank my family, close friends and coworkers, for their inspiration and support.

Antoine Kalmbach
Jyväskylä, Finland

List of Algorithms

List of Figures

List of Tables

Contents

1 Introduction

"Begin at the beginning," the King said, gravely, "and go on till you come to an end; then stop."

– Lewis Carroll, *Alice in Wonderland*

This thesis presents a data importation techniques for simplifying the optimisation of the vehicle routing problem (VRP). The vehicle routing problem is a NP-hard (Lenstra and Kan 1981) combinatorial optimisation problem. Much work has been done for developing algorithms for its optimisation; this thesis does not concern itself with that particular research problem. In this thesis, I make the assumption of the existence of an environment, in which a *solver* or a *vehicle routing system*, a computer software that can process vehicle routing problems and generate optimised results based on certain input, with the added capability of processing different variants of this problem.

What I attempt to solve in this thesis is, is fundamentally a problem about data. Generally, optimisation solvers work much like a pipeline: problem descriptions go in, optimised results come out. (Puranen 2011; Drexl 2012) While this sounds like a gross simplification, the crux of it is that problems need to be modelled with data, bits of information that describe the problem itself. This data is then analysed and the solver churns this data into a result.

This process of data analysis always begins with understanding the format in which the data is represented. This data is usually in a structured format, the simplest of which being a tabular text file or a spreadsheet. This format is known *a priori* to the problem describer and the solver—the data can be relied to be in a certain format. As a result, strict requirements are imposed: the data must adhere to a certain logical structure for the solver software to work.

What if one were to provide a solver which could process data in a wide variety of formats, to induce a certain *softness* into the requirements? Of course, a solver could be built in such a way it merely *understands* a wide range of formats. This shifts the question towards effort and quantity: in that scenario, the solver would be required to be able to understand a multitude of different file formats and structures, and the more generalised the solver becomes, and the larger its usage context (e.g. the number of users wanting to use it) becomes, complexity increases, rendering the task difficult and laborious to maintain and develop.

As a result, what if the requirements were loosened, so that a potential data provider would need to merely adhere to the requirements in terms of *content*, but not structure? That is, what if the actual structure, e.g., the order in which data is presented, of the data would be regarded as secondary, as long as the data is there in a locally consistent format? If one were to develop techniques to automatically *infer* such structures, the data could be processed, and then analysed without having to check whether the data conforms with the desired input format or not.

The described scenario here is a classical data exchange problem. While our scenario is highly specialised into a certain domain, importing VRP descriptions into a solver, the theoretical underpinnings of this problem have been widely studied in literature. However, the highly specialised nature also means that little research has been done in scenarios like this. To my knowledge, data exchange has not been researched or used in the context of optimisation software.

To this end, my aim is not to construct an original data exchange technique, but to study the generic application of data integration and exchange techniques in a new context, the optimization of vehicle routing problems, and combining data exchange technologies to facilitate problem data processing. The study from this perspective has not been done before, and it is in this particular novel context that I present my contributions:

1. A *self-contained* formal characterisation of the *join inference* problem, using referential integrity concepts and relational algebra. I also provide a implementation for its computation, and to the classification problem of it I develop new features and improve existing ones.

2. A formal definition of the *attribute classification* problem in a domain-specific context using abstract formulations as mappings. Like above, I provide machine learning methods and entirely new features for classifying the attributes.
3. Unification of the *attribute classification* problem and *join inference* into **fleet inference**, and presenting it as a machine learning problem.
4. A new formal description of a *case model*, a mathematical construct of the VRP fleets.

We now start with the *background* associated with this thesis, where I explain my motivations for developing the above contributions.

1.1 Background

This section provides an informal introduction to our research setting. Our aim is to articulate the goals and ends we are after. In other words, this section presents arguments why we are interested in problems described later on, to justify the effort and work spent in this thesis.

Subsection 1.1.1 describes a new concept of *Optimisation as a Service* and we highlight one key aspect thereof that we think is a problem worth solving, which is further elaborated upon in Subsection 1.1.2. Later sections then describe the *context* in which we operate, to properly characterise the different parts that establish our research problem.

1.1.1 Scenario: Vehicle Routing Optimisation as a Service

The notion of offering optimisation as a service further clarifies our purpose. Offering optimisation services means roughly that an entity is offering a service of optimising VRP instances for any user—in a consultant–consultee relationship. The consultant, the software provider, provides a *service* for the user (the consultee), which uses the service from an endpoint. This endpoint can be any interface, a web browser, an application. The implementation of the protocol with which the service provider and user communicate is a technicality. Another key characteristic is that traditionally the service provider *hosts* the service on its own computer platform, e.g., a cloud platform, the user needs only the client software to use the service.

This scenario or paradigm is uncommon in the VRP industry. Traditionally, VRP software are provided as costly and complex standalone applications, meaning, that the user has to hostthe software himself and pay a license fee for the software. The optimisation-as-service model provides a stark contrast: the user has only to acquire a license to use the service, but there is no need for installing any complex software—which, given the complex nature of VRP, also requires computing power—only the client software is necessary.

The term *software-as-a-service* is a loose definition; the *anything-as-a-service* paradigm is a broad term that characterises an entity offering something in a service-oriented manner. Much research has spawned in the past few decades over concepts such as *software as a service*, *platform as a service* or even *infrastructure as a service* (for all definitions see Mell and Grance (2009)). A general review of the *x-as-a-service* can be obtained by referring to the works in any of the reference given above.

Thus, our *scenario* implies that we are offering the optimisation of VRP as a service. The optimisation back-end (call it a *solver*) takes in VRP instances as an input and produces results as an output. The format or the form of the input need not be defined strictly here—we are satisfied with the simple notion of the whole process acting like a pipeline. The pipeline concept can be seen as a metaphor for things going in on one end of the pipe and the results coming out from the other end. One might agree that such exuberance in metaphors can be detrimental to our purposes. However, this generalisation provides, in our view, an elegant simplification of the problem itself. As we will see later on, our language will eschew such informal nomenclature and move on to favour rigorous formalisms.

The data for a VRP instance is usually stored in a textual format. This format is then interpreted by the solver program using whatever information parsing technique available to it. Curiously, VRP is usually studied and experimented with in such a manner the structure of the input data is usually a casual afterthought. The different algorithms for optimising VRP are tested on standardised benchmarks, examples of which are available on online. The formatting of the benchmarks is given with their associated documentation. As a result, the whole field of VRP research has agreed on one global benchmark format. What if we are to offer optimisation as a service, wherein no predetermined formats exist, or if they did, imposing them on a service user would be counterproductive. Could we forgo such formats completely for the sake of usability?

The arising question is that we must have some kind of a definition of the structure of the problem data for the solver to be able to function, as elaborated in the introduction. Would it not serve our purposes better if the format definitions were defined with a certain degree of *softness* to them? This is indeed a pertinent question, and we need to consider the structuring of input data a bit further.

1.1.2 A Flexible Source File Format Is Necessary

"The essence of soft computing is that unlike the traditional, hard computing, soft computing is aimed at an accomodation with the pervasive imprecision of the human mind."

– Lofti A. Zadeh, *Neuro-Fuzzy and Soft Computing*

The need for a flexible file format, as argumented in this subsection, is glaringly obvious when inspecting the service from a user's perspective. If the service is not as strict when it comes to file formats, or *softer*, the user needs to invest less time in preparing his data so that it would conform to the format requirements. In a way, this is optimisation of optimisation, or *meta-optimisation*.

Descending further down the ladder into the problem domain, the input data, when supplied in a VRP service scenario (not to be confused with an optimisation-as-a-service scenario, a software architecture paradigm), judging by past experience in our research group[1] is usually structured in a tabular format, e.g, Microsoft Excel® or

1. Computational Logistics Group at University of Jyväskylä

comma-separated-values (CSV).

The tables contain data separated—visually, in the case of Excel® by a grid—or by commas, as in CSV. No relationships between each table exist, and these are inferred by hand, using key rules of thumb or format agreements. For example, there might be a table A which contains a list of vehicles accross rows, each row identified by an unique identifier, that describes the details of each vehicle available in the problem instance. Another table B might describe the places where we need to travel within the instance, and another table C might describe additional speed profiles for a specific vehicle type. The vehicles in A may *reference* the identified rows in the table C to indicate that, for any given vehicle in A, the vehicle has a specific speed profile detailed in table C.

Furthermore, each table contain some information about each vehicle, task, location, and so on. These are sometimes identified by column headers to mean some kind of an attribute for each entity it describes—as an example, the *maximum speed* column in C might say that, the vehicle type *Lorry* can travel at a maximum speed of 80 km/h. This vehicle type is then attached an unique identifier, say the number 1, that the vehicles in A can reference. Each vehicle in A that has a profile identifier 1 is then associated semantically of being a *Lorry* type of vehicle.

The next part is deciding in which context and purpose each attribute resides in. A maximum speed is usually given as a kilometre-per-hour unit, so this column is easy to understand. By looking at the data the service provider manually determines the specific context of it. As an example, to get speed limitations for each vehicle type, we need only to look for columns the values of which are specificed in a kilometre-per-hour unit. This process of classification is then repeated for each attribute until no attribute remains classified (or are discarded as redundant), and the data is then fit into a proper problem instance description in the language of the solver.

In summary, what the service provider has to do is to (i) manually infer relationships accross tables and (ii) identify the context an purpose of each column in the table. This two-step process is sometimes a very daunting task; indeed, could the service provider benefit from automating parts, if not everything, of this process?

The answer is a resounding *yes*. By simplifying this process—understanding the source format—even if the source format is already strictly agreed upon, we speed up the optimisation-as-a-service process by a great deal. We now on move to a more accurate description of the two-step process described in the above paragraphs.

1.2 Research Problem

This section is a slightly more concise rewording of the above section, and also presents the research problem itself. Suppose we have a VRP instance that consists of several different flat, tabular-oriented files, e.g., several Excel®sheets or tables. Our objective is to find a set of vehicles and a set of orders called *tasks* from this data. To do this, we must find some way to *connect* the sheets. For example, if the first table in the set of sheets contains a list of locations that ought to be visited and the addresses thereof; the second table contains descriptions of each location and the amount of cargo to be delivered or picked up from each location, and then the third table contains a list of vehicles ready for use, a capacity for each vehicle, and so on. In a problem like this, it is obvious that there is a connection between the first and second table.

This is called a reference, or a *referential relation*. Each *tuple* in the first table contains a value that points to an identical value in the second table, which indicates a link between the two rows. This is generally called a referential *constraint* since the values of the first table effectively restrict the values of the second table to take a particular form—in this case, to match those of the list of values in the first table.

After the referential constraints have been calculated, or to put in another way, all the tables have been linked to two distinct tables—one table for vehicles, one for tasks—we must properly classify each *column* in each table. For example, we must determine that a `capacity` column indicates a vehicle capacity for a vehicle in the fleet, or that `address` is mapped as an *address* for each task in the fleet.

Thus our research problem is more succinctly: " Given a set of data tables representing a VRP, using some method, partition these tables into two subsets, such that the data of one subset pertains to vehicles, and the other to tasks. Then use this information to parse the data set into a VRP solver. "

1.3 Structure of this thesis

This thesis is structured as follows. This chapter provided an introduction to the general topic and presented an informal problem formulation in the previous section. The next chapter, Chapter 2, gives a formal introduction to the theory surrounding the topic. Chapter 3 presents our actual contributions as the fleet inference problem and methods for solving it; Chapter 4 benchmarks our results and tests its performance and accuracy. We analyse the implementation and its usefulness in Section 4.4; Chapter 5 concludes the thesis with reflection and opens topics for future exploration.

"Journalists say that when a dog bites a man, that is not news, but when a man bites a dog, that is news. Thanks to the mathematics of combinatorics, we will never run out of news."

– Steven Pinker

"Outside of a dog, a book is a man's best friend. Inside of a dog, it's too dark to read."

– Groucho Marx

2 A Theoretical Background

"Theory is the first term in the Taylor series expansion of practice."

– Thomas M. Cover (1997)

This chapter presents a formal introduction to the theory surrounding the presented problem. The chapter begins with a short introduction to the Vehicle Routing Problem (VRP) and its relevance to our work in Section 2.1. The basic formulation is presented and some relevant variations of the problem are also explained; methods for solving it on the other hand are left out. Section 2.2 puts the research problem into a theoretical context, and explores fields that are closely related to it. We move on to defining the *join inference* in Section 2.3 and *attribute classification* in Section 2.4 problems and review methods for their solving, through studying relational algebra, we give formal representations of these as well. The chapter concludes with a general introduction to machine learning in Sections 2.5 and 2.6.

Compared to the previous chapter, this chapter is heavy on formalisms. I consciously selected a strong formal approach to problem definition, as it enforces a certain quality of correctness. To understand the notation no deep mathematical understanding is necessary; basic set theory principles and formal logic is sufficient, Section 2.1 deals with some basic graph theory concepts. Some sections deal with relational algebra and database theory, but these chapters are mostly self-contained and the material is kept concise. Section 2.5 delves more into probability and statistics as machine learning is strongly connected to both. As such, a brief understanding of probability theory and basic linear algebra will help, but is by no means mandatory.

2.1 The Vehicle Routing Problem

The Vehicle Routing Problem (VRP) is a combinatorial optimisation problem initially presented by Dantzig and Ramser (1959) as an extension of the Travelling

Salesman Problem (TSP). TSP is a problem where the objective is to find, given a set of points and the distances between them, starting from one point called the *starting point*, find the shortest route such that each point is visited exactly once and the route returns to the starting point. The name *traveling salesman* then signifies a travelling salesman that visits each city (i.e. the points) only once and seeks to calculate the shortest way of doing so to minimise travel costs.

VRP is a generalisation of the TSP in that there are *multiple* salesmen[1]. The salesmen are called *vehicles* and the central starting point is called a *depot*, from which each vehicle departs. The typical objective is to find a shortest set of routes for each vehicle such that each order is visited exactly once and all vehicles return to the depot. This can be described more formally as follows. A VRP *instance* is a complete directed graph $G = (V, E)$ where $V = \{0, 1, \cdots, n\}$ is the set of vertices to be visited and E is the set of edges. The vertex 0 is the depot. A positive *travel cost* c_{ij} is given to each edge $(i, j) \in E$ where $i \neq j$. The edge costs are symmetrical: $c_{ij} = c_{ji}$. The cost matrix c satisfies a triangle inequality: $c_{ik} + c_{kj} \geq c_{ij}$ for all $i, j, k \in V$, so the direct link i — j is the shortest route between each vertex. The problem objective is to find a minimum number of k Hamiltonian cycles, the cost or weight of each cycle defined as the sum of each edge. Each vertex is visited only once in each cycle and the cycles start and end at the depot.

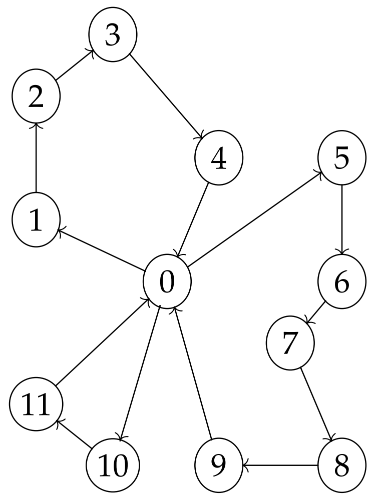


Fig. 1: A VRP instance.

Keen readers might have noticed straight away that this problem is too abstract to be applied in the real world. Typical real world scenarios include a much richer set of constraints, for example, the fleet of vehicles might be heterogeneous, each vehicle type can carry only a certain type of cargo, the cargo capacity of each vehicle

1. One could also say that TSP is a special case of VRP in that there is only one vehicle and no depot.

might be limited, each vertex (or point) has a *demand* to be picked up or delivered. Moreover, each vertex might have a certain *time window* wherein it can be visited ... and so on. The most basic of these *models* are the capacitated VRP (CVRP) and VRP with Time Windows (VRPTW). We present the formulations for each of these.

CVRP issues a *demand* d_i for each vertex $v_i \in V \setminus \{0\}$. The order might be a cargo delivery or picking up, for the purposes of our formulation this difference is not relevant. Let Γ be a finite fleet of n vehicles and $\gamma_i \in \Gamma$ denote a vehicle in this fleet and each vehicle has a capacity $Q_i \geq 0$. A route can be given for each vehicle as a sequence of vertex indices, that is $r_i = (v_0, \cdots, v_n)$, where v_0 and v_n represent the depot, satisfies the capacity constraint if $\sum_{j=1}^{n-1} d_{(v_j)} \leq Q_i$ for each vehicle $\gamma_i \in \Gamma$.

VRPTW extends CVRP by adding the concept of travel times, time windows, and service times. Each vertex must be visited within these service times; this is constrained by the travel time for each edge. Each edge $e \in E$ has a positive travel time $\tau_e \geq 0$ and each vertex $v_i \in V \setminus \{0\}$ has a service time σ_i, that is, the duration which the visit to vertex v_i will last. Lastly, the visit has to occur within a time window $[a_i, b_i]$.

Usually, from a model perspective, we will use the term *task* over vertex. We will use this shorthand from now on since it nicely characterises the notion of orders, places and requirements.

These formal requirements are but the most basic ones. The above two variants can be extended, modified and generalised into an infinite mix of different problem models; a comprehensive list of these can be found in Toth and Vigo (2002). For the purpose of this thesis we will limit ourselves to deal with VRPTW, and as a result, CVRP.

In general, the complexity of solving VRP and its variants is a computationally complex task, in fact, it is a NP-hard problem (Lenstra and Kan 1981). A multitude of methods have been developing for solving it, these range from exact algorithmic methods, heuristics and metaheuristics. We omit the discussion of VRP solving completely and instead refer the reader to the book by Toth and Vigo (2002) for a complete study of the field.

Typically, these algorithms are implemented in such a manner that the input source of the problem is in a predetermined format, to be used in somewhat standardised benchmarks. This is particularly useful for measuring the efficacy of an algorithm since it allows for an easy comparison between algorithm implementations as the input is guaranteed to be in one format. However, for real world applications, the usefulness of such a format is abysmal. Not only are the benchmark instances nothing like real world data, their complexity is far inferior to the richness of real world VRP instances. Usually, benchmark data is synthetic and mimics some sort of a geometrical structure, as in the case of the symmetrical benchmarks first examined in Christofides, Mingozzi, and Toth (1979).

Recall from previous sections that based on our empirical studies in client–provider interactions, VRP instances are usually delivered to the *service provider*, i.e., problem solver, in a tabular format. Thus any approach used to handle benchmark data so far in scientific literature is rendered useless for us. We must develop our own, which we will call *fleet inference*.

2.2 Discovering Fleet Data

This section presents a formal introduction to the fleet inference problem. We start by describing the case model and move on to describing methods for its discovery, first starting with the introduction of data exchange and integration. We then move on to a related field of schema matching and then finally conclude why the tools developed in neither data exchange nor schema matching are fit for our purposes.

To begin with, we must first define what a fleet actually is. Generally, a *fleet of vehicles* in a VRP instance is a set of vehicles V each to be assigned to a set of *tasks* T. The goal of the optimisation process is to find which task is best assigned to which vehicle. Furthermore, each vehicle V can have different constraints, e.g., capacities as in the capacitated VRP. The same applies to tasks as well, in that each task can have a specific *time window* associated with it, as in the VRP with time windows.

Additional formalisations are given for the case model later in Chapter 3, where we present a formal case model and problem formulations for the whole discovery process.

The goal of the fleet discovery is to take a source of data, e.g., tabular CSV filer, and produce a cohesive case model. The process is two-fold. First, we find the referential constraints between each flat file in order to produce one singular relation. Second, now that our data resides within one relation in a cohesive format, the data in this relation is then extracted using attribute classification, and fitted to our case model.

Loosely speaking, we are interested in finding the vehicles and tasks from any source set of data. This section presents a formal introduction to each problem and also develops some key theory associated with each. The first part consists of identifying in which research domain our fleet discovery problem resides. We begin by describing the related fields of data exchange and schema matching.

2.2.1 Models And Data Exchange

Our data discovery problem in this case is fundamentally a kind of a *data exchange* problem. In data exchange, the aim is to take data from different sources and assimilate it into a *target* that provides an accurate representation of the source data. (Kolaitis 2005; Fagin et al. 2005) In our context, the sources are different VRP optimisation problem data sources and the target is the cohesive *case model* we seek to fill.

Data exchange differs from *data integration* in that data exchange deals with materialised data transformations, from sources into target instances, whereas in data integration the target schema is only *virtual*. A data integration system provides interfaces for *querying* the source instances through an abstract global view. In data exchange the target instance tries to represent the source instances as accurately as possible. (Lenzerini 2002; Calì et al. 2006)

Schemas are a familiar term in database environments. We use the term in a broader sense to refer to not only database schemas, but abstract models and ontologies as well, as defined by Bellahsene (2011). Generally speaking, a *schema* is an abstract list of constraints in some formal language for any structured data. We give the following definition:

Definition 1. ***Schema****. A schema $\mathcal{S}$ is a tuple $\mathcal{S} = (\Sigma, \Delta)$ where Σ is an alphabet of predicate symbols in some logic $\mathcal{L}$ over Δ and Δ is a set of* integrity constraints expressed in Σ.*. Each predicate symbol $s \in \Sigma$ has a fixed* arity*, the number of arguments associated with it.*

A schema can be used to express integrity constraints between different relations (or tables) in a database $\mathcal{DB}$. We say that a relational database $\mathcal{DB}$ is constrained under a schema $\mathcal{S}$ if all its database relations $\mathbf{R}$ satisfy the constraints in $\mathcal{S}$.

The notion of a schema provides another characteristic for the context in which our problem resides. In data exchange problems we try to assimilate data from mostly *heterogeneous* sources, in that the sources are usually in different formats and varying schemas. Our fleet discovery usually operates within the following constraints:

- the data usually resides in flat, tabular formats (e.g. comma-separated-values, CSV), in different files,
- no integrity constraints between these files exist, that is, we do not know how they relate to each other, and
- once the integrity constraints have been created, can we start pairing attributes.

Given that there are no source schema to work from, this separates us very clearly from the worlds of data exchange and data integration. As we will later find out, our method is somewhat orthogonal to the classical data exchange approach. However, our problem is related to the schema matching and mapping problem, and we will first introduce the meanings behind those two terms.

2.2.2 Schema Matching And Mapping

The related fields of schema matching and mapping are important fields of their own, and they have been intensely studied. It should be noted that these terms are often used interchangeably, and we feel like there is a need to point out the semantic differences between the two terms *mapping* and *matching*.

Schema matching can be seen as the act of reconciling two different relational schemas together. The foundational goal is to find correspondences between attributes for purposes of, e.g., exchanging data between two databases. (Acar and Motro 2009). A multitude of methods have been developed for schema matching, Bernstein, Madhavan, and Rahm (2011) provide an excellent and recent review of the state-of-the-art.

Schema mapping is the act of using the information gained in the schema matching process to map elements between two schemas. The distinction is clear: in schema matching, we are interested in finding the *correspondences*, that is, to answer the question, whether two schema elements resemble each other; in schema mapping, we are interested in using the resemblance information to provide the transformation between these schema elements. In other words, a schema matching operator *Match* would identify semantic correspondences and a mapping operator *Map* would use these correspondences to map one schema element to another.

As an example, consider a situation where a source schema contains a price in one currency (e.g. dollars) and the source schema has one in another, e.g. pounds. A matching operator will identify that these items *correspond* to each other. A mapping operator will specify the exchange multiplication (from dollars to pounds) in the actual *transformation*. A generic survey on evaluating both methods can be found in Bellahsene (2011).

Caveat lector. We made the above distinction because it highlights the difference between the two steps that are usually contained within either definition. That is, sometimes schema matching is used to refer to both matching and mapping, and conversely mapping is used to refer to both matching and mapping. Bernstein, Madhavan, and Rahm (2011) and Rahm and Bernstein (2001) use schema matching to produce mapping. In a later, more abstract *model matching* framework, Bernstein (2003) describes mappings in that contexts as a way to "[...] describe how two models are related to each other" (Bernstein 2003), where *model* is are formal descriptions of different artifacts, e.g., a relational schema.

Conversely, data exchange settings usually define schema mappings as an integral part of the process. This is exemplified by Kolaitis (2005) and Fagin et al. (2005). These conflicts in terminology might appear slightly confusing. To remedy this, we reconcile the differences between by always referring to the whole process as **schema mapping**. In our goal of fleet discovery we are interested in mappings as a mathematical concept, not as plain data correspondences, as schema matching would imply.

In general, these two terms work on a metadata level, and we work on the data level. This is further corroborated by Rahm and Bernstein (2001): "[schema matching] operates on metadata (schema elements) and joins on data (table rows)". This means that our problem of fleet inference is not contained within these fields, as we target the actual *data contents* in our schema mapping inference. According to Bernstein, Madhavan, and Rahm (2011), such schema matching is considered "extensional", because it tries to extend the very bare-bones schema into a coherent one. We can conclude that our problem *intersects* with these fields.

Clearly, the join inference problem does not reside within either of these two fields. What is more, the attribute classification problem is a more specific instance of schema matching because in schema matching we usually have both the target and source schema as variables. In our scenario the target schema is a fixed domain model. This does not mean the tools in schema matching are of no use to us—on the contrary, the toolsets of schema matching will be of great use to us, as will be seen in Section 2.4.

As a conclusion, the processes of schema mapping or matching are strongly related to our field. We differ in the sense that our scenario has no source schema to work with, it must be created. Thus level at which we operate is intuitively at the *data level*. Schema mapping works on the *metadata level*. Only later on in our process, in attribute classification, does the problem become a form of schema mapping. The other bit that separates us from traditional schema mapping is that usually we work with only a single relation with very domain specific data. Our process begins with the discovery of schema-like constraints, such as foreign keys, which we call *join inference*.

2.3 Join Inference

This section introduces the concept of join inference. We start by defining some key characteristics of dependencies Subsection 2.3.1, starting from very fundamental relational axioms, after which we proceed to *joins* in Subsection 2.3.2 and finally move on to a formal characterisation of the problem as *join inference*. The focus of this section is to formalise join inference.

Common terminology. In the following subsections we talk about relations, tuples and attributes. These terms are often seen in contexts involving relational databases, but we use them here in a more general sense. Building upon the definitions by (Silberschatz 2006), we characterise the terms as follows:

- A **relation** (or a *table*) R within a database $\mathcal{DB}$ is a collection of unique n-tuples $(\mu_1[X_i], \ldots, \mu_m[X_n])$ where $m \geq 0$ and $n \leq |A|$. We say $\mu[X]$ where X is some subset of the attributes A of R, i.e., the list of values in the for each attribute in A. The cardinality of A is defined as $|A|$ and is equal to the number of attributes in the relation, this is also called the *arity* of the relation.
- A **domain** is the set of permitted values for each attribute, e.g., a numerical attribute might be restricted to an integer domain. A mathematically inclined person would say that relations are Cartesian products between domains $D_1 \times D_2 \times \cdots \times D_n$ (Silberschatz 2006).
- In traditional relational database texts *tuple* is used interchangeably with **row** and *relation* is with **table**. In this thesis, we shall use the former definitions as they provide a higher level of abstraction.

To properly define what functional dependencies, primary keys, and foreign key are, we must first start with some basic formal definitions of dependencies, starting from *functional dependencies*.

2.3.1 Constraints and Dependencies

This subsection presents a basic characterisation of relational dependencies. The first one are functional dependencies (FD), which form the fundamental concept of database keys. Another dependency class, inclusion dependencies (IND), are used to characterise foreign key constraints between two relations. To properly distinguish the two, we start by defining them formally, starting from basic relational axioms.

Functional dependencies state that for a functional dependency to occur between two attributes A and B in a relation, for each tuple for the attribute A there must be only one tuple for the attribute B. These dependencies are useful for assigning unique identifiers for tuples in a relation, because it lets us state, e.g., that given a FD with two attributes `id` $\rightarrow$ `person`, it states that for each `id` there exists only *one* `person`. This property is useful in determining key constraints. The concept of FD can be stated more formally as follows, adapted from Aho, Beeri, and Ullman (1979) and Casanova, Fagin, and Papadimitriou (1984):

Definition 2. ***Functional dependencies****. A functional dependency $A \rightarrow B$ for a relation R, where for an attribute set $A \subseteq R$ and an attribute $B \in R$, signifies that for all tuples of the attribute set A there exists only values of attribute B. The functional dependency is satisfied if for all tuple pairs $\mu, \nu \in R$ that are equal on attribute A: $\mu[A] = \nu[A]$, are also equal on attribute B: $\mu[B] = \nu[B]$. We say that the tuples μ and ν* ***agree*** *on A and B.*

Given a FD $A \rightarrow B$, we say that B is *functionally dependent* on A.

As we spoke about key constraints earlier in Section 2.2, using functional dependencies, we can now characterise those definitions even further, building upon the definitions given by Huhtala et al. (1999) and Bernstein (1976). An attribute set X of relation R is a *superkey* if and only if *no* tuples in R agree on X (see Definition 2), as a result, a superkey can be used to uniquely identify tuples.

The attribute set X is said to be a *key* of R, expressed as $\mathcal{K}(R) = X$ if it is a superkey and no subset of X are superkeys. More formally, for every attribute $A \in R$, if X is a key, it holds that

$$A \notin X \Leftrightarrow X \to A.$$

A *minimal superkey* is the minimal set of attributes, i.e., the superkey with the least number of attributes, that can be used for identification—dropping any attribute would destroy the functional dependency. A minimal superkey is also called a *candidate key*. Lastly, only one of these candidate keys can be the *primary key* of a relation.

Finally, this lets us characterise the difference between *primary keys* and a *foreign keys*. A *foreign key* in another relation references the *primary key* in another. A *foreign key constraint* (FKC from now on) is a statement

$$R[A] \subseteq S[B]$$

where A is a set of attributes of R and B is the set of attributes of S, where B is the primary key of S. A FKC is satisfied if each tuple μ_1 in R there exists a tuple ν_1 in S such that $\mu_1[A] = \nu_1[B]$. (Calì et al. 2006)

From this characterisation it follows that foreign keys are *semantic relationships*. This is because, unlike primary keys, they cannot be reduced to functional dependencies, and among other reasons, can occur by pure chance. Attributes with coinciding tuple values (for each tuple) does not constitute a primary–foreign key relation in itself. Hence, foreign keys are fundamentally a form of *inclusion dependency*.

Inclusion dependencies. Inclusion dependencies (IND) are another kind of relational dependency. Loosely stated, inclusion dependencies indicate that there is a connection between two different database relations. An inclusion dependency might be that every manager residing in one relation might also reside in the employee relation, logically, every manager is also an employee. More formally, given two relations R and S, each tuple in R might also be in the relation S. (Casanova, Fagin, and Papadimitriou 1984)

Definition 3. ***Inclusion dependencies***. *An inclusion dependency (IND) between two relations R and S, when, $R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n]$, i.e., all the tuples in the relation $R[A_1, \ldots, A_n]$ are also present in $S[B_1, \ldots, B_N]$, where A_i and B_i are attribute names.*

It is now apparent that FKC are actually a form of inclusion dependencies! However, the terms come from different directions. Foreign keys are semantic references to primary keys, which themselves originate from FD. Figure 2 demonstrates this abstraction chain. Each arrow $\longrightarrow$ indicates a higher degree of generalisation, and a wobbly line $\rightsquigarrow$ indicates a semantic relation.

$$\begin{tikzcd}
\boxed{\text{Foreign Key}} \arrow[r, squiggly] & \boxed{\text{Primary Key}} \arrow[r] & \boxed{\text{Candidate Key}} \arrow[r] & \boxed{\text{Superkey}} \arrow[r] & \boxed{\text{FD}}
\end{tikzcd}$$

Fig. 2: The abstraction chain from primary keys to functional dependencies.

Furthermore, whenever there exists a FKC there also exists a IND, but not necessarily vice versa. The question becomes, is an inclusion dependency enough proof to characterise a proper primary–foreign key relation? In other words, does the mere existence of an IND imply that the values are semantically related?

The semantic problem of FKC becomes relevant in its definition. Recall that a FKC between two relations R and S states that all tuples present in R, which contains the foreign key, are also present in S, which contains the primary key. As a result, thus the tuples in R are dependent on the tuples of S. The similarity between an IND is apparent, but it is worth noting that a FKC is a *semantic* relation. (Rostin et al. 2009) point that an IND may occur by pure chance due to the subset precondition from Definition 3, as a result, the presence of an IND is not a valid reason to classify the relation as FKC. They further state that "foreign keys are semantic relationships and cannot be inferred with certainty from an instance of schema alone."

To conclude, we give the final final characterisations for functional and inclusion dependencies:

- **Functional dependencies** occur within one single relation, and can be strictly defined as attribute dependencies that preserve uniqueness. They can be further constrained into their most relevant form of *primary keys*.
- **Inclusion dependencies** are inter-relational references, which simply state that an attribute set of a relation may be contained within another. At face value,

this does not provide enough information to confirm whether it is a **foreign key constraint**, because the containment (subset precondition) may occur by pure *chance*. As a result, finding inclusion dependencies becomes a much more complex problem—semantic relations can be hard to infer without careful scrutiny.

Given the problem of semantic inference, we now move to the methods used in finding the semantic relationships automatically, by first defining the source of the problem: relational joins between tables.

2.3.2 Computing Joins

In this subsection we review some basic relational algebra operations, in order to define the concept of a relational *join* operation $\bowtie$. To begin with, we define the following relational operations, building on the definitions given in Silberschatz (2006, Sections 2.2 and 2.3):

The **select** operator σ_p *selects* tuples from a relation according to a given predicate p. Given a relation T_1 in Figure 4, the following select operation

$$\sigma_{Type=Lorry}(T_1) \tag{2.1}$$

would return all the tuples in relation T_1 that satisfy the predicate condition $Type = Lorry$.

The **project** operator Π_s *projects* all the tuples from a relation with the attributes listed in the expression s, with the attributes *not* in s left out. This is useful if we want to select only some of the attributes from a relation, for example, in T_2, to select only the $RegNumber$ attribute from T_2 (Figure 4) we would write

$$\Pi_{RegNumber}(T_2)$$

to get all tuples in T_2 with only the attribute $RegNumber$.

We also define three additional operators, the Cartesian product $R \times S$ and the union operation $R \cup S$, and the intersection operation $R \cap S$:

VEH_ID	TYPE
2	Lorry
3	Car
4	Truck
5	Tractor
6	Lorry

(a) T_1

REGNUMBER	ID
XYZ-123	2
SSY-313	3
BCE-132	4
QQA-312	5
AEA-141	6

(b) T_2

Fig. 3: Two relations T_1 and T_2

- The Cartesian product $R \times S$ produces tuples that are concatenated such that for each tuple $c \in C$, where $C = R \times S$, there is a tuple μ_i in R for which $c[R] = \mu_i[R]$ and $\nu_i \in S$ for which $c[S] = \nu_i[S]$. If R contains n_1 tuples and S contains n_2 tuples, then C will contain $n_1 \times n_2$ tuples.
- The *union* operation $R \cup S$ produces tuples that appear in either or both relation, as in set theory, but with the following restrictions: (i) the relations must have the same *arity*, i.e., the same number of attributes and (ii) their attributes domain must match . Duplicate tuples (rows) are removed.
- The *set intersection* operation $R \cap S$ which produces all the tuples with the same attributes.

Basic binary operators are allowed in the predicate expression p: basic comparison operators $=, \neq, <, \leq, >, \geq$ and common first-order logic operators used to combine predicates into larger ones, AND $\wedge$, OR $\vee$ and *not* $\neg$. Using the operators σ and Π, we can now define the *natural join* operation.

The methods described in the previous subsection gives us means for creating *join plans*, unifications of different relations into a single relation via a *join* operation. A (natural) join operation between two relations R_1 and S using two attributes A, B is expressed as $R_1 \bowtie_{A=B} S$. This means that all tuples relations R_1 and S are projected into a single relation using the attribute equivalence as a key constraint, such that a tuple should be created whenever the equivalence $A = B$ holds. In Figure 4 the join operation is done using the attribute equivalence ID $=$ VEH_ID. In relational algebra, the join can be expressed more formally as follows. Let A be an attribute of R and B be an attribute of S. Their natural join is the projection

$$R \bowtie S = \Pi_{R \cup S}(\sigma_{R.A_1 = S.A_1 \wedge \cdots \wedge R.A_n = S.A_n} R \times S) \tag{2.2}$$

where $R \cup S = \{A_1, \ldots, A_n\}$, this can be read as *project* all attributes that exist in both relations ($\Pi_{R \cup S}$) from the relation given by *selecting* all the tuples with equal attributes from the Cartesian product $R \times S$.

The *equi-join* operation is a shorthand for a join operation $R \bowtie_{A=B} S$ where attributes or attribute sets A and B are equal. Thus the join operation referenced from now on will always be an equi-join unless otherwise stated.

ID	TYPE	REGNUMBER
2	Lorry	XYZ-123
3	Car	SSY-313
4	Truck	BCE-132
5	Tractor	QQA-312
6	Lorry	AEA-141

Fig. 4: The join $T_1 \bowtie_{\text{ID}=\text{VEH\_ID}} T_2$. The choice of ID and VEH_ID as attributes is obvious: all the tuples for both attributes are equal.

Joins can be chained together, e.g., $R_1 \bowtie_{A=B} R_2 \bowtie_{B=C} R_3$ is a join of three relations using the attributes A, B and C.

A note on conventions. Because the attribute names in relations might sometimes be very lengthy to write, for a shorthand we simply use attribute numbers when expressing joins, such that a join operation $\bowtie_{i=j}$ means to join two relations with an equivalence of the ith and jth attributes of their respective relations, starting from the attributes $A_1 \ldots A_n$, ordered and read from left-to-right in graphical representations. For example, the join operation in Figure 4 becomes $\bowtie_{1=2}$, because VEH_ID in Figure 3a is the first attribute in that relation and ID is the second attribute in Figure 3b from left-to-right. Thus in the future with a relation R, $R[1]$ will simply mean the set of tuples for the first attribute, and so on. If the relations R and S have no attributes in common, $R \cap S = \emptyset$, then the join operation is simply their Cartesian product $R \times S$; or if $R = S$, their join is the set union $R \cup S$.

The optimal method for joining these relations is to use joins based on foreign keys (Acar and Motro 2009; Rostin et al. 2009; Lopes, Petit, and Toumani 2002). We can formulate the problem of *join inference* as defining it as an inference problem over a *join dependency*. A join dependency for a relation R means that R can be recreated by joining other relations together, and that the join is *lossless*, i.e., any relation that satisfies the join dependency can be *decomposed* into distinct relations and then be joined back together without any loss of information.

Definition 4. ***Join dependency***. *(Silberschatz 2006; Deutsch 2009) A join dependency for a schema $R(U)$ is an expression*

$$R : \bowtie [X_1, X_2, \ldots, X_n] \tag{2.3}$$

where $1 \leq i \leq n$, $X_i \subseteq U$, such that the lossless-join decomposition $\bigcup_{i=1}^{n} X_i = U$ holds. An instance r of schema $R(U)$ satisfies a join dependency *if*

$$r = \Pi_{X_1}(r) \bowtie \Pi_{X_2}(r) \bowtie \cdots \bowtie \Pi_{X_n}(r) = \underset{i=1}{\overset{n}{\bowtie}} \Pi_{X_i}(r). \tag{2.4}$$

Here, $\Pi_{X_i}(r)$ is the projection of r on the attributes of X_i. The dependency is trivial if any R_i is **R**.

Example 1. *(Adapted from Deutsch 2009) Let there be two relations $V : \{truck, driver\}$ and $T : \{truck, cargo\}$. In these relations, driver and cargo are not correlated together for obvious reasons. To produce a relation that entails a list of tasks for each truck called tasks, we require that the projection of tasks on V and T produces the tasks relation:*

$$tasks : \bowtie [\{truck, driver\}, \{truck, cargo\}].$$

Definition 4 efficiently gives us a precise restriction for our problem of join inference. Our task is to find some set of constraints such that produces the best possible join such that the join dependency between its relations is satisfied. This process is called *join inference*. (Acar and Motro 2009; Hristidis and Papakonstantinou 2002)

Definition 5. ***Join inference***. *Let* **R** *be a relation such that there exists a set of n relations* $R_1, \ldots, R_n$ *that* approximate *the lossless-join decomposition of* **R**. *The task of* ***join inference*** *is to find a conjunctive set of constraints c such that*

$$\sigma_c(R_1 \times R_2 \times \cdots \times R_n)$$

is maximally *equivalent to* **R**. *The conjunctive terms in c are equality constraints between two attributes of differing relations* R_i.

Maximally in Definition 5 means that the constructed join plan yields a representation of the data that is the closest match to it, e.g., the maximum number of retained attributes per the decomposition. Thus, the problem is now how to infer the conjunctive set of foreign key constraints c, constraints that are good enough to provide the maximal solution. Finding such constraints can be tricky. As stated before, an inclusion dependency can happen by pure chance. Since this chance is non-trivial, we must develop robust and efficient methods for finding proper FKC to our end.

2.3.3 Finding Foreign Keys

The basic method for inferring these foreign keys is by first looking for IND and further promoting them into FKC using automated reasoning. The name of this procedure has a multitude of different names in literature. Rostin et al. (2009) talk of finding FKC but Acar and Motro (2009) talk of finding *join plans* and *join inference*. While these two terms are seemingly distinct, they fall under the general category of *dependency inference*, in which the ultimate goal is to find foreign keys for creating a join plan. In this thesis when we shall from now on refer to *foreign key inference* when talking about computing joins using inferred foreign keys.

Historically, the study of join plans dates back to the universal relation model (Maier and Ullman 1983), where database schemas are traversed automatically through join dependencies. In other words, the universal relation creates a transparent layer through which a database can be explored. A problem arises when join dependencies are *cyclical*, when two schemas reference each other in a circular fashion. (Acar and Motro 2009) The original universal relation model was improved on by Maier, Ullman, and Vardi (1984), introducing the notion of maximal objects. Maximal objects are partitions of the graph-based representation of join plans, partitioning it into acyclic subgraphs—eliminating the chance of cyclical references from occurring.

Otherwise, not much of the existing work in schema matching, mapping and data integration has dealt with finding FKC. Much more work can be found in the field of finding IND. This can be partly attributed to the fact that, as stated above, the inference of FKC is not as simple as of inferring IND. On the other hand, FD, the most famous of database integrity constraints (De Marchi, Lopes, and Petit 2009), have been widely studied in literature. Contrary to IND, their automatic discovery (see Kantola et al. 1992; King and Legendre 2003)) has seen greater interest (De Marchi, Lopes, and Petit 2002; Rostin et al. 2009). De Marchi, Lopes, and Petit (2002) argue that this due to two things: (i) the discovery of IND is difficult due to its complexity (Casanova, Fagin, and Papadimitriou 1984) and (ii) they lack popularity, as functional dependencies have traditionally been studied a lot more.

To this end, we found a handful of approaches in existing literature. The one closest to our application is using automatic classification, as suggested by the work done in Rostin et al. (2009), who suggest using classification methods for inferring FKC. For classifying attributes as foreign keys, Rostin et al. (2009) considered a handful of features, built by "careful observations and common sense". According to Zhang et al. (2010), the most important of them for a foreign key candidate are:

1. **Cardinality**. A foreign key should usually consist of mostly distinct values and contain a significant number of them.
2. **Coverage**. Because a foreign key is an IND, it should cover most of the primary key it refers to, and they should belong to the same domain.
3. **Uniqueness**. Foreign keys should not be primary keys to other relations, i.e., they should not have foreign keys themselves.
4. **Unary dependence**. A foreign key should reference only one primary key at a time.
5. **Length difference**. The difference between the average length of values for

foreign–primary attribute pairs should more or less be of the same length.

6. **Completeness**. The foreign keys should contain a minimal number of values not in the primary key.
7. **Name similarity**. The names of the primary and foreign keys should be similar.

In their classification methods Rostin et al. (2009) found that features 2, 7, 5 and 6 to be consistently the most discriminative features. Their method for implementing feature 7 was just by checking for exact matches or containment of two attributes. They state that using more advanced measures for this particular feature would be an "obvious improvement". As we will see later on in Chapter 3, we improved this method by using string distances.

Note that some of these characteristics in the above list have been used in the schema matching problem (see Subsection 2.2.2) for example in Kang and Naughton (2003).

For non-machine learning approaches Acar and Motro (2009) suggest a method that constructs a connected graph of all possible join relations, called a *join graph* and then transform this graph into a *maximum weight spanning tree*, where edge weights are governed by their suitability as join plans. Figure 5 shows that the number of edges in the join graph is equal to the number of attributes in the Cartesian product $R \times S$.

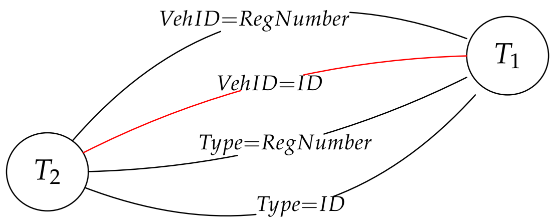


Fig. 5: Join graph for the relations T_1 and T_2. The number of edges is equivalent to the number of attributes in the Cartesian product $T_1 \times T_2$.

Acar and Motro (2009) also use the fact that foreign keys, as referential attributes towards other primary keys, usually have functional dependencies within their own relation. This fact can be used to further identify foreign keys. The first item in the list on page 26 is an example of such a feature.

Other methods include using SQL statements for finding FKC, e.g., Lopes, Petit, and Toumani (2002). For finding plain INDs, Bauckmann, Leser, and Naumann (2006) use similar SQL statement based methods. A lot of previous work based on automatic reasoning for schema matching has been done in Bernstein, Madhavan, and Rahm (2011) and Doan, Domingos, and Halevy (2001), but as stated in Subsection 2.2.2, in our context the schema is decidedly unknown.

Furthermore, our context is entirely data-driven—there is no schema to work from. As a result, methods commonly used in schema matching are of little use for us (Rostin et al. 2009). Bauckmann, Leser, and Naumann (2006) underline this succinctly by stating that "instance-driven [...] approaches directly analyse the data of a given database instance" and point out that work is shifting towards instance-driven approaches, instead of the classical schema-based approach. Acar and Motro (2009) agree and also point out that there is a shift towards semi-schemaless data where, e.g., one primary key is known but no foreign keys are. Examples of such methods are BHUNT (Brown and Hass 2003) and CORDS (Ilyas et al. 2004). The similarity between our work and the latter two works has been noted.

Zhang et al. (2010) consider the problem of foreign key discovery in a multi-column context. Recall that foreign keys are defined as attribute *sets*, and that multiple attributes can act as a foreign key. Such a key is usually called a *composite key*. For the discovery, Zhang et al. (2010) describe an algorithmic method called *Randomness*. They compared it to the method proposed by Rostin et al. (2009), with the result of *Randomness* having an overall F-measure of 1 versus the F-measure of 0.95 for the method of Rostin et al. (2009), making their approach slightly more reliable. Since it was not a machine learning method, no training was needed. Like Rostin et al. (2009).

To summarise, we have seen several attractive approaches for FK inference in schemaless or semi-schemaless data. This thesis will focus on using machine learning methods for finding these foreign keys, as exemplified by the work of Rostin et al. (2009). We think that the presented machine learning method will be quite robust and adaptable, and by extending it with more features, such as functional dependencies, we will improve upon that framework even further.

In general, machine learning methods have the ability to exploit existing data, and unlike most of the approaches that have been mentioned above, machine learning methods do not have to rely on fixed methods—these can be inferred on an instance basis. (Rostin et al. 2009) This allows us to build flexible systems that after some refinement can infer many semantic relationships which might otherwise be a very onerous task, which may or may not include developing a sophisticated algorithm for FKC detection.

In summary, join inference is a rather complex problem. This section served as an introductory definition thereof, and we also introduced the reader to the functional and inclusion dependencies, and formally defined foreign key constraints. The terms were introduced in a mostly self-contained manner using basic relational algebra. Now that we have methods for constructing joins using join inference, let us move onto detailing the process that takes place afterwards—attribute classification.

2.4 Classifying Attributes

"The most important property of a program is whether it accomplishes the intention of its user."

– C. A. R. Hoare

This section presents the final *pièce de résistance* of our fleet inference problem: attribute classification. We introduce some key concepts of attributes and attribute domains, then move on to describe the *case model* in the context of routing, and finally explore some methods for *finding* the correct data domains for each attribute—which, in learning terms, is called *classification*.

Attribute classification can be loosely stated as the finding of attribute correspondences between two *data domains*. Generally a data domain is a specific space of values that is distinct from other data domains in the data domain set $\mathcal{D}$. For example, an attribute `speed` might be represented as an integer data type, and allowed to have a range of 0 to infinity, disallowing negative speeds.

Associating data types with each attribute is a common pattern in schema creation (Doan, Domingos, and Halevy 2001; Naumann et al. 2002). A data type allows the discrimination of attributes based on their *values*. In a reverse context, when the attribute domain is not known, the values can be of great help to determine in which domain they belong. If dealing in a domain-specific context, e.g., VRP, we can attach some key features to a certain data type: for example, we know to expect certain constraints for individual vehicles such as its *speed* or *capacity*. The second bit is that we also know that both of those constraints are usually expressed as natural numbers.

Attribute domain inference thus becomes the second part of our fleet inference problem. Our goal is, given a finite set of attributes $A_1, A_2, \ldots, A_n$, to find the target correspondence from the attributes of the domain schema of a case $\mathcal{C}$. We now move on to formally describing what this case model actually is.

2.4.1 The Case Model

A *fleet* of vehicles is generally an set of vehicles. The fleet is assigned a set of orders, to be divided among the vehicles. The semantic combination of both can be referred to as a *case* from the sense that each problem scenario is typically unique and could typically be dealt on a *case-by-case* basis. Using the models described in Section 2.1, we can list some key features of each.

Vehicle

- a registration number, or another unique identifier
- a maximum capacity (CVRP)

Task

- a *demand* for each task
- a **location** for the task
- a **time window** (VRPTW)

Recall from Section 2.1 that these features are in accordance with the VRP with Time Windows (VRPTW), which incorporates a *time window* for each task, on top of the constraints for the capacitated VRP. Using these basic constraints, we can characterise a fleet $\mathcal{F}$ to be a tuple $\mathcal{F} = (V, T)$, where V is an unordered set of vehicles and T is an unordered set of tasks. Of the fleet we define the following:

- Each *vehicle* $v_i \in V$ is *unique*, thus a vehicle v is a tuple $v = (i, \Delta)$, where v_k indicates that it is the vehicle in V with its identifier $i \equiv k$; Δ is a set of unique *constraints* δ, e.g., capacity δ_c, for each vehicle.
- Each *task* $t_i \in T$ is a quintuple $t = (l, d, s_i, w_a, w_b)$, where l is some location in some data type, e.g., Cartesian co-ordinate on a map or a string denoting an address; $d \geq 0$ is a real-valued *demand* of that task; the *time window* (w_a, w_b) is the time window in which the task must be completed. w_a is the start of the time window and w_b is the end; usually described in a time stamp format or some absolute numeric measure of time, and it holds that $w_a \prec w_b$, where $\prec$ indicates that w_a *precedes* w_b. Finally, a service time s_i associated with $t_i \in T$, it must hold that $|w_b - w_a| \geq s_i$

We could extend this model a lot further for much more complicated VRP instances, using additional characterisations. Generally, in the field of operations research, VRP models are described in their linear programming standard form, but these describe the optimization problem itself, not the domain model. Examples of much richer VRP models can be found in Toth and Vigo (2002). Generic *case domain models* like the one above are scarce in literature; but an example formulation using Z notation can be found in Puranen (2011).

Thus the goal in attribute classification is to look at the attributes in a source relation, first formed by join inference, then decide which one of these attributes correspond to a target attribute in the case model. We begin the definition of this process by concentrating on attribute domains.

Table 1: Data Domains Of Attributes In A VRPTW Case Model

Variable	Domain Attribute	Description	Primary Domain	Secondary Domain	Entity
i	Id	A unique identifier	string	$\mathbb{N}$	Vehicle
δ_c	Capacity	Maximum vehicle capacity	$\mathbb{N}$		Vehicle
l	Location	Task location	string	$\mathbb{R}^2$	Task
d	Demand	Task cargo demand	$\mathbb{R}$		Task
w_a	WindowStart	Task Time Window Start	string	$\mathbb{N}$	Task
w_b	WindowEnd	Task Time Window End	string	$\mathbb{N}$	Task

2.4.2 Attribute Domains

We say that an attribute belongs to a certain *data domain* if its values are restricted to some type of data. What this actually means requires additional clarifications. Generally, we can say that a *data domain* is any set of values allowed to a certain attribute. As with `age`, we can say that an attribute `gender` can belong to the binary domain of *male* or *female*, or that a continuous attribute such as `temperature` may belong to the domain of $\mathbb{R}$.

Common data types encountered in any sort of computerised data are *integers* $\mathbb{Z}$, floating point numbers, strings of characters, or boolean values $\{0,1\}$. These basic types serve as data domain *data types*. In general, values can be continuous or discrete, they can also be categorical in the sense that their allowed values belongs to a certain discrete set of values.

To generalise even further, we can use the terms developed by Sekhavat and Parsons (2012), that themselves originate from the Bunge–Wand–Weber ontology in Wand and Weber (1990). We can say that an instance i of a domain is any existence of a particular domain, e.g., 60 might be an instance of the domain $\mathbb{N}$. In the context of attributes, we say that an instance is a *manifestation* of a domain using the following definitions, adapting the definitions given in the former reference:

Definition 6. ***Manifestation****. Let D be the domain of values for any property p. A manifestation of p is a value $v \in D$ of p assigned to that property p, denoted by a tuple $\mu := \langle p, v \rangle$.*

Intuitively, we see the connection between properties and attributes. We can say that a tuple in a relation R is a set of manifestations of the attributes $A \subseteq R$. Example 2 demonstrates this using a simple numerical attribute.

Example 2. *Let a domain D_1 be a subset of integers $D_1 \subseteq \mathbb{Z}$ and A_1 an attribute with this domain named* `capacity`*. A tuple $\mu_i = (\texttt{capacity}, 5000)$ is* manifestation *of this attribute A.*

For example, any attribute `age` (in the context of a *Person* relation) is restricted to be in the domain of natural numbers $\mathbb{N}$. The concept of a data domain gives us one tool: we can say that attributes are always bound to a certain domain, because it would be illogical to have tuples of the same attribute with differing data domains. This allows us to put some constraints or restrictions on the values of an attribute A.

This realisation also provides us a useful way of inferring whether two attributes are equal. Intuitively, two attributes A and B share a domain D if *all* of their manifestations belong to the same data domain.

2.4.3 Schema Matching Revisited

Using the concepts developed in the previous subsection, we can now further characterise the problem of finding pairings between two attributes A and B. Obviously, if attributes have incompatible domains, the mapping does not make sense. In general, we are interested given a set of attributes $\mathbf{A}$ and $\mathbf{B}$ we are interested in finding a mapping $\mathbf{A} \rightarrow \mathbf{B}$ that maps each attribute $A \in \mathbf{A}$ to a corresponding attribute $B \in \mathbf{B}$. A *similarity* between A and B is indicated by $A \cong B$. The similarity score $sim(A, B)$ is an arbitrary number that measures the *goodness* of the similarity, it is usually a value between $[0, 1]$.

Example 3. *Let a target schema $\mathcal{T}$ be the domain schema that defines the following attributes:*

$$\{VehicleType,\ Capacity\}.$$

Consider a source schema $\mathcal{S}$ with the following attributes: $\{Type,\ Volume,\ Manufacturer\}$.

The goal is to find a mapping c that correctly maps $Type \mapsto VehicleType$ and $Volume \mapsto Capacity$.

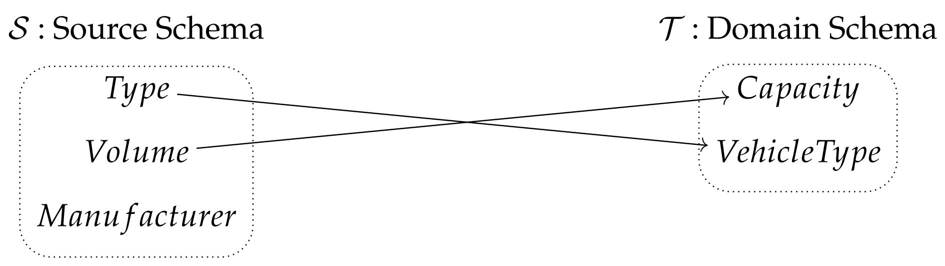


Fig. 6: A visualisation of the sample mapping sought in the above example.

In our case domain model, the target schema is a schema $\mathcal{D}$. The schema matching process can be now further specialised into the following definition:

Definition 7. ***Domain Schema Matching****. Let $\mathcal{D}$ be a* domain schema *with a defined set of m attributes D. Given a set of n attributes of a source instance S, find a* matching operator *that is an injective mapping $M : S \to D \times D$. The resulting* match set

$$\{(S_1, D_1), (S_2, D_2), \ldots, (S_n, D_m)\}$$

consists of attribute pairs $S_i \in S$ and $D_j \in D$. The domains of attributes must be equal: $dom(S_i) \equiv dom(D_j)$. Finally, for each pair (S_i, D_j) it holds that

$$S_i \cong D_j \Leftrightarrow \underset{i,j}{\arg\max}\ sim(S_i, D_j).$$

We could have also specified it as a mapping of $f : \mathbb{N} \to \mathbb{N}$, where $f(i) = j$ where i is the ith attribute of S and j is the jth attribute in D. The codomain $S \times D$ is used in Definition 7 for expository purposes.

Given this definition, the aim in domain schema matching is to find a mapping that using some similarity metric finds a pairing between two attributes with the highest similarity score. This mapping problem is later defined as a classification problem in Section 2.5.

Example 4. *The match set of Figure 6 can defined as*

$$\{(Type,\ VehicleType),\ (Volume,\ Capacity)\}$$

using attribute names instead of numerical indices in match set, or,

$$\begin{aligned} Type &\cong VehicleType \\ Volume &\cong Capacity \end{aligned}$$

using similarities. The attribute $Manufacturer$ is left out because no mapping was found.

2.4.4 Similarity Features

In the previous subsection, we presented the problem of finding a mapping scheme that can map source attributes to specific target attributes. In this subsection we seek to answer the question of what kind of similarity *features* can be used to devise such a mapping. It quickly turns out that a lot of different features exist, and the same applies to the number of mapping generation algorithms, i.e., schema matching methods.

Learning algorithms employ some kind of a learning process (defined later in Section 2.5) to produce accurate *prediction*. This requires *prior knowledge* in the form of teaching data. Non-algorithmic versions use prior knowledge embedded into the inference model—they use pre-calculated inference rules to establish heuristics or exact methods for determining whether some property holds. We will begin this subsection with a review of works that use learning methods.

The problem of finding a proper mapping is characterised as being *automatic*: we are interested in algorithms that produce the best kind of matches automatically. In the past decade, a lot of work has been done in the world of automatic schema matching and mapping. Good surveys of the field as a whole include Rahm and Bernstein (2001) and Bernstein, Madhavan, and Rahm (2011). We refer to those studies for a thorough review of the different methods used in schema matching; here we focus on the different features used to calculate similarities. However, Rahm and Bernstein use the term *matching criteria* but we will prefer the more abstract term *feature*.

Features. In general, a *feature* is a trait or property that describes the target. This concept is put into a proper context later on in the next section, but for now, we shall contend with this definition. Since often the choice of features is more important than the choice of classifier (Hastie, Tibshirani, and Friedman 2001), this section will focus on the actual features used, instead of the classification methods, which are only mentioned by name.

Calculating a similarity measure between two attributes A and B means that we need to identify an inherent *likeness* between the attributes. Consider two attributes whose domain are strings of characters. The general idea is that if the values or this attribute, of these attributes are more or less similar, we can probably derive a kind of likeness measure between them; or if two attributes share the same name—or the names resemble each other to a certain degree—we can also say that these attributes may mean the same thing.

Note that the implied similarity may be a *false positive* in that the attributes may *appear* to be similar, yet they are not. Numerical features that share the same domain may contain the same numbers, yet their semantic meaning is entirely different; or attribute names may be similar by coincidence (Kang and Naughton 2003).

The choice of correct features is *a fortiori* an important step in the whole process of constructing a classification method for schema matching; even more so due to the fact that usually a feature set determines the efficacy of a classifier. As such, we look on to examples in the field to see what kind of features have been considered. We return to the problem of choosing actual classifier features in Subsection 3.3.2.

Some example features. Probably the earliest systematic evaluation of features to be used in automatic schema matching is Li and Clifton (1993). They listed the following features useful: (1) synonym dictionaries for comparing similarities between attribute names, (2) schema metadata, e.g., key constraints and data type specifications and (3) the data values themselves . They used the features listed in item (2) to develop different probability coefficients for measuring similarities.

The work by Li and Clifton eventually led to the development of the SemInt tool (Li and Clifton 1994, 2000). SemInt is an automatic schema matching tool built using neural networks. The methods presented in the first paper are further developed and contrary to the original work, the algorithm looks at the data values for features. For example, string patterns such as the present of a whitespace between two words can indicate a person's name; or for numerical fields, similar statistical measures—variance, mean and coefficient of variation—can indicate a strong similarity.

Another machine learning approached called LSD, or *Learning Source Descriptions*, (Doan, Domingos, and Halevy 2001) combines a number of learning techniques to infer matches. The system is trained by combining several methods to form a kind of an *ensemble* method. They considered *word frequencies* of domain-specific words in instance (e.g. the word *great* in description fields within real estate databases), *value distributions* and *integrity constraints* (e.g. foreign keys).

Comparing value distributions is particularly useful. In our domain, we could say that a *vehicle capacity* is logically a higher number than the individual *cargo* of a task, or that the speed of a vehicle is a much smaller number than its capacity—yet, their data types may all be similar. As a result, by using domain-specific data and known data distributions, we can derive some key hypotheses about what kind of features each attributes have.

A note on categorical attributes. Attributes with no meaningful distance metric within their values are hard to map together (Andritsos et al. 2004). This is the case with categorical values; a string attribute such as *color* with values such as *red* or *green* has no adequate distance measure, thus computing a similarity measure between them is hard. To this end, Andritsos et al. (2004) present a clustering algorithm that use *information entropy* to cluster attribute values into groups. This clustering information can be then used as a similarity measure between attributes. Similar measures using entropy for attribute compatibility can be found in Acar and Motro (2009, 2010, 2008). Berlin and Motro (2002) use the concept of *information entropy* for feature selection, in a machine learning framework. In that framework, a Naïve Bayes classifier is used to look for information gain when pairing attributes.

In general, the strongest coefficient between attribute similarity is data domain compatibility. The next similarity coefficient is the similarity of the data itself. We say that a domain compatibility is the first *similarity metric* S_0 between two attributes, S_1 are value *pattern similarities* and then the final one S_2 is the value similarity. Pattern similarity roughly means that the attributes tend to look like each other—in strings this can be interpreted as having a language that can, to some degree of fuzziness, recognise both strings, or for numerical values, a similar normal distribution.

In summary, this section presented the latter part of our fleet inference, attribute classification. Attribute classification can be generalised into *schema matching*, because in both we are interested in finding attribute correspondences. Our scenario fixes a certain target schema $\mathcal{D}$ to which we want to map attributes, and the methods for doing this involve using some kind of similarity or correspondence measures. We broached the topic of features in this subsection and further elaborated what kind of features have generally seen as been useful. The contexts in which these features are used are in the *learning* and *classification* phase. Features form a key aspect of *machine learning*, which is the topic of the next section.

2.5 A Primer On Machine Learning

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

– Tom M. Mitchell, *Machine Learning*

This section introduces the reader to the domain of machine learning. This is by no means a small task; thousands of technical papers have been written on the subject for over fifty years. The goal of this section is to provide a brief introduction to machine learning in a general environment, and in particular, to the abstract concept of *learning*. This is to elaborate on the techniques and terms mentioned in the previous sections.

The introduction is intentionally kept short and concise. The field of machine learning is vast, its roots go deep and intersect at the peculiar boundary of statistics, prob-

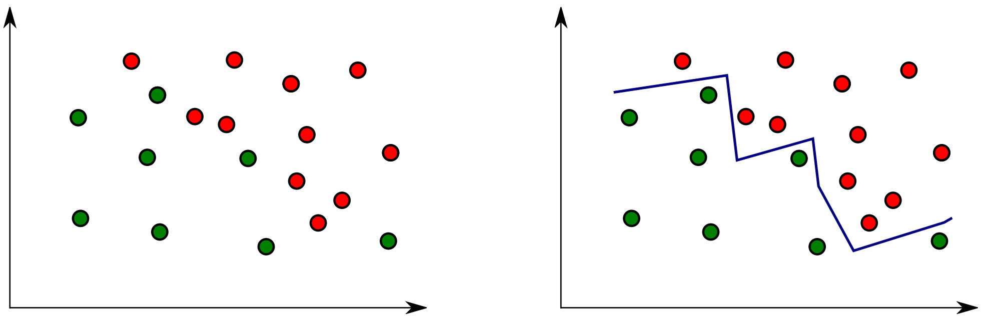


Fig. 7: A machine learning problem. (a) Illustrates the set of points we wish to classify into two categories. (b) A line that splits the set of points into two based on their color.

ability theory, computer science and software engineering. For the presented introduction to be sufficiently informative, only one particular part of machine learning—classification—is covered, due to its relevance to the rest of this thesis. For further reading, we provide the reader a list of material, both practical and theoretic, in Subsection 2.5.5.

The principal sources of this section are Mohri, Rostamizadeh, and Talwalkar (2012), Bishop (2006), Mitchell (1997) and Hastie, Tibshirani, and Friedman (2001) and Schapire and Freund (2012).

2.5.1 Introduction

The goal of machine learning is to use computational methods to give accurate predictions using experience gained by training. This could be, for example, face recognition, optical character recognition, stock market prediction, spam detection, computer-aided diagnosis, control theory, and so on. The list of different applications of machine learning is vast. There are a multitude of sub-problems within machine learning, some of the most notable of which are:

- **Regression** is using training data to be able to predict the value of a function, such as the stock market index. In regression, accuracy can be easily measured as the difference between the correct value of the predicted function and the predicted value.

- **Classification** is using training data to classify a set of items into distinct categories. Optical character recognition is an example of this, where images are assigned target characters in some alphabet, or spam recognition, where the goal is to predict whether an email is spam or not.
- **Clustering** is the task of dividing a set of values into distinct groups. An example goal is to find some connection between data values.

Other important purposes of machine learning are *ranking*, where the goal is to find some ranking for a set of data, e.g., movie recommendations and *dimensionality reduction* where a large set of features are reduced into more meaningful ones while preserving accuracy. In this thesis, we will mostly focus on classification, since we are interested in mapping a set of instances to certain categories.

Terminology. When discussing machine learning we will come accross various terms related exclusively to different kind of entities associated with learning. These are:

- **Features**: features of an attribute, e.g., in detecting an inclusion dependency, this could be any of the features described in the list Subsection 2.3.3, or in the case of detecting spam, the presence of certain words typical to spam.
- **Instance**: any unit of data to be either *validated* or *tested*, i.e., classified in some sense.
- **Label**: a flag indicating to which category a training instance belongs to, e.g., in the case of spam, this could be a Boolean value, 1 for spam, 0 for ham.
- **Training sample**: a labeled *training set* consists of labeled instances given to the learning algorithm, to produce a set of classification rule set to form a *classifier* or a *hypothesis*.
- **Validation sample**: labeled instances that are used to tune the free parameters of the learning algorithm to prevent *overfitting* whereby the algorithm constructs a model that is too complex and generalizes poorly.
- **Test sample**: test instances on which the learning algorithm is tested on — this data is unlabeled and is not available in the learning process. Intuitively,

this is generally the data which the user will want to see to know whether the learning algorithm works or not.

A learning algorithm might make errors when tested on the training set, this is called a *training error*. The fraction of the correct answers given on the test set is called the *accuracy* of the algorithm; the number of *errors* on this set is called the *test error*. We assume that the samples are drown using some random distribution D, and we denote the probability of *misclassification* of this set as the *generalisation error*. The concepts of learning can be generalised to a more abstract model, in these association it is common to talk about the *loss function* and *hypothesis set*:

- **Hypothesis set**: a set of functions that map features to their labels $\mathcal{Y}$. $\mathcal{Y}$ could be $\{0,1\}$, for a binary classification problem, or a real-valued score $s \geq 0$ for spam, the higher a score, the more likely it is spam. A *hypothesis* is often synonymous with the term *classifier*, see below.
- **Loss function**: measures the goodness of the hypothesis. It measures the distance of a predicted label from the correct label. More formally, let $\mathcal{Y}$ denote the set of labels and $\mathcal{Y}'$ be the set of output labels, the loss function is defined as a mapping $L : \mathcal{Y} \times \mathcal{Y}' \to R_+$. Given a hypothesis h, if $y' = h(x)$, and the correct label $y \in \mathcal{Y}$, the loss $l = L(y,y')$. If l is zero, then the prediction is correct.

  Different methods exists for measuring loss (see below), usually $L(y,y') > 0$. For a binary classification problem, where $\mathcal{Y} = \{0,1\}$, the loss function is the indicator function $1_{y \neq y'}$, which is defined as follows. Let x be a sample and h the hypothesis, and $y' = h(x)$ the predicted label for x,

$$1_{y \neq y'} = \begin{cases} 1 & \text{if } y \neq y' \\ 0 & \text{if } y = y'. \end{cases} \tag{2.5}$$

- **Model selection**: tuning the parameters of the algorithm, that is, tuning the parameters of the selected hypothesis itself.

In general, the hypothesis set is the set of functions a learning algorithm must *form* to be able to learn. A canonical spam classification scenario is a good way to illustrate the process of a learning algorithm. First, we analyse a random subset of a set of emails for relevant features. These could be the length of the email, its grammatical accuracy, the presence of some red flags such as the word "viagra".

After the features have been selected, we select a random subset of emails, calculate feature vectors for them and associate labels to them—i.e., determine whether they are spam or not spam, by hand. This forms a set of *training examples*. The training examples are then used to *train* or *learn* the algorithm. Using these training examples, the learning algorithm, by fixing different values of its free parameters, will choose from a set of hypotheses the best hypothesis that fits the training data. Finally, the algorithm is tested on the *test set* (disjoint from the training set) by applying its hypothesis on those instances.

In general terms, the hypothesis function is the set of values for a learning algorithm's parameters; in classification contexts, these are the fixed parameters for a classification function. For example, a learning algorithm might set a threshold value of some kind for the number of grammatical errors, and after training it might fix some value δ to this parameter and whenever an instance's grammatical error number $\delta' > \delta$, by the hypothesis, this will either be classified to spam or ham.

We will use the words *classifier* and *hypothesis* interchangeably. Other synonyms for these are terms in literature are *classification ruleset*, *predictor* or *classification model*. In the previous list we left out an important sample set called the *validation sample*. If the training set and test sets are small, how can we know whether the learning algorithm can generalise to an independent data set? Usually, a learning algorithm will contain a number of free parameters, which are then used to fit a *model* based on the training data. The fitting process essentially chooses the best values for its parameters—the best hypothesis—that fits the training data. This is called *model selection*.

Learning scenarios. There are different scenarios for learning. For example, there is a learning scenario where *no* correct labels for features are provided, where the learning algorithm must infer the labels itself. This is called *unsupervised learning*.

A good categorisation between different learning scenarios can be found in Mohri, Rostamizadeh, and Talwalkar (2012), which we will summarise below.

- **Supervised learning.** The learning algorithm receives labeled training sample and makes predictions on unlabeled data.
- **Unsupervised learning.** The learning algorithm receives unlabeled training data and makes predictions on these. Examples of these are clustering and principal component analysis (PCA), a type of *dimensionality reduction*.
- **Semi-supervised learning.** The learning algorithm receives both labeled and unlabeled training data, and makes predictions on unseen data. This scenario is typical in situations where features are easy to calculate but obtaining the correct labels is expensive.
- **Transductive inference.** A kind of semi-supervised learning where a learning algorithm receives labeled training data and unlabeled test data. Unlike in semi-supervised learning, where the objective is to make predictions on *any* unseen data, in transductive inference the goal is to make predictions only for the given test data. This is used in, e.g., learning finite automata, where the idea is to learn a language represented a finite automaton.
- **On-line learning.** In on-line learning, there is no difference between the learning and training phase. The goal of on-line learning is to minimise a loss function over many iterations. For each iteration, the learning algorithm receives an unlabeled sample and predicts it. Then it is shown the *correct* label and calculates the loss between its own label and the correct one. The successive goal is to eventually minimise this loss over time.
- **Reinforcement learning.** The learning algorithm acts as an *agent* in an *environment* and seeks to maximise some kind of a reward. This could be, using an example from control theory, a robot that tries to maximise free-roaming so that it does not bump into walls. Over time, the robot will get better at not hitting walls. This is a key characteristic of reinforcement learning: the reward will get successively smaller and the agent must decide whether to perform new actions, *explore*, or stay in a familiar environment, *exploit*. This is known as the *exploration versus exploitation* dilemma.

- **Active learning.** Lastly, the active learning scenario is where the learning algorithm actively seeks out labeled data by querying the user for them. This tries minimise the manual labeling involved in a supervised scenario should the manual labeling prove to be an expensive operation. The learning algorithm selects the examples to be labeled.

In conclusion, learning scenarios are plentiful. The latter part of the above list either falls into the first three categories, or are more special cases of them. In this thesis, we will concern ourselves with supervised learning. This is obvious since we essentially know the features of a correct classification. Unsupervised learning would be useful in case we were not familiar with, e.g., the possible features of an inclusion dependency.

This subsection presented key terms of machine learning and we will now move on to a way of determining the *reliability* of a learning algorithm. We use the term reliability in a loose sense here; in general, we are interested in how well an algorithm can *generalise* or how *accurate* it is.

2.5.2 Cross-validation

If we select another independent set of the training samples *not* used in training, called the *validation sample*, and test the learning algorithm on this set, it usually turns out that the validation sample does not fit the model as well as the training sample. This phenomenon is called *overfitting*, and it happens usually when the learning algorithm is complex in its number of parameters or the training sample is small (Bishop 2006, p. 32). Informally, it states that the algorithm does well on data it has encountered, but does not do well on data it has not. To minimise the risk of overfitting, a technique called *cross-validation* has been developed.

Cross-validation is a statistical technique for analysis the generalisation capability of a statistical model. In this context, the model is the hypothesis, or classifier, and we are interested in how well the classifier can generalise to an independent data set. In machine learning contexts, we usually talk about *n-fold cross-validation* for

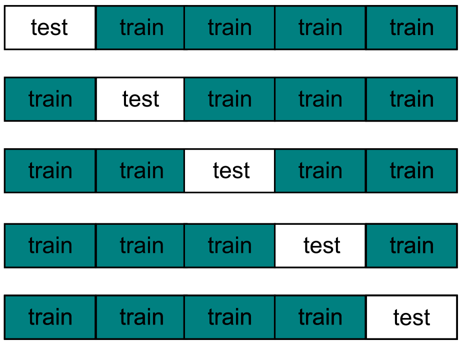


Fig. 8: Cross-validation where $m = 5$. The algorithm is trained on $m - 1$ fold and tested on the left out fold.

model selection. In cross-validation, we partition the training data into specific disjoint subsets: a part of the training data is used to train and another part is used to test. Using this process we can further tune the free parameters θ of an algorithm. This allows the assessment of the algorithm's performance to be done entirely on the test set; the training data can be fully used in the training phase. This can be defined more formally as follows, adapting from Mohri, Rostamizadeh, and Talwalkar (2012).

Let θ be the free parameter vector of a learning algorithm. For some fixed value of θ, partition the sample S of m labeled samples into n sub-samples, or *folds*. The ith fold is a labeled sample $((x_{i1}, y_{i1}), \ldots, (x_{im}, y_{im}))$ of size m_i. The learning algorithm is trained for any $i \in [1, n]$ in every sample but the ith fold to generate a hypothesis h_i. The performance of h_i is then tested on the ith left out fold. The parameter values θ are then evaluated using the average error of the hypotheses h_i which is called the *cross-validation error*, denoted by $\hat{R}_{CV}(\theta)$:

$$\hat{R}_{CV}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \underbrace{\frac{1}{m_i} \sum_{j=1}^{m_i} L(h_i(x_{ij}), y_{ij})}_{\text{error of } h_i \text{ on the } i\text{th fold}} \tag{2.6}$$

Choosing the number of folds n is an important question, and much work has been devoted to this in theoretical studies. The curious case of $m = n$, called *leave-one-out cross-validation* (LOOCV), exactly one instance is left out with every iteration of cross-validation. The disadvantage of cross-validation is that it can be costly to compute. Indeed, in LOOCV, the algorithm has to be trained n times on the sample

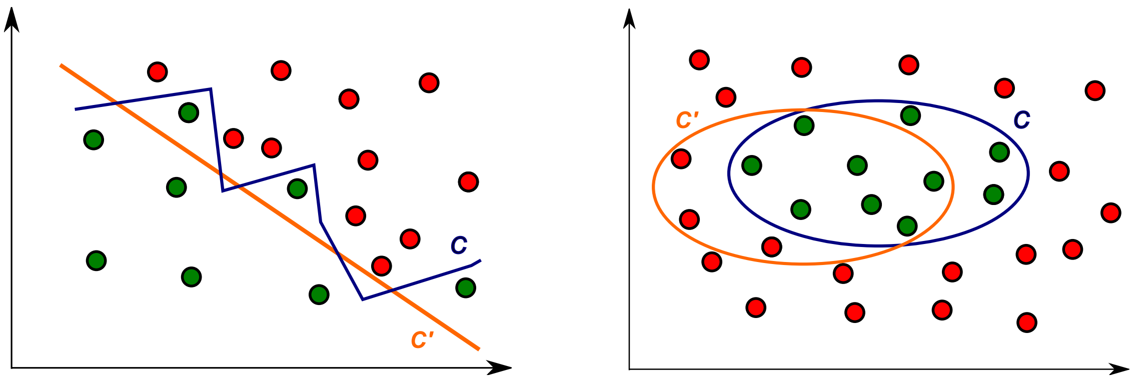


Fig. 9: Examples of different learnable target concepts C and learned hypotheses C' for two categories.

size of $m-1$, as evidenced by Figure 8.

2.5.3 Model Selection

The learning ability of an algorithm can be measured in two ways. The first, called the *generalisation error*, is a measure of how far the current hypothesis is of the actual concept to be learned. This measure tells us how good the algorithm is using the rules it inferred on independent data, and will tell us the expected error of the algorithm. The *empirical error* is the average error of the learning algorithm on some sample. The notable difference is that the generalisation error is not available to the learning algorithm, since it requires knowing the target concept to be learned; the empirical error is available as it simply requires a concept and a labeled sample to work with.

More formally, these error concepts can be defined using the notions of *concepts* and *hypotheses*. Adapting the formal definitions from Mohri, Rostamizadeh, and Talwalkar (2012), we define the learning problem as follows.

Let $\mathcal{X}$ be the *input* or *instance space* and $\mathcal{Y}$ the set of *target labels*. We will use the *binary classification* scenario of spam detection as an example here, so $\mathcal{Y} = \{0,1\}$, where 0 is ham and 1 is spam. A *concept* c is a mapping $c : \mathcal{X} \rightarrow \mathcal{Y}$, i.e., a mapping that maps inputs from $\mathcal{X}$ to a subset of labels in $\mathcal{Y}$. In general terms, this concept may be some

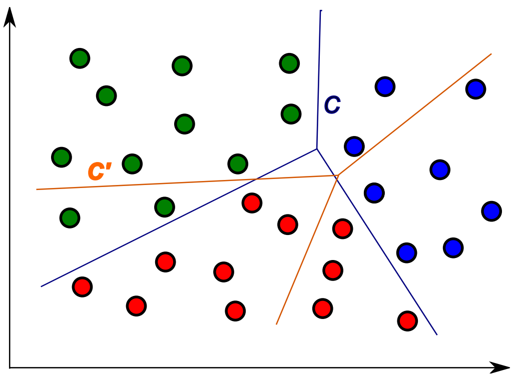


Fig. 10: Another example of target concepts and learned hypotheses, with three categories.

hyperplane in a geometric space that splits the input space into two distinct classes, or a rectangle within $\mathbb{R}^2$ that contains the points belonging to a certain concept. The objective is to learn a set of concepts C called the *concept class*, for example, all the possible hyperplanes or rectangles that separate the input space. Examples of such concepts are in Figure 9 and Figure 10.

The learning algorithm receives a set of concepts, C; and H, the set of hypotheses, which may or may not coincide with the concept class C. The learning algorithm is given a sample $S = (x_1, \dots, x_n)$ drawn independently and identically distributed variables (i.i.d.) using an unknown distribution D and the associated labels $(c(x_1), \dots, c(x_n))$. The task of the learner is to use the labeled sample S to select a hypothesis $h_S \in H$ with the smallest generalisation error to the concept c. The generalisation error is defined as follows:

Definition 8. ***Generalisation error.*** *Let $h \in H$ be a hypothesis and $c \in C$ some concept to be learned. Given a distribution D, the* generalisation error *of h is:*

$$R(h) = \Pr_{x \sim D}[h(x) \neq c(x)] = \mathop{\mathrm{E}}_{x \sim D}\left[\mathbf{1}_{h(x) \neq c(x)}\right] \tag{2.7}$$

where 1_ω is the indicator function of the event ω.

Another method of defining the generalisation error would be some absolute distance measure akin to the loss function, but the probabilistic approach allows an abstraction, which is useful in cases where the distance (e.g. mean squared error). Using the loss function $L : \mathcal{Y} \times \mathcal{Y}' \to \mathbb{R}_+$. Some common choices for the loss function

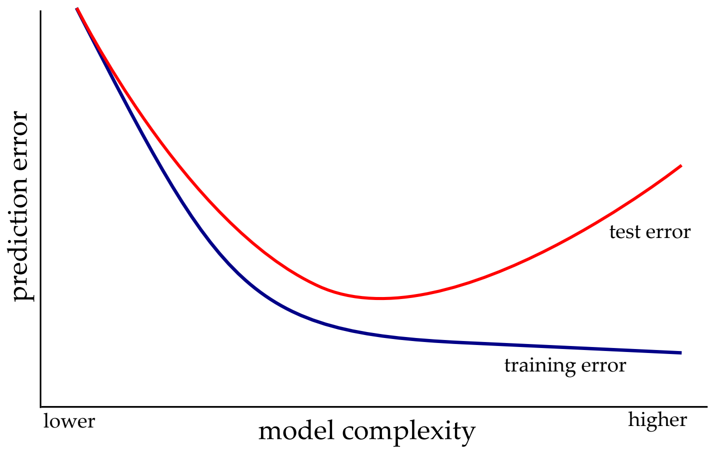


Fig. 11: A learning curve. As the hypothesis increases in complexity over successive training iterations, the *training error* becomes smaller.

are

$$L(y,y') = \begin{cases} (y-y')^2 & \text{squared error} \\ |y-y'| & \text{absolute error.} \end{cases} \tag{2.8}$$

The generalisation error would have then been defined as the expected value of the loss function, but using the indicator function $1_{y \neq y'}$ (see Equation 2.7) is better suited for binary classification.

A defining characteristic of generalisation error is that it is not known to the learning algorithm itself, as the concept $c(x)$ is not known to the learning algorithm. Intuitively, this is because the distribution of values D is unknown—the algorithm has no *a priori* information about the data is meant to generalise to! However, the algorithm can measure its empirical error based on *seen data*.

Definition 9. ***Empirical error.*** *Let $h \in H$ and $c \in C$ like in Definition 8 and a sample $S = (x_1, \ldots, x_n)$. The* empirical error *of h is defined as*

$$\hat{R}(h) = \frac{1}{m} \sum_{i=1}^{m} 1_{h(x_i) \neq c(x_i)} \tag{2.9}$$

The main difference between these two error measures is then that the empirical error measures the error over a sample S and the generalisation error measures the *expected error*. Indeed, if we fix some hypothesis $h \in H$, then the expected empirical error on a sample S is equal to the generalisation error:

$$\mathrm{E}\left[\hat{R}(h)\right] = R(h). \tag{2.10}$$

The PAC learning framework. The *Probably Approximately Correct* (PAC) learning framework was introduced by Valiant (1984). The PAC learning framework enables us to define learning as an abstract minimisation problem. The agnostic PAC model essentially sets the goal to select a hypothesis $h \in H$ with a low generalisation error. This is the minimisation problem. To clarify, the learning algorithm needs to find a hypothesis which with high *probability*, which will have a low generalisation error, that is, be *approximately correct*. A concept is *PAC learnable* when there exists an polynomial algorithm that fits the PAC model. Generic notions and very specific computational definitions, such as bounds for the hypothesis set H, are omitted from this thesis; we include the PAC learning framework for the sake of exposing the general idea of learning as a computational model.

The idea of learnability is connected to optimisation in the sense that a learning algorithm wants to minimise its loss L. Much advances towards discovering the intersection between optimisation and machine learning have been made in the past decade. This in general general known as *risk minimisation*.

These notions of error and risk will be further visited in Chapter 3, in section where we discuss the overall confidence measures of the learning algorithms. For further reading about model assessment we refer the reader to Mohri, Rostamizadeh, and Talwalkar (2012, Chapters 2 and 10) and Hastie, Tibshirani, and Friedman (2001, Chapter 8). For information about optimisation applied to machine learning see Sra, Agarwal, and Wright (2012).

2.5.4 Multi-class Classification

The target space $\mathcal{Y}$ of binary classification problems is intuitively representable as Boolean values. As the name implies, this domain is perfectly suitable for cases wherein categories are binary, such as spam recognition, or detecting inclusion dependencies. What about cases where there are multiple categories? Consider the attribute classification problem: for each source attribute, there are multiple possible target attributes in the domain schema. This makes the problem unrepresentable as a traditional TRUE–FALSE dichotomy.

In the context this thesis, the introduction of multi-classification is particularly relevant. This is because we can contrast it with binary classification, as it is used in foreign key inference. For attribute classification, we have multiple target categories, hence the problem is a multi-classification problem.

In practice, there are two distinct ways of categorising multi-class classification: the *mono-label* case $\mathcal{Y} = \{0,\ldots,k\}$ and the *multi-label* $\mathcal{Y} = \{0,1\}^k$. The mono-label case refers to cases where an instance can have a *single label* of the label set $\mathcal{Y}$ and in the multi-label case, an instance can be a part of several.

The classification scenarios found in this thesis are clearly binary classification, and mono-label, multi-class classification. For the former, attributes can either be or not be classified as foreign keys. For the latter, attributes can only be assigned to one single attribute at a time. When attributes have been assigned to a data domain attribute, no other attribute can be paired to this data domain attribute.

A multi-class classification problem is defined as follows. The learning algorithm is given a labeled sample $((x_1,y_1),\ldots,(x_m,y_m)) \in (\mathcal{X} \times \mathcal{Y})$ where $x_1 \ \ldots \ x_m$ is drawn i.i.d. from a distribution D; $y_i = f(x_i)$ for all $i \in [1,m]$ where $f : \mathcal{X} \to \mathcal{Y}$ is the classifier function. (Mohri, Rostamizadeh, and Talwalkar 2012, p. 183)

Using multiple binary classification tasks. A multi-class classification problem can be reduced efficiently to an aggregated set of binary classification problems. In the first method, called *one-versus-all* (OVA) or *one-versus-rest*, using binary notation, a sample belonging to the target label as 1 and 0 for the others. This requires then training k separate classifiers that try to label, for a label l, try to assign 1 for this class and 0 for all others. The hypothesis h_l can be derived from a scoring function $f_l : \mathcal{X} \to \mathbb{R}$, usually $h_l = \text{sgn}(f_l)$. The resulting multi-class hypothesis $h : \mathcal{X} \to \mathcal{Y}$ is then, for all $x \in \mathcal{X}$:

$$h(x) = \operatorname*{arg\,max}_{l \in \mathcal{Y}} f_l(x). \tag{2.11}$$

In *one-versus-one* (OVO), each class is pitted against another. The formed pairs are then used as the binary dichotomy. Each pair is used in creating a classifier that are trained *only* on the points labeled with the classes in that pair. This results into training $\binom{k}{2} = k(k-1)/2$ separate classifiers. The classifiers form an aggregated hypothesis, for each pair $(l, l') \in \mathcal{Y}$, and for all $x \in \mathcal{X}$:

$$h(x) = \operatorname*{arg\,max}_{l \in \mathcal{Y}} |l : h_{ll'}(x) = 1| . \tag{2.12}$$

This results in a binary tournament where the winning classifier will be eventually assigned to the label l.

The presented two techniques are the most prevalent aggregation techniques for multi-class classification. OVA traditionally exhibits a so called *calibration problem*: the scores given by the scoring function f_l are not always comparable. Intuitively, this might be because a scoring function cannot be universally calculated for each classification sample. The OVO technique solves this by not using a scoring function, instead comparing suitability on case by case basis. Finally, a concept called *Error-Correction Codes* (ECOC) has been developed to simplify the aggregation of multi-class classification into binary classification; for more information on this we refer the reader to Mohri, Rostamizadeh, and Talwalkar (2012, Subsection 8.4.3).

In summary, this subsection provided a brief examination of multi-class classification. We showed the common tools and methodologies used in these situations and also contrasted the differences between the two most popular techniques, lastly concluding with their deficiencies. For more information about multi-class classification, the reader is referred to Mohri, Rostamizadeh, and Talwalkar (2012, Chapter 8) which acted as a principal reference for this subsection.

2.5.5 Summary And Future Reading

The field of machine learning is vast and progress therein is even faster. This section presented a general introduction to machine learning from a computational perspective—rather than analysing individual algorithms—namely because understanding the theoretical foundations of the field is, in my view, essential for gripping the possibilities and limitations of machine learning itself. For this reason, the analysis of individual classification algorithms and techniques is not present in this section, nor are techniques for measuring the confidence of said algorithms listed here.

Individual algorithms, namely decision trees and ensemble methods, are visited further down in Chapter 3. Confidence analysis of these techniques are analysed in Section 4.4. For further reading about the theoretical computational limits of machine learning, we refer the reader to *Foundations of Machine Learning* by Mohri, Rostamizadeh, and Talwalkar (2012). This reference is differentiated from other classic works by its emphasis on the computational analysis via the PAC model and an emphasis on proofs. For more general introductions on various ML techniques and applications, the reader is referred to *Pattern Recognition and Machine Learning* (Bishop 2006) and *Machine Learning* (Mitchell 1997). The importance of statistical methods in the analysis of learning needs to be highlighted; for it we recommend *The elements of statistical learning* (Hastie, Tibshirani, and Friedman 2001). Although not visited upon in this section, ensemble methods and boosting are important techniques and an recommended source for it is the new book by Schapire and Freund (2012).

This part also concludes the whole chapter. This chapter laid the theoretical framework the upcoming chapters are based on.

2.6 Classifiers

In this thesis, we have talked about classifiers in a broader, hypothesis-oriented context, but have so far eluded their specifics. This section discusses the various classification algorithms that we used in fleet inference. We review formal definitions of classification algorithms and discuss *decision trees*, a classification algorithm. We also look at *ensemble methods*, which are about combining a group of classifiers into a single body. Then we show how these classification algorithms are trained and used in both join inference and attribute classification.

The usage of stems from the necessity to predict categorical data. Usually, the underlying question behind a classification task is whether something either belongs to a category or not. In the presence of multiple possible N categories, recall from Subsection 2.5.4, we train N classifiers in an OVA scenario, and determine which one of the classifiers is the correct one.

Using formal notation, the target space of a binary classifier is $\mathcal{Y} = \{0,1\}$. Let $\mathcal{X}$ be its input space and $f : \mathcal{X} \to \mathcal{Y}$ the target (concept) function. Given a hypothesis set H of functions from $\mathcal{X}$ to $\mathcal{Y}$, and inputs $(x_1, x_2, \ldots, x_n)$ drawn i.i.d. from an unknown distribution D, a binary classifier is the *task* of finding a hypothesis $h \in H$ that the generalisation error is small:

$$R_D(h) = \Pr_{x \sim D}\left[h(x) \neq f(x)\right]. \tag{2.13}$$

One can select different hypothesis sets for this part, but using the simplest hypothesis is usually the best choice (Mohri, Rostamizadeh, and Talwalkar 2012). The notion is due to Occam's razor, which states that *given a set of competing hypotheses, the least parsimonious, or the least complex one, is usually the correct one*. This is true in machine learning as well, the theoretical foundations for this are based on Vapnik–Chervonenkis dimensions (Vapnik and Chervonenkis 1971; Vapnik 2006) and Rademacher complexities (Koltchinskii 2001; Bartlett and Mendelson 2003). Due to the scope limitations of this thesis, we omit discussions of both VC dimensions and Rademacher complexities.

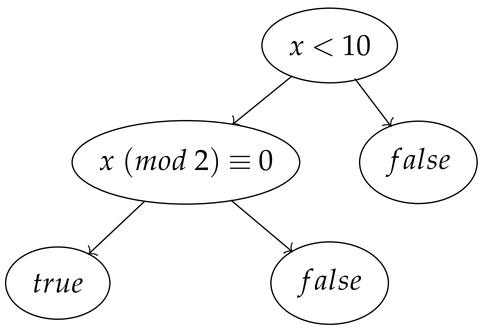
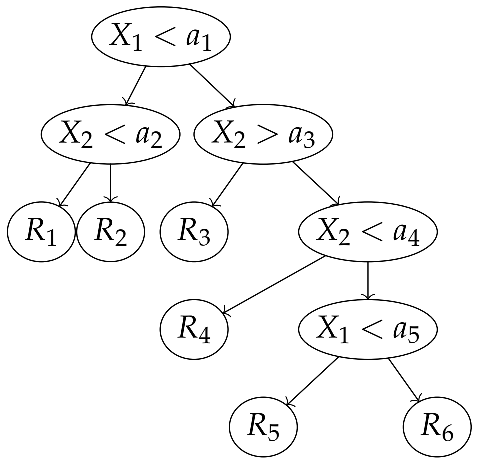


Fig. 12: An example decision tree with two leaves testing whether a positive integer x is smaller than 10 *and* even.

The classifier algorithms we have used in this thesis are *decision trees* (Breiman et al. 1984; Quinlan 1986), *random forests* (Breiman 2001), *extremely randomised trees* (Geurts, Ernst, and Wehenkel 2006), AdaBoost (Freund and Schapire 1995) and *gradient tree boosting* (Friedman 2001, 2002). Decision trees are a graph-like decision tool that work by making individual decisions in nodes and arriving to a conclusion.



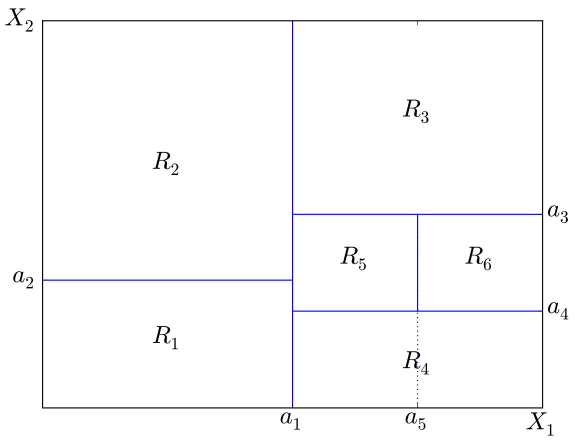


Fig. 13: A decision tree and its decision surface for two features X_1 and X_2.

These can be used in decision making in an intuitive manner: the tree can be easily visualised and one needs only to understand the visual representation. The latter three are *ensemble methods*, which combine multiple learners to form a single hypothesis. The last two methods, AdaBoost and gradient boosting, are boosting algorithms, which are a meta-algorithm that aim to reduce bias in learners.

2.6.1 Decision Tree Learning

"Trees sprout up just about everywhere in computer science."

– Donald E. Knuth (2011)

Decision tree learning is a supervised learning technique that can be used for both classification and regression. (Breiman et al. 1984) In classification, we are interested in using trees to determine correct categories for input data; in regression, we are trying to predict the outcome of a function using decision trees. The performance of decision trees is not up to par with comparable methods, but using boosting and ensemble methods, decision trees can be transformed into effective learning algorithms. (Mohri, Rostamizadeh, and Talwalkar 2012, p. 194)

In decision tree learning learning, the goal is *generate* decision trees based on observed data. We want to infer a decision tree from the ground up. To infer the decision tree in Figure 12, we would select a hypothesis that given an input space $\mathbb{Z}_+$ would give the label `true` (i.e., 1) for numbers 8, 6, 4, 2, and 0. This is called *decision tree learning*, which we will discuss in this subsection.

Formally, given a labeled input vector from $((x_1,y_1),\dots,(x_n,y_m)) \in \mathcal{X} \times \mathcal{Y}$, a decision tree is a classifier that recursively splits $\mathcal{X}$ such that the sample labels from $\mathcal{Y}$ are grouped together. For each node n, let (X_i, q_t) be a pair where X_i is some feature and q_t a question. The question can be a numerical question of the form $X_i \leq q_n$ where q_n is some number, or a categorical question $X_i \in \{a,b,c\}$. The resulting leaf nodes are the labels $l \in \mathcal{Y}$. For any input $x \in \mathcal{X}$, to get its predicted value, the tree is traversed until a leaf node is encountered. The input x is associated with the label $l \in \mathcal{Y}$. Each leaf represents a partition of $\mathcal{X}$, and no partition can intersect.

The algorithm for learning the decision tree works by iteratively selecting leaf nodes and splitting them. An outline for decision tree learning can be found in Mohri, Rostamizadeh, and Talwalkar (2012, p. 196). Here, we summarise the learning method briefly. The initial state of a decision tree is when the tree only has a root node. With each iteration, a leaf l is selected from the tree and a split is computed for it, replacing the leaf node with two internal leaves. The split is computed using *node impurity*. The idea of node impurity is based on the idea of reducing the so-called impurity of the training set. A "pure" split results in the best split of the node: a pure leaf will dominate the training set, resulting in the smallest amount of misclassification. Splits are chosen by decreasing node impurity according to some impurity function F.

Computing the impurity $F(n)$ for a node n is done as follows. Let $n^-(q_t)$ be the *left* child of the node after the split and $n^+(q_t)$ the *right* child of the resulting split nodes. For a node n, let R_n be the region of points defined by n, the proportion of observations $p_l(n)$ for the label $l \in [1,k]$ is:

$$p_l(n) = \frac{1}{K} \sum_{x_i \in R_n} I(y_i = l).$$

For a label l, let $p_l(n^-)$ and $p_l(n^+)$ be the number of points inside the region that the split generates, and K the amount of observations. The impurity is calculated using the impurity function $F(n)$:

$$G(n,q) = \frac{p_l(n^-)}{K} F\left(n^-(n,q)\right) + \frac{p_l(n^+)}{K} F\left(n^+(n,q)\right).$$

The selected split $\hat{\theta} = (n_t, q_t)$ is the one that minimises impurity:

$$\hat{\theta} = \arg\min_{n,q} G(n,q).$$

Common choices for the impurity function F are:

$$F(n) = \begin{cases} 1 - \max_{l \in [1,k]} p_l(n) & \text{misclassification,} \\ -\sum_{l=1}^{k} p_l(n) \log(p_l(n)) & \text{entropy,} \\ \sum_{l=1}^{k} p_l(n)(1 - p_l(n)) & \text{Gini index.} \end{cases} \tag{2.14}$$

Learning strategies. To learn a decision tree there are a two main strategies, *greedy* and *grow-then-prune*. The greedy strategy builds a tree until no more nodes can be feasibly split based on node *impurity*, as seen on the previous page; grow-then-prune builds a gigantic tree and minimises it by pruning it to a smaller tree using an objective function based on its *size* (the amount of leaves) and its empirical error. The greedy technique was presented as the *top-down induction of decision trees* by Quinlan (1986) in ID3, or *Iterative Dichotomizer 3*. Grow-then-prune was introduced with CART (Breiman et al. 1984).

The most widely used decision tree algorithms are C4.5 (Quinlan 1993), C5.0 and CART (Classification And Regression Trees, Breiman et al. (1984)). C4.5 and C5.0 are both improvements on ID3, the latter of which is a commercially licensed, less memory and time consuming version of the former. CART differs from the C4.5 family in that it also supports regression. The decision tree classification algorithm used in this thesis is from scikit-learn (Pedregosa et al. 2011), which uses an optimised version of CART.

2.6.2 Ensemble Methods: Random Forests and Boosting

"How is it that a committee of blockheads can somehow arrive at highly reasoned decisions, despite the weak judgment of the individual members? How can shaky separate views of a panel of dolts be combined into a single opinion that is very likely to be correct?"

– Schapire and Freund, *Boosting* (2012)

Ensemble methods are classification methods that combine different *weaker* learners into one *strong* learner. Many ensemble methods, notably random forests and extremely randomised trees, are tree based, i.e., they use decision trees as the underlying weak learners. AdaBoost, a boosting meta-algorithm, uses decision tree classifiers as its weak learners, and gradient tree boosting uses decision tree *regressors*.

The types of ensemble methods described here can be divided into two categories: methods that build a set of different learners *independently* and produces the best averaged result, called *averaging*, and methods that iteratively train on subsamples of the training data, called *boosting*. In boosting, the trees are built sequentially: each tree is dependent on its predecessors when it comes to how it is trained.

Forests of trees. Random forests and extremely randomised trees are both averaging methods. Their goal is to build a *forest* of randomised trees using varied subsamples of the data: each grown tree is shown a slightly different sample of the training data. For prediction, each tree is shown the input sample and the vote[2] of all trees is chosen as the prediction. This method works similarly for both random forests and extremely randomised trees, they vary only in the way the randomisation occurs.

Additionally, to compute the tree splits at each node, the best split among a *subset of features* is chosen, instead of the best split for the whole training data. This gives the classifier an overall reduced variance, but might increase its bias.

2. The implementation provided by scikit-learn used in this thesis does not use a mode vote. Instead, the average of the prediction probabilities of all trees is chosen.

In extremely randomised trees, the whole training set is used, but the splits are chosen at random. Consequently, in extremely randomised trees, this *extreme* randomisation results in reduced variance compared to random forests. (Geurts, Ernst, and Wehenkel 2006) The increase in bias is then compensated by training on the whole training sample, yielding a smaller increase in bias as well.

Boosting. The principle in boosting is to train weak classifiers and iteratively train them increasingly on samples that are prone to misclassification, while decreasing the amount of training on samples that are correctly classified. Intuitively, boosting focuses subsequent training iterations on samples that are "difficult" to classify, samples for which classification is easier are given less weight in training.

AdaBoost (Freund and Schapire 1995) uses this principle in its process. Initially, training samples are drawn with a uniform distribution D_1 (recall equation (2.9)). AdaBoost selects a hypothesis h_t for the distribution D_t, after checking the points that h_t classified correctly or incorrectly, the weights in the subsequent distribution D_{t+1} are given less and more weight, respectively. Once T training rounds are complete, the resulting classifier H is a sign function of a linear combination of the hypotheses weighted by their error rates α_t: $H(x) = \text{sgn}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$. (Mohri, Rostamizadeh, and Talwalkar 2012)

Gradient Tree Boosting (Friedman 2001) and **stochastic gradient boosting** (Friedman 2002) work similarly to other boosting methods, i.e. by iteratively creating new weak learners, but the main difference is the addition of a *differentiable* loss function. In other boosting methods, the loss function is usually a zero-one-loss function, but in gradient tree boosting, the loss function is minimised using gradient descent. Stochastic gradient boosting presented an important tweak to gradient tree boosting, by adding the notion of bootstrap aggregating ("bagging"): each learner is trained on a subsample (without replacement) of the original training set, introducing randomness. This is analogous to the forests of trees methods. Friedman (2002) observed that the subsample size f produces good results when $0.5 \leq f \leq 0.8$. Consequently, the implementation in this thesis uses a subsample rate of 0.5. It is worth noting that if $f = 1$, the subsample rate is identical to regular gradient tree boosting.

"Think of all the psychic energy expended in seeking a fundamental distinction between "algorithm" and "program"."

– Alan Perlis

"If you automate a mess, you get an automated mess."

– Rod Michael

3 Implementation: Fleet Inference

"The question of whether computers can think is like the question of whether submarines can swim."

– Edsger W. Dijkstra

This chapter, along with the next one, presents the principal contribution of this thesis. We present *fleet inference*, a novel data importing technique for route optimisation systems. This chapter relies heavily on the theoretical framework presented in the previous two chapters. While we reference some of the central concepts presented earlier, most of the definitions herein aim to be self-contained, with the intent that this chapter (and the next) can be read independently. Chapters 1 and 2 will help the reader to better understand the techniques used here.

Fleet inference consists of finding an efficient method for importing data for route optimisation from a multitude of formats. Preferably, these formats should be *learned*; in the sense that as little human oversight of the importing process is required. We propose the use of the following approach, which is presented in detail in this chapter:

1. finding an efficient way to link multiple data documents, e.g., CSV files, together by inferring their referential attributes, and constructing an aggregation of them, and
2. correctly inferring the context and meaning of data within this aggregated document.

The idea of *fleet inference* is to recognize a *fleet* from input data, using automated reasoning. This automated reasoning framework uses machine learning, which we discussed in the previous chapter. Generally, we aim to teach algorithms how fleets are represented. The algorithm's task is to learn an efficient method for *inferring* its structure. Hence the name *fleet inference*.

Before we discuss fleet inference thoroughly, the reader is introduced to its proper formulation. This is done in Section 3.1, where we link the concepts of *join inference* (see Section 2.3) and *attribute classification* (see Section 2.4). An overview of the system and the process it encapsulates is given in Section 3.2. Our learning aspect is deconstructed in section Section 3.3, where we look at common features used for learning the inference rules, and in Section 2.6 we look at the classification procedures our learning methods use. Finally, we conclude with a theoretical analysis on the system's robustness, that is, how well the learned fleet inference model can tolerate corrupted or missing data.

3.1 The Fleet Inference Problem

Fleet inference (FI) is essentially a composite of two different problems. The first one, join inference, is fundamentally a problem about finding referential constraints between sets of database relations. The second, attribute classification, is a schema mapping problem where we infer and assign contexts for attributes by finding associations between attribute pairs.

At a glance, these two problems seem somewhat unrelated. As noted in their respective sections, individually, these problems have been widely studied and the studies have intersected rarely. Only specific contexts of schema mapping broach the notion of join inference. This is largely because schema mapping deals with complete, and thereby *joined*, schemas, and join inference is usually done as a preparatory step for the schema matching.

What follows is a brief outline of the fleet inference process. We first teach an algorithm what referential constraints, i.e., foreign keys, look like. The trained algorithm forms a *hypothesis* of what foreign keys between two unseen relations might look like. The hypothesis is used on a set of relations and if a candidate foreign key is obtained, an equi-join is performed on the attribute deemed as a foreign key. The resulting joined relation (a combination of n possible relations) is then shown to the attribute classifier, which has been trained using a similar approach: the classifier is trained using domain specific data for a finite set of attributes. The attribute classifier then classifies each attribute in the joined relation. If suitable target attributes are found, the attribute classifier produces a list of pairings based on these potentially related attributes.

In comparison to either problem component, the FI approach differs in that we automate the whole process of finding references between relations and classifying their attributes entirely. Our overall goal is to have little oversight of the algorithm as it functions: inferring heuristics for what constitutes a suitable foreign key or attribute pairing is done by the algorithm.

Some considerations. At this point, it is very important to consider whether the first step—join inference—is at all necessary. Given that the attribute classifier could very well operate on unrelated[1] data, we simply go through each attribute and show it to the classifier. If pairings are produced, we can one way or another obtain a suitable case model from this data.

The answer becomes obvious when one considers the importance of *context* in the classification process: although pairings by themselves can be useful, how would one construct an algorithm that could properly infer the semantic meaning of each attribute? The basic case model introduced in the previous chapter has a low number of attributes, so the complexity of the attribute pairing is equal to the number of the Cartesian product of the source attribute set S and the target (domain) attribute set T, $|S \times T|$.

One could simply *guess*, and using a rather crude approach, forcibly trying to shoehorn source attributes into a sensible case model. Obviously, this is rather time-consuming and inefficient. One of the advantages of join inference is that by grouping attributes together semantically, we can usually deduce their context and meaning. Because we can group attributes such as *capacity* or *vehicle number* together, we can understand that these attributes are related to the *vehicles* in the case model.

1. We use *unrelated* in the relation sense: attributes that are unrelated may or may not reside in the same relation.

One significant caveat in the proposed fleet inference model is that for the time being, the correct separation of relations that contain information pertaining to vehicles and tasks must be done beforehand. That is, we must tell fleet inference that a particular subset of **R** pertain to vehicles and another disjoint subset pertains to tasks. An obvious improvement for fleet inference is the inference of attribute contexts while performing foreign key inference. This would reduce the complexity of join inference since less potential foreign keys have to be analysed.

The second consideration is the usage of machine learning. It is completely possible to create heuristics for detecting foreign keys or mapping attributes. As was noted in Subsection 2.3.3, machine learning methods have proven to be flexible while retaining good generalization ability. Arguably, the lack of need for creating heuristics for such a specialized data saves time—the learning algorithm need only be trained, and if sufficiently sophisticated, it can construe these heuristics on its own.

3.2 Architectural Overview

The fleet inference model and its implementation are based on the *case model* concept from earlier chapters. A case model is a mathematical construct that encapsulates components of fleets—vehicles and delivery tasks. The bipartite nature of the case model translates directly to the fleet inference model. The fleet inference model architecture consists similarly of two parts, *modules*. The join inference module constitutes the first part and the attribute classifier the second. The nature of join inference is reduction from many to one—assimilating multiple relations in **R** into one; the attribute classifier is a one-to-many relation, as it can produce a multitude of target attribute mappings for each attribute.

In describing the architecture illustrated in Figure 14, in this brief overview, we will operate in terms of *input* and *output*. The input for the join inference module is two sets of relations, one set for vehicle data, another for task. The separation at this level is currently done by the user. At a later stage, semantic inference based on neighbouring cells can be used to detect the pertinent semantic purpose of each

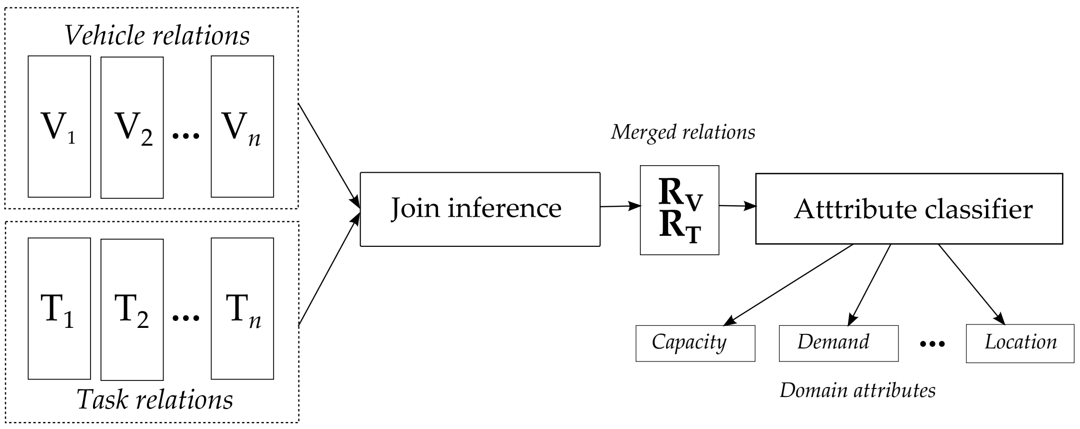


Fig. 14: The architecture of fleet inference.

relation, that is, whether a group of relations belong to the vehicle category or the task category.

Once these sets are processed, i.e., inferred into relations $\mathbf{R}_V$ and $\mathbf{R}_T$, join inference sends this output to the attribute classifier. The attribute classifier uses the domain data to detect which source relations are used in mapping domain attributes. Thus the attribute classifier would select attributes *Capacity* and *Speed* from the vehicle data, and *Location* would be mapped from the task data.

The attribute classifier produces domain data mappings for each attribute and the process of fleet inference is thus completed. Once the domain data mapping is done, all is left is data sanitising and parsing, and inputting the data into a solver system. The solver receives a file format, constructed by fleet inference, that it can interpret. In practice, I recommend that fleet inference be implemented as a separate *service*, i.e., using a *service-oriented architecture* or SOA (Perrey and Lycett 2003), which can be used to query an individual data sets' file format and structure—but not be used as a system to parse data into the solver itself.

The architecture will be further visited on in following chapters, in particular, both modules will be scrutinised in depth. This subsection ends the abstract architectural level, we now move on to examining the process of fleet inference at a practical level.

Algorithm 1: Fleet inference

```
Data: $\mathbf{R}_V$ and $\mathbf{R}_T$, domain schema $\mathcal{D}$
Result: A mapping from the attributes in Data to the domain model.
foreach relation $\mathbf{R}_V$ and $\mathbf{R}_T$ do join inference
    find primary-foreign key pairs for every relation pair;
    merge relations into prime relations $V'$ and $T'$
end
foreach attribute $\alpha \in V' \cup T'$ do attribute classification
    foreach attribute $a$ in the domain schema $\mathcal{D}$ do
        calculate a similarity score $sim(\alpha, a)$;
    end
    choose the highest ranking similarity score for $\alpha$;
end
use the sorted similarity scores to construct the schema mapping to $\mathcal{D}$;
```

3.2.1 Problem Formulation

This subsection articulates the research question presented on page 8 by defining it formally using knowledge constructed in the chapters that followed that section. We also provide an algorithmic outline of how fleet inference works, first as a whole followed by both halves in detail.

To properly present an outline of fleet inference in a human-readable format, we have have to make some assumptions on the data. These assumptions may or may not require additional work from the user. As an example, the partitioning of subsets into domain-pertinent subsets (vehicle and task data) is a requirement.

We assume that the input of fleet inference is a data set representing a VRP. This data is assumed to consist of relations. The user partitions this data set $\mathbf{R}$ into two disjoint subsets, $\mathbf{R}_V$ and $\mathbf{R}_T$. The former relation set pertains to vehicles and the latter pertains to tasks. Algorithm 1 outlines the procedure of fleet inference.

Using this algorithm outline, we can now refine the research problem found in 1.2.

> Given two relation sets $\mathbf{R}_V$ and $\mathbf{R}_T$, infer all the foreign key constraints between the relations in them. For each subset, select their prime relations V' and T', and using the foreign key constraints, merge other relations using equi-joins into these. For each prime relation, and for each attribute therein, find the highest ranking mapping to the domain schema $\mathcal{D}$.

This definition is notably more succinct than the one on page 8. We now move on to examining how the individual processes, i.e., those described in the for–each loops, work.

3.2.2 Module Overview

"An algorithm must be seen to be believed."

– Donald E. Knuth (1968)

This subsection provides additional detail on the procedures of join inference and attribute classification. We look closely at the requirements of both modules, join inference and attribute classification, by examining their data inputs and outputs. Sections of Algorithm 1 are opened and the pseudo-code lines are explained. This subsection is still about architecture, in that we deal with the modules as an input–output pipelines, i.e., specific components of a larger whole. Actual discussion of how classification algorithms work, or how the features for classification are selected, are discussed later on.

To begin with, join inference, due to its complexity, is a time-consuming process. For join inference to function properly, it requires well-trained classification algorithms. Recall from Subsection 2.5.3, with *well-trained* we mean that the algorithms have a low generalisation error. The algorithms have to be *robust*, i.e., their *softness margin*—how much errors in the input data are tolerated—must be sufficiently big to accommodate for errors in the data.

In contrast, attribute classification has less information to manipulate, and its input is sanitised by join inference. Thus attribute classification has to make fewer assumptions about its data, as the input data consists of a set of attributes with their respective tuples (value sets).

```
Algorithm 2: Find pairings and classify
Data: a set of relations $\mathbf{R}$ for which $|\mathbf{R}| \geq 2$.
Result: a list of primary-foreign key pairings
foreach relation $R_i \in \mathbf{R}$ do find pairings
    initialize list $FKC[i]$;
    foreach relation $R_j \in \mathbf{R} \setminus \{R_i\}$ do
        initialize list $FKC[i][j]$;
        foreach attribute pair $(a_n, a_m) \in R_i \times R_j$ do
            if $\neg isIND(a_n, a_m)$ then is not IND
                skip this pairing;
            else
                $p \leftarrow Classify(a_n, a_m)$;
                if $p$==1 then is FKC
                    append $(n, m)$ to the list $FKC[i][j]$;
                end
            end
        end
    end
end
```

Due to these requirements, we have to make a couple of assumptions. We assume that the join inference module has been trained properly. This training aspect is covered in the next section; for now, we assume that the training data set consists of large normalised relational data sets residing in flat CSV files. The other assumptions are the following pre- and post-conditions: (1) both subsets have to have at least 2 relations, or no join inference is performed; (2) for n relations, the output of primary–foreign key relations must number $n - 1$; additionally, (3) during join inference, if any relation has two (or more) foreign key candidates to a primary key,

```
Algorithm 3: Pruning excess pairings.
Data: List of pairings $FKC[i,j]$
Result: Pruned list $FKC[i,j]$
foreach pair $(i,j)$ in FKC do loop relation pairs
    $count \leftarrow CountFKC(FKC[i])$;
    if $count \leq 1$ then
        one pairing or less, ok, one-to-one;
    else
        foreach pair $(n,m)$ in $FKC[i][j]$ do prune excess pairings
            count total of pairs for $n$ and $m$;
            if one-to-many for $n$ then choose best $m$;          /* many $m$ for $n$ */
            else if many-to-one for $m$ then choose best $n$;     /* many $n$ for $m$ */
            else if many-to-many for $n$ and $m$ then choose best $n$ and $m$;
        end
    end
end
```

excess ones must be pruned.

Condition 1 is straightforward: if only one relation per subset is provided, join inference is unnecessary or cannot be performed. Condition 2 restricts the prime relation from referencing itself. Condition 3 gives the instruction on how to enforce condition 2. The method for performing 3 is currently done by hand.[2]

Finding all possible FKC. Finding FKC starts by creating a list of possible IND candidates, also known as *spurious* IND (Rostin et al. 2009). These are created by iterating over all attributes in a relation subset **R** and trying to look for IND to every other attribute in the set of attributes.

More formally, let $\mathcal{A} = \{A_1, \ldots, A_n\} \in \mathbf{R}$ be the set of attributes we work on. For each attribute A_i, test for IND with every attribute in $\mathcal{A} \setminus \{A_i\}$. The source attribute

2. This issue is noted and will be elaborated on further in Section 4.4

A_i is called a *referenced attribute*, i.e., a potential primary key to a foreign key, and the set of attributes an IND is tested to are *dependent attributes*. As a result, the IND algorithm iterates over $n \times m$ attributes.

For two attributes A and B, let $s(A)$ and $s(B)$ be their distinct values. To satisfy and IND, recall from Definition 3, if $s(A) \subseteq s(B)$, then $|s(A)| \leq |s(B)|$, or for $s(A) \subseteq s(B)$, it must hold that $|s(A)| \geq |s(B)|$. The number of subset inclusions we need to perform is then $\frac{n^2-n}{n}$ (Bauckmann, Leser, and Naumann 2006).

Testing for the subset inclusion is a simple combination of the coverage and out of range features: $\texttt{coverage} \geq 0 \wedge \texttt{outOfRange} \leq 1.0$. These are the same features as described on page 74.

Once all IND candidates have been retrieved, features are calculated for IND candidates (see Section 3.3) and candidates run through the binary classifier that tells whether the candidate is a possible FKC or not. The binary classifier algorithm returns either 1 for a suitable foreign key pair, or 0 for a non-suitable one.

This IND approach is the brute force method as described by Bauckmann, Leser, and Naumann (2006) and later used in foreign key discovery by Rostin et al. (2009). Bauckmann, Leser, and Naumann (2006) describe an advanced, computationally efficient mining algorithm called SPIDER which is considerably faster than the brute force approach. I selected the former brute force approach because the training and featuring aspect are not a significant aspect of the process. Furthermore, the number of attributes in any data set that would be used in fleet inference is relatively low, the only amount when the number of attributes might be high would be in the training process.

Using found FKC. Let S, R be relations in $\mathbf{R}$. If the number of FKC for a relation pair is **one-to-many**, i.e., if the algorithm finds a key in S that has two (or more) possible foreign keys in R, a FD $\mathcal{F}(A)$ is computed for each attribute $A \in R$. The attributes are then ranked by their FD, and the key with the highest FD is selected. If the result is **many-to-one**, then the situation is reversed. If the result is **many-to-many**, then the sorting is applied on both. Algorithm 2 shows how all pairings are found,

```
Algorithm 4: Classify attributes
Data: A relation union $\mathcal{S} = V' \cup T'$, target attributes $\mathcal{T}$, domain schema $\mathcal{D}$,
      classifier set $\mathbf{C}$
Result: A mapping from $\mathcal{S}$ to $\mathcal{D}$
initialize pairing list $pairs$;
foreach attribute $A \in \mathcal{S}$ do
    $data \leftarrow \Pi_A(\mathcal{S})$;
    foreach attribute $T \in \mathcal{T}$ do
        $clf \leftarrow \mathbf{C}[T]$;
        if $clf(A, data)$ then                              /* classification match */
            $Append(pairs,\ (A,\ T))$
        end
    end
end
return $pairs$
```

Algorithm 3 displays the pruning process in cases differing from one-to-one cases.

To find the *prime relation*, we select the one that has the highest number of **one-to-x** mappings per attribute, where x is either **one** or **many**. This means that the primary relation will likely be the one with its key being referenced the most by other keys. Choosing the best relation simply sorts the pairs (n, m) for each individual element in the pair and picks the best n or m. This can be done by sorting the attributes by their FD within their own relations, and selecting the attribute with the best FD score to act as the referential attribute. To compute the FD score, the TANE algorithm by Huhtala et al. (1999) can be used.

The merging process starts by finding the prime relation. Find the i in FKC which has the biggest number of true pairings (i.e., ones) in its list for every relation, i.e., sort $FKC[i]$ by counting its js for which $FKC[i][j]$ has a pairing. Fix the found i as the prime relation, then loop through its js and merge iteratively using equi-joins on the attributes that make the pairing.

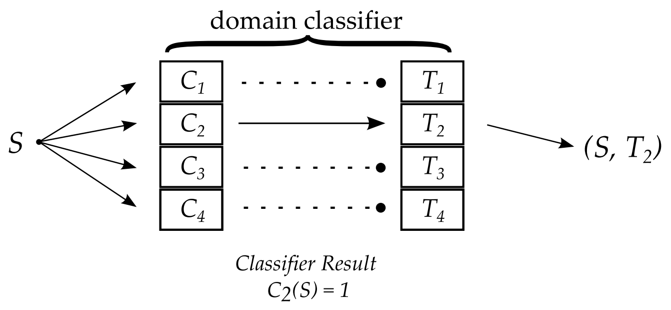


Fig. 15: An example of attribute classification. Four classifiers are trained for four domain attributes. In this example, C_2 returns `true` for S matching T_2. The *domain classifier* has been trained with domain data for each attribute.

Classifying attributes. Once the merging process is complete, we have two relations, V' and T'. we can classify the attributes of the relations. Suppose our domain schema $\mathcal{D}$ has n attributes. Train n binary classifiers, one for each attribute. Give each attribute $A \in V' \cup T'$ to every classifier. If any classifier returns 1 for an attribute, map that attribute to the domain schema attribute corresponding to the classifier. If multiple attributes are assigned 1, currently the algorithm relies on user input to choose the best attribute. This could be improved by refining the classification algorithms and using better features, but this is left as an obvious improvement. The procedure of attribute classification is illustrated in Figure 15 and also shown in Algorithm 4.

This concludes the architectural overview of fleet inference. In this chapter, we presented thorough design and architectural features of the implementation itself, and demonstrated some of the details using pseudo-code algorithms. We now move to discussing feature selection, and how the classification algorithms we mentioned work in practice.

3.3 Feature Selection

Choosing good features is decidedly an important part of creating a classifier, in fact, according to Hastie, Tibshirani, and Friedman (2001), it is arguably more important than the choice of a classifier. In general, a *feature* is any kind of knowledge that pertains to a specific learnable concept. In spam classification, an example feature might be whether a sample document contains questionable words.

Obviously, selecting the actual features that are used in classification is a crucial step. A classification problem might have a vast number of features, yet only some of these are relevant. In fact, ”the selection of relevant features, and the elimination of irrelevant ones, is one of the central problems in machine learning” (Blum and Langley 1997). There are various methods for selecting the best features for any given problem. Guyon and Elisseeff (2003) provide a useful check list to find out what features to select, the first in which states that if one has **domain knowledge**, one can construct features in an *ad hoc* manner.

Conveniently, a part of our classification problem is domain-bound: attribute classification. We can construct good features just by looking at the target data to which we are mapping. Join inference, on the other hand, is a more generic problem: it is not data centric *per se*. The rules of join inference can be applied to any data set—while attribute classification, in the scope of this thesis, are exclusively rooted in the routing context.

Thus, for join inference, we have selected features that have been used by others in literature, and developed some our own, by using example domain data. An important point is that the domain data used to test join inference is only exemplative in natural. None of the features can be construed reliably based on simple observations of the data. As such, we have rely on the empirical knowledge built by others, and using our own judgment when constructing new ones.

3.3.1 Join Inference

Features in join inference exhibit a notable lack of domain knowledge. Domain knowledge, in short, is knowledge about the data or its environment. As mentioned in the previous paragraph, attribute classification is directly dependant on domain knowledge, whereas join inference is not. This is because with attributes we are trying to look specific correspondences to our own domain data, but in join inference, the structure and contents of the relations is unknown. The influential paper by Acar and Motro (2009) underlines the fact that with schemaless data there is no prior information on the domain.

While our problem is decidedly in a highly specialised context, wherein domain-based information is abundant, we similarly lack any domain knowledge of the structure and contents on *real* vehicle routing problems. With real we mean problems that are based on actual real-world transportation problems, not problems that are relevant only in academic contexts. We should be making some educated guesses about the data, but we have chosen to remain domain-agnostic. This is because we do not want to construct heuristics or hypotheses by ourselves. I believe that by letting an algorithm construct inference rules, hypotheses, and heuristics, the process results in an methodology that is more efficient, more precise and most of all, more practical.

The domain agnostic approach is a perfect justification for a machine learning approach. Consider the fact that constructing inference rules from data first of all *requires that data*. I have tried to mitigate the ramifications of this issue by using large relational data sets to train the learning algorithms. Secondly, machine learning algorithms have been expressly constructed with this goal in mind—constructing hypotheses, as we learned in Section 2.5.

Operating method used by our machine learning approach is simple. Train a classification algorithm on big relational data sets, which results into a hypothesis of what primary key–foreign key relations look like. After training is complete, use the classifier in our problem domain. For our purposes, we trained the classifier on the HetRec'11 Last.fm, IMDB and Delicio.us data sets (Cantador, Brusilovsky, and Kuflik 2011).

This approach was also selected by Rostin et al. (2009) for foreign key discovery. Their method was a decision-tree based classifier trained on six different data sets. The method was then cross-validated on the other data sets. We perform a similar measurement of join inference, where we perform LOOCV on the three data sets.

Chosen features. The features chosen for join inference are listed below. Each feature has an literature origin labeled A for Zhang et al. (2010), B for Rostin et al. (2009), C for Acar and Motro (2009). Items prefixed with a star ($\star$) were modified by me. The appropriate origins are listed in parentheses, with their source names appearing after the colon. The features marked with † are not implemented in the presented version.

Notes. In Rostin et al. (2009), feature 4, unary dependence, is split into two features, MultiDependent (F4), which is the number of times the values of an attribute appear in the set of all IND value sets, i.e., decreasing its chance of being a FKC; its antagonist, MultiReferenced (F5), which counts how many times the values are *referenced* in the set of all IND value sets, increasing its chance to be a PK.

0. **Domain compatibility**. Attributes must have a similar data domain, i.e., in programmatical terms, this means that their data types must be an exact match.
1. **Cardinality**. A FKC should consist mostly of distinct values. In other words, it should contain few if any duplicates. (A, B: F_1)
2. **Coverage**. An inclusion dependency should *cover* the values of the PK it refers to. (A, B: F2)
3. **Uniqueness**. A FKC should not have any FKCs referring to it, i.e., a FKC is not a PK to another FKC. (A, B: F3)
4. **Unary dependence**. A FKC should reference only one PK.
5. $\star$ **Name similarity**. The column names should be similar. This uses string similarity, A uses exact name matches, B did not specify. (A, B: F6)
6. **Length difference**. The difference of average lengths of the values as strings for the PK–FK pairs. (A, B: F7)
7. **Completeness.** The amount of values in a FK that *are not* in the PK. (A, B: F8)
8. **Typical suffixes.** FKC usually end in a suffix that is characteristic of keys, primary or foreign. Examples include "_ID", "_KEY", and so on. (B, F9)
9. **Size ratio.** Usually, a PK–FK candidate pair has similar approaching ratios. (B, $F_1$0)

Implementations. The features listed above are relatively simple in their own right.

Most of these features were implemented using the functions provided by the pandas (McKinney 2011) toolkit, and NumPy and SciPy (Jones, Oliphant, and Peterson 2001; Oliphant 2007).

We improved on feature 5 using a soft string metric using Damerau–Levenshtein distance, a combination of Levenshtein distance (Levenshtein 1966), which is based on *edit distance*, i.e., the deletion, insertion and substitution of a character; with Damerau transpositions, i.e., swapping two *adjacent* characters with each other (Damerau 1964). The distance margins used are discussed in Chapter 4. Feature 5 was implemented using Damerau–Levenshtein distances where the cost of deletion is zero. In other words, terms such as `foo` and `foo_id` are considered equal. This is because, as corroborated by Rostin et al. (2009), PK–FK pairs exhibit shared suffixes such as "_KEY" or "_ID" (see feature 8). I noted that this suffix was sometimes *not* present on the referenced primary key. As a result, the name similarity uses free deletion. The distance itself is a free parameter of the classification algorithm, thus the classifier itself decides what is an appropriate distance for typical columns.

Features not implemented. There were a couple of features that I did not implement and are left as future improvements and refinements.

1. † **Functional dependency**. The strength of the functional dependency for a PK–FK within their own relation, i.e., for relations **R** and **S** and attributes $R.A$ and $S.B$, the strength of their FD $A \rightarrow \mathbf{R}$ and $B \rightarrow S$. (C) *Will be implemented using the TANE partition algorithm from Huhtala et al. (1999).*
2. † **Information gain.** Using information entropy based on the Kullback–Leibler divergence metric (Kullback and Leibler 1951), this feature measures the information gain when two relations are joined using the PK–FK link. Details about this feature are given below. (C)

3.3.2 Attribute Classification

The features for attributes were built using domain knowledge. Domain knowledge allowed us to construct *ad hoc* features that closely modelled what we would expect as input to a VRS. Since each attribute and its set of tuples are statistically distributed in one way or the other, the features model their resemblance to actual data. As Doan, Domingos, and Halevy (2001) suggest, using domain knowledge in creating schema matching can be helpful. Our aim was not to create a general purpose attribute classifier, hence we can restrict ourselves to the domain of vehicle transportation.

Consider for example a slightly heterogeneous fleet of vehicles for which the different classes $C = \{18, 38, 60\}$, where each weight class $c \in C$ is a vehicle's gross weight in metric tons. Suppose we have a fleet of 20 vehicles, of which four have a weight class of 18 tons, eleven of 38 tons, and five of 60 tons. We know that if we plotted the data into a histogram, the chart would peak at $x = 38$, as Figure 16 shows.

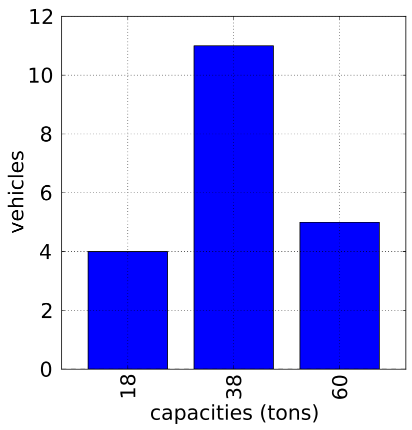


Fig. 16: Sample capacities in our fleet.

In general, my observations have noted that fleets tend to have low heterogeneity, because vehicles fleets, even large ones, tend to consist of only a handful of different vehicle types. These could be lorries or articulated lorries with distinct weight limitations, categories, and types.

Based on this brief domain excerpt, we can infer the following: first, capacity (or in this case, gross weight) tends to have very few distinct values, and the values are integers, hence capacity might have variance. Depending on the heterogeneity of the fleet, capacity distributions may skew to one direction on the x-axis (weight), and may peak at one point in the distribution, at the most common vehicle weight class, hence having high *kurtosis*. Although this sounds speculative, I believe that *any* distinctive profiles and shapes of the data can be captured with statistical moments: mean, variance, kurtosis, and skewness.

Obviously, it would be counterintuitive to make inferences on such small sample sizes. Therefore, it is imperative that the training aspect is done with a significant data volume to maximise reliability and reduce the margin of error of statistical classification methods. This is the case with statistical moments, the mean, variance, skewness and kurtosis; our sample size must be big enough for these shape profiles to be used reliably. The moments allows quantitative evaluation of the shape of domain data, but the domain data sample must be large enough for its shape profiles to have any relevance. Another caveat is that these features can only be used with numerical variables, hence we exclude categorical or string attributes from these features. To qualify for quantitative measurements we exclude variables that have a non-numerical data domain or the number of distinct values is 1.

Selected features. The following is a listing of the selected features. Each feature operates on an attribute and its tuple, therefore, for a relation R and attribute A, the input of the feature is $\Pi_A(R)$, if the attribute is sequence-specific.

1. **Average word counts**. Counts the average number of words in a sequence. Usually addresses consist of bigrams (two tokens, here, words) in Europe, e.g., “Streetname 2”.
2. **Average value lengths.** Counts the average length of values converted to strings.
3. **Average number of digits.** Counts the average number of digits in the sequence.
4. **Statistical moments**. Mean, variance, skewness and kurtosis.
5. **Time series.** Indicates whether the sequence is a time series. Implemented by feeding each tuple into a string-to-time parsing function and measuring whether all of the values in the sequence evaluate to true.
6. **Name similarity.** Name similarities between two attributes. Implemented using the Damerau–Levenshtein distance with free deletion.

Implementation. Similarly to the features for join inference, the features here were implemented using the same libraries (Jones, Oliphant, and Peterson 2001; Oliphant 2007). Features 1-3 are relatively easy to implement, as their implementation consisted of converting values into strings. For feature 4, which consists of four different features, we used standard functions in the pandas library. Name similarities were implemented using Damerau–Levenshtein edit distances.

In summary, this section presented the practical aspects of this thesis. We focused on describing the different tools and methodologies that were used in building fleet inference. We used the theoretical knowledge presented in Chapter 2 and built on existing work to create fleet inference. We now on move to the *experimental* section, where we present the experiments and the results, and analyse the performance and quality of the presented implementation.

"Maybe the only significant difference between a really smart simulation and a human being was the noise they made when you punched them."

– Terry Pratchett, *The Long Earth*

4 Experiments & Benchmarks

In this chapter, we test our implementation on real data, and where applicable, compare our results to the ones found in other academic publications. We provide empirical evidence to show that (a) join inference and attribute classification can be easily done via machine learning (b) their combination, fleet inference, actually *works* in practise. The methodologies for proving both points are relatively simple, but they require different evaluation metrics. The reason for this is that while join inference and attribute classification have been researched in conjunction and separately. Fleet inference as a problem is new, and existing methods for solving it are nonexistent.

For join inference, finding comparable benchmarks and training data was not problematic. We reviewed the literature and used some of the influential papers and performed tests similar to them. This included finding data sets for which we could provide comparable results. For attribute classification, the task was harder: as attribute data is heavily domain specific, finding comparable benchmarks proved to be impossible. This extends to fleet inference: as a new problem, with little data to draw influence from, we had to resort to synthetic data sets when evaluating the whole system.

For attribute classification, the experiments relied on synthetic example data that would in a real situation represent input data for an optimisation problem. The attribute sets were kept simple, since likely use cases will contain only a relatively small set of attributes.

This chapter is divided into three sections: training, in which we show how the learning algorithms were trained; feature selection, where less relevant features from the training sets are pruned; performance, where we test the classifiers on actual data and tweak the classifiers using hyperparameter tuning.

4.1 Training Data

As was discussed previously, classification algorithms depend heavily on the data used for their training. With larger training sets we can increase their accuracy. This section details five data sets we used on training the join inference module of fleet inference, as well as presenting the synthetic data set and case model data that we used in training attribute classification.

We used different data sets for join inference and attribute classification, as they are fundamentally two different problems. The data sets used for join inference are shown in Table 2.

Name	Tables	Attributes	Tuples	IND	FKC
IMDB	10	37	1.3M	49	11
Last.fm	5	15	334k	11	4
Delicio.us	5	18	1M	6	5
TPC-H	8	61	8.7M	147	11
SCOP	4	11	470K	11	5

Table 2: Description of the data sets used in training of the join inference classifiers.

Join inference. The classifiers trained on the HetRec'11 (Cantador, Brusilovsky, and Kuflik 2011), TPC-H (Transaction Processing Performance Council 2013), and SCOP (Conte et al. 2002) data sets. The first three data sets consisted of three database dumps from the Last.fm, IMDB and Delicio.us websites. The IMDB data set is based on the **MovieLens10M** data set, which is a data set of movies combined with links to their IMDB[1] and Rotten Tomatoes[2] review websites. The Last.fm dataset is a dump of users and their tags of favourite artits from the Last.fm website[3]. Lastly, the Delicio.us is a data set from the Delicio.us link sharing website (`http://www.delicio.us`). The TPC-H is a transaction processing performance benchmark, a synthetic database used in measuring database performance. We selected it for convenience,

1. `http://www.imdb.com`
2. `http://www.rottentomatoes.com`
3. `http://last.fm`

as it contained a relatively large number of tuples, tables and FKC. The SCOP dataset is a protein database, which served as an additional training set.

These data sets were chosen because of their use in other academic publications: in real world fleet inference, we would combine a multitude of different data sets, iteratively improving its accuracy. Due to fleet inference being at very early stages, and the limited amount of available fleet data to train on, we used these data sets to gain initial insight on the performance of the system.

Testing was conducted using data-set based cross validation: for 5 data sets, train the classifier on 4 data sets and evaluate its performance on the left out data set. Actual testing of fleet inference was then used on a synthetic database consisting of three tables, representing a rudimentary fleet of vehicles.

4.2 Feature Selection

Which one of the features we listed earlier are actually relevant? Which one is more relevant: column name similarity or value length similarity? Are some of these features completely insignificant? Questions like these arise in *feature selection*, where the problem is selecting the relevant features of a feature set.

Feature selection (also known as *variable selection*) has its benefits: reducing the number of features reduces model complexity and training time, thereby speeding up the classification process. According to Meinshausen and Bühlmann (2010), it is notably a difficult problem when the feature space has a high dimensionality. The reason we incorporate feature selection in this thesis is to study its methods and impact: we can show that for some of the classifiers, feature selection actually improves its accuracy (see the next section).

We used feature selection only for join inference, and not for attribute classification. The reason for this was that attribute classification has a total of N classifiers (for N attributes), for which the features used are usually a subset of all the attribute classification features, performing feature selection would not yield anything consistent

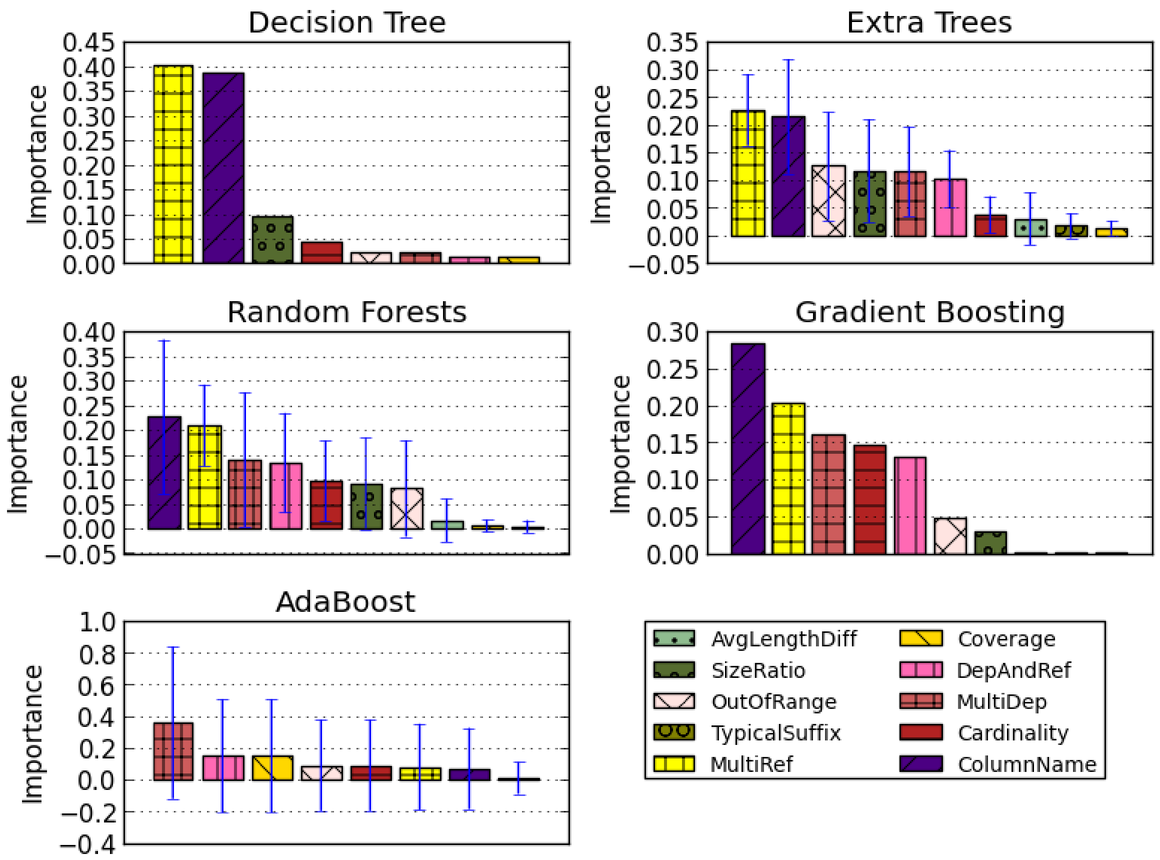


Fig. 17: Feature importances for the four different tree-based classifiers used in join inference, after training on all data sets. Features `ColumnName`, `MultiReferenced`, and `TableSizeRatio` were consistently among the most important features. For the ensemble methods, the blue error bars show a confidence intervals in which 95% of the estimators' values are distributed.

or something that we could generalise for *all* classifiers. The same reason applies for a per classifier basis: useful individual features are so few in number for each feature and the total number of features per classification task is already small. We may consider feature selection for join inference to be extraneous itself for such a small number (10) of features, hence replicating the same effort for attribute classification would generate an unneeded amount of work.

But for the rest, the variance was too broad and the importances non-applicable. As such, feature selection is only done for join inference, and for attribute classification

Method	RLR	Anova	χ^2
`Coverage`			
`ColumnName`	x	x	x
`DistinctValues`			x
`MultiDependent`	x	x	x
`MultiReferenced`			
`DependentAndReferenced`			
`TableSizeRatio`	x	x	
`OutOfRange`	x	x	
`ValueLengthDiff`			x

Table 3: Feature selection for all features. For Anova F-values and χ^2 we selected the K best features.

it is omitted. It should be noted that a quick observation of a set of eight attributes in the target domain produced a heightened importance for attribute name similarity, but other features were not as consistent.

For join inference, we can begin at looking at the feature importances as a visual representation reported by each classifier trained on all data sets in Figure 17. For decision trees, it seems that only `MultiRef` and `ColumnName` trump all others; for forest-based ensemble methods `MultiDep`, `DepAndRef`, `Cardinality` and `OutOfRange` are important too. Gradient boosting reports a mixture of the two models, whereas AdaBoost ranks all methods rather equally.

Feature selection for join inference was done with three different methods: selecting k best features (where $k = 4$) using either χ^2 ranking or Anova F-value based ranking, and *randomised logistic regression* (Meinshausen and Bühlmann 2010) (RLR). RLR is an effective method for sparse features in particular, that is, features only a small subset of which is important. As we can see from Figure 17 and Table 3 that for most classification methods, only up to four features are most discriminating. The results of randomised logistic regression can be seen in Table 3. The tree methods of

Test set	DT	GTB	AdaBoost	ExtraTrees	RF
Last.fm	0.62	0.62	0.62	0.62	0.62
IMDB	0.47	0.47	0.47	**0.85**	0.84
Delicio.us	1.0	1.0	1.0	1.0	1.0
SCOP	0.77	0.77	0.77	0.75	0.77
TPC-H	0.65	0.65	0.65	0.60	0.65
Average	0.7	0.7	0.7	0.76	**0.77**

Table 4: F_1 **measures for cross validated data sets in join inference.** The data set in the leftmost column is the *test set* and the training sets are all the other data sets *except* this set. DT = Decision Trees, GTB = Gradient Tree Boosting, RF = Random Forests. As can be seen, all classifiers predicted all results correctly for the Delicio.us data set, while Random Forests came as the most accurate predictor.

the scikit-learn library also provide a built-in feature analysis. The built-in feature selection algorithms were used in obtaining the results of Table 3 and Figure 17.

4.3 Performance

Measuring the performance of the join inference classifier was done by testing its F_1 measures using cross-validated data sets. Because training and testing on the same data sets can be misleading (see Subsection 2.5.2 and Subsection 2.5.3), using the example benchmarks of Rostin et al. (2009), for training, the data was split using a *leave-one-out* cross-validation. This means that when evaluating the performance of each algorithm, for total of N data sets, we select $N - 1$ data sets for training and test it on the left out data set. In other words, the benchmark loops over each data set and excludes that data set from training at each step.

The F_1 measure is the harmonic mean of *precision* and *recall*: precision is the number of true positives divided by the sum of true positives and false positives, i.e., the fraction indicating the quantity of true positives out of all positives; recall is the number of true positives divided by the sum of true positives and false negatives, i.e., the number of true positives that could have been retrieved. To paraphrase, precision is ability to classify correctly across its findings, recall is the ability to *find* correct classifications.

Labeling the number of true positives with TP, false positives with FP; true negatives with TN and false negatives as FN, **precision** is defined as

$$\text{precision} = \frac{TP}{TP + FP}$$

and **recall** is given by

$$\text{recall} = \frac{TP}{TP + FN}.$$

The F_1 measure is calculated as the harmonic mean:

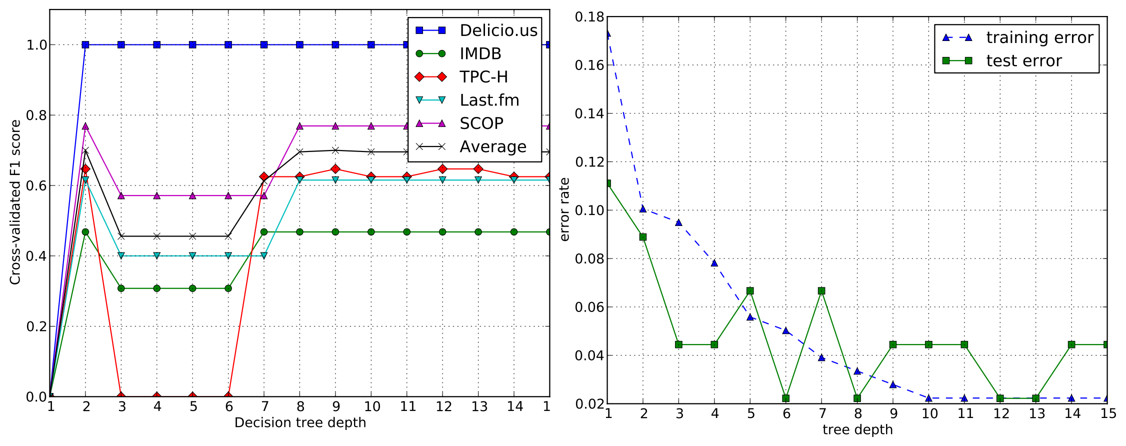
$$F_1 = 2\frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}.$$

Evaluating the algorithm using F_1 measures provides insight on its ability to generalise. The F_1 measure has a maximum at 1 and a minimum at 0, when the algorithm is at its best or at its worst, respectively. We can observe the effect on F_1 measures when tuning an algorithm by adjusting its set of parameters θ. This is a form of hyperparameter tuning or as previously mentioned, more broadly, and more commonly, referred to as *model selection*.

4.3.1 Model Selection: Hyperparameter tuning

An algorithm can be measured by its generalisation performance, which is determined by its ability to predict correctly on independent data (Hastie, Tibshirani, and Friedman 2001). This practise is called *model selection*, which is the general selection of a learning model based on its generalisation quality. The techniques and common methods used in model selection have been elaborated in Subsection 2.5.3.

Tuning the free parameter set θ of a learning algorithm greatly adjusts the learning algorithm itself, consequently, the prediction outcome is adjusted as well. To illustrate, let us consider decision trees. When limited to a tree depth of one, a decision tree will be prone to misclassification. Conversely, not limiting this depth at



(a) (b)

Fig. 18: **Decision tree depth.** Measurements of the effects of decision tree *depth*, where depth is $x = [1, 15]$. (a) measures the effects on the cross-validated scores as seen previously, where each data set is tested on and all the others are trained on. In (b), we measure the general training versus test error by training on *all* data sets using cross-validation where 20% of the tuples of the data set is tested on and the remaining 80% acts as training data.

all might result into overfitting and similarly poor generalisation. With a huge data set, the tree might reproduce itself at one node and continue this an infinite amount of times. This applies more generally to most learning algorithms that have some free parameters. For decision trees a common parameter, as mentioned above, the tree depth; for gradient boosting it might be the subsample size f and for AdaBoost the number of training rounds.

The effects of tuning the tree depth and measuring generalisation capability via F_1 measures can be seen in Figure 18. It can be seen that the F_1 measures of decision tree stabilises after tree depth d of 10. Model selection by using the built-in model selection methods of `scikit-learn` and using their default values for all algorithms where available. Only for decision tree learning did we see some notable variation

4.4 Analysis

This section is about comparing fleet inference to other similar methods and evaluating its performance based on its competition. The next section revolves around a qualitative analysis on its ability to solve the problem it was meant to; this section gives an indicator whether the technologies that form fleet inference could compete with other similar problems.

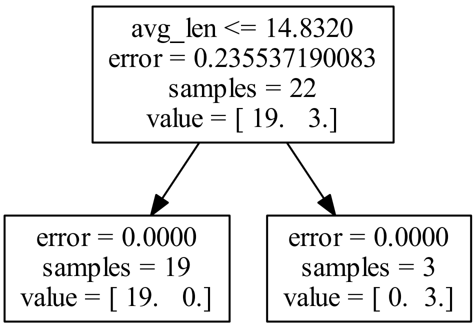


Fig. 19: Decision tree for the *location* attribute. The learned tree found only name difference to be significantly discriminating for task locations.

Finding comparable benchmarks and experiments proved to be difficult. This was due to the to the high domain specificity of our problem. Anything resembling fleet inference was not found in an extensive literature review, therefore, we omit any quantitative analysis of the whole method, and instead focus on evaluating its parts, join inference and attribute classification. For join inference, we found quite a few attempts at tackling the problem, which belatedly proved to be a lot better than our implementation. The comparisons can be found below.

Attribute classification showed similar difficulty: the domain specific nature of fleet inference removes any ability to produce any general results. Some example techniques were retrieved from influential work in schema matching, e.g., Sekhavat and Parsons (2012) and Naumann et al. (2002), but no comparable results were found.

Comparison of join inference. Evaluating the join inference classifier shows that its performance is sub par compared to the work of Rostin et al. (2009), who show consistent F_1 measures of 1 across all cross-validated data sets. Their method relied on machine learning, the problem was to correctly classify IND into FKC, which join inference is very similar to. Their classification method was a modified version of the C4.5 decision trees. Their reported learned decision tree was very similar to ours and feature selection also showed the same features, yet our results show that using similar experiment methods produced worse results. We used some, but not all, of the data sets, this could have affected the outcome.

Similarly, comparing the results to Acar and Motro (2009) shows that their algorithmic approach is more reliable and scalable compared to ours. Machine learning methods always rely on training, the method used by Acar and Motro does not require any—which can be advantageous in some cases.

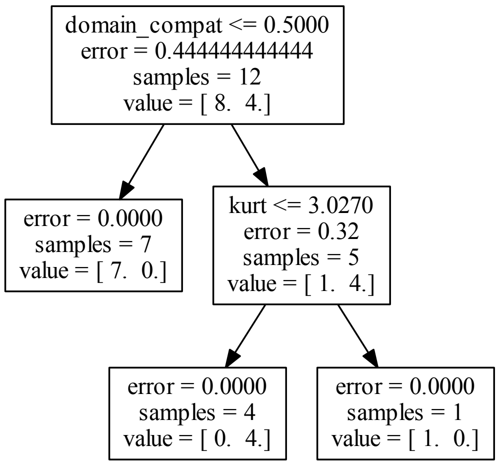


Fig. 20: Decision tree for the *capacity* attribute, showing how leaves were created for *domain compatibility* and *kurtosis*.

Another algorithmic approach is by Zhang et al. (2010), who produce a method called *Randomness* that detects single and multi-column foreign keys. Their performance also trumps the presented methodology by join inference. Their method proves effective and also surpasses our implementation of join inference due to its ability to detect multi-column foreign keys. It begs the question of why not using their method. Thus far, the algorithmic solutions we have encountered seem to satisfy our needs, when considering performance. The answer is that I deliberately chose machine learning methods as an item of study, instead of using the best method available. Hence the implementation of, and benchmark comparisons, to the methods described by Zhang et al. (2010) and Acar and Motro (2009) were been omitted intentionally.

Thus, it can be safely said, that our implementation of join inference does not match that of the state-of-the-art. However, it should be noted that our purpose was not to implement the best join inference method that could compete with the other problems, but to implement a robust, working method that could infer relationships within data sets, in order to speed up the element of input processing.

Improvements on join inference could be reached in a multitude of ways: (a) creating new, better features and developing existing ones (b) finding more and adding more suitable data sets, based on the problem domain, and (c) increasing the number of data sets and fine-tuning the algorithms with hyperparameter tuning even further to produce an even better classifier . Although item (b) carries the risk of overfitting (training the classifier too much will reduce its ability to generalise!), the other items provide clear points to work upon.

Attribute classification comparisons. Attribute classification itself was tricky to combine as comparable work did not exist at the time of writing. What is more, testing had to be done with synthetic data that, to a degree, closely resemble real-world transportation data. For testing purposes entirely, we implemented two features: *capacity* and *name similarity*, based on the previously developed attributes. Training the two classifiers on these two data sets produced the decision trees seen in Figure 19 and Figure 20.

Testing the implementation. For fleet inference, as I was not aware of any existing solutions similar to it, no comparative testing was produced. Therefore, a simple implementation test case that can offer a glance into how fleet inference functions once implemented follows.

Using a synthetic case data set, we constructed the fleet inference module using two sets of classifiers, for join inference and attribute classification respectively. We built the join inference classifier by using all the testing data presented earlier (all five data sets) to generate one classifier. The attribute classification module was built using the two attribute classifiers (location and capacity) presented above. The synthetic data set was external in the sense that its contents did not intersect with any of the training data. Once the case data was constructed, it was split into its corresponding *vehicle* and *task* subsets. For vehicles, the fleet inference classifier was able to correctly to link two tables referenced by a primary–foreign key relation, and was able to locate the column for the *capacity* attribute. For tasks, the same happened with *locations*.

The above describes but the simplest use case of fleet inference and is rather simple. Join inference classification was robust until a third table was added, after which it stumbled on too many primary–foreign key relations and couldn't accurately classify INDs from FKCs. Upon creating a third table, attribute classification did not yield any false positives but no additional true negatives were found despite adding a corresponding *capacity* column for the third task table.

It is apparent that fleet inference shows promise, but it falls short on its robustness. In the upcoming section we will discuss the shortcomings and strengths from a more holistic perspective: while these comparisons provide great insight in the defects and strengths of join inference, I maintain it not be the final word, as the implementation of fleet inference should be analysed as a whole.

"When Rutherford showed that atoms were mostly empty space, did the ground become any less solid?"

– Greg Egan, *Quarantine* (1992)

"One should never mistake pattern. . . for meaning."

– Iain M. Banks, *The Hydrogen Sonata (2012)*

5 Conclusion

This thesis presented a way to automatically discover references between a set of column-oriented documents, and a way to automatically recognize the nature of each column within those documents to match a particular target domain. In this case the domain was the vehicle routing problem, for which a certain set of columns is required, e.g., vehicle capacities and task locations. The first part, the discovery of references, is called join inference; the second part, the discovery of attribute natures, is called attribute classification.

The two-part method was implemented using machine learning classifiers. The idea of the presented solution is to train an algorithm to recognize references between or how to recognize attributes from a certain domain. The first part is called join inference was mostly implemented using knowledge gained from existing literature but for the latter knowledge was scant and thus the method was implemented with a similar approach to the first.

What stands out from this thesis is the proper characterisation of the fleet inference problem and its subproblems, notably that of join inference. The formulation was built from scratch by assimilating pieces of existing formulations, but the thesis provides, for the first time, a comprehensive characterisation of the join inference problem, using basic relational algebra as a foundation.

The performance of the join inference portion did not measure to existing research. There are multiple possible culprits for this. The first is the selection of proper training data, as machine learning algorithms are only as good as their training data sets. I showed that in comparison to existing methods of join inference, my presented solution did not attain a similar level of accuracy. For attribute classification such a measurement was not possible to conduct because I found no existing work with that particular application.

As a whole, however, the merits of this work as evidenced by the results show promise. The principal point I will state as the contribution of this thesis is this:

using fleet inference for speeding up VRP data processing works. By improving the algorithms, choosing the right approach—machine learning or heuristic—will almost certainly make the process more accurate. Building on this work will produce an accurate data integration system for optimisation systems—the data need not be for VRP. Theoretically, any data domain model can be constructed, as long as the source data is somehow relational.

As such, the presented work will hopefully serve as a basis for implementing future data processing elements inside optimisation systems. The important caveats identified at the end of Chapter 4, to mention a few, a proper training of the algorithms and selecting quality data sets, should be taken into account when building such a system based on machine learning.

When it comes to implementation, it would seem that, based on literature, in some cases it is more viable to not use machine learning methods for implementing join inference. We saw that some existing algorithms were capable of solving the problem heuristically with near-perfect accuracy. However, the strengths of machine learning lie in its robust nature and ability to produce heuristics from training data.

Another factor that had an effect on the quality of the presented solution was its considerable breadth and difficulty. As far as I know, there have been no attempts at solving data importing issues in optimisation softwares in this manner, hence much of the theoretical work had to be assembled from existing pieces. Obtaining real-world testing sets for attribute classification proved to be impractical. However, the theoretical foundation, along with the unpolished implementation, are the main contributions of this thesis.

The future development would undoubtedly consist of amending the implementation, e.g., by improving its training algorithms, or to consider changing its methodology from learning-based to heuristic, developing tests to determine the robustness of said algorithms and creating ways of generating synthetic testing sets for the attribute classification algorithms. Obtaining working data sets from the target domain, and showing how to integrate it into an existing optimisation framework, would definitely provide insight on whether the solution is able to solve the problem effectively, and would definitely be required to ensure the capabilities of the proposed data import scheme.

To conclude, the presented work is a fundamentally a *proof-of-concept*. Now a possibility exists for operations researchers to spend less time in curating data in optimisation scenarios. This work lays the groundwork for inferring the optimisation model. When data parsing and problem modelling are unified into one, an immense amount of time could be saved if optimisation models, i.e., the problem types and scenarios, could be recognized from the data. Instead of carefully constructing complex formulas, constraints could be inferred from the data, and the problem type could be *recognized* from source data. The first step of recognizing source data is understanding its structure and purpose, and to this end, we present fleet inference.

Bibliography

Acar, A. C., and A. Motro. 2008. "Query Consolidation: Interpreting a Set of Independent Queries Using a Multidatabase Architecture in the Reverse Direction". In *Proc. of the International Workshop on New Trends in Information Integration, VLDB'09,* 4–7.

———. 2009. "Efficient discovery of join plans in schemaless data". In *Proceedings of the 2009 International Database Engineering & Applications Symposium,* 1–11. IDEAS '09. New York, NY, USA: ACM. ISBN: 978-1-60558-402-7.

———. 2010. "Inferring user goals from sets of independent queries in a multidatabase environment". *Advances in Intelligent Information Systems:*225–243.

Aho, A. V., C. Beeri, and J. D. Ullman. 1979. "The theory of joins in relational databases". *ACM Trans. Database Syst.* 4 (3): 297–314. ISSN: 0362-5915.

Andritsos, P., P. Tsaparas, R. Miller, and K. Sevcik. 2004. "LIMBO: Scalable clustering of categorical data". *Advances in Database Technology-EDBT 2004:*531–532.

Bartlett, P. L., and S. Mendelson. 2003. "Rademacher and gaussian complexities: risk bounds and structural results". *J. Mach. Learn. Res.* 3 (): 463–482. ISSN: 1532-4435.

Bauckmann, J., U. Leser, and F. Naumann. 2006. "Efficiently computing inclusion dependencies for schema discovery". In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on,* 2–2.

Bellahsene, Z. 2011. *Schema Matching and Mapping.* Springer.

Berlin, J., and A. Motro. 2002. "Database Schema Matching Using Machine Learning with Feature Selection". In *Advanced Information Systems Engineering,* 452–466. Lecture Notes in Computer Science. Springer Berlin Heidelberg. ISBN: 978-3-540-43738-3, 978-3-540-47961-1.

Bernstein, P. A. 1976. "Synthesizing third normal form relations from functional dependencies". *ACM Trans. Database Syst.* 1 (4): 277–298. ISSN: 0362-5915.

Bernstein, P. A. 2003. ”Applying model management to classical meta data problems”.

Bernstein, P. A., J. Madhavan, and E. Rahm. 2011. ”Generic schema matching, ten years later”. *Proceedings of the VLDB Endowment* 4 (11): 695–701.

Bishop, C. 2006. *Pattern Recognition and Machine Learning.* Springer. ISBN: 978-0-387-31073-2.

Blum, A. L., and Pat Langley. 1997. ”Selection of relevant features and examples in machine learning”. *Artificial Intelligence* 97 (1–2): 245–271. ISSN: 0004-3702.

Breiman, L. 2001. ”Random forests”. *Machine learning* 45 (1): 5–32.

Breiman, L., J. Friedman, R. Ohlsen, and C. Stone. 1984. *Classification and regression trees.* Wadsworth International Group.

Brown, P. G., and P. J. Hass. 2003. ”BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data”. In *Proceedings of the 29th international conference on Very large data bases-Volume 29,* 668–679.

Calì, A., D. Calvanese, G. De Giacomo, and M. Lenzerini. 2006. ”Data Integration Under Integrity Constraints”. In *Advanced Information Systems Engineering,* edited by A. B. Pidduck, M. T. Ozsu, J. Mylopoulos, and C. C. Woo, 262–279. Lecture Notes in Computer Science. Springer Berlin Heidelberg. ISBN: 978-3-540-43738-3, 978-3-540-47961-1.

Cantador, I., P. Brusilovsky, and T. Kuflik. 2011. ”2nd Workshop on Information Heterogeneity and Fusion in Recommender Systems (HetRec 2011)”. In *Proceedings of the 5th ACM conference on Recommender systems.* RecSys 2011. New York, NY, USA: ACM.

Casanova, M. A., R. Fagin, and C. H. Papadimitriou. 1984. ”Inclusion dependencies and their interaction with functional dependencies”. *Journal of Computer and System Sciences* 28 (1): 29–59. ISSN: 0022-0000.

Christofides, N., A. Mingozzi, and P. Toth. 1979. ”The vehicle routing problem”. *Combinatorial optimization* 11:315–338.

Conte, L. L., S. E. Brenner, T. J. P. Hubbard, C. Chothia, and A. Murzin. 2002. "SCOP database in 2002: refinements accommodate structural genomics". *Nucleic acids research* 30 (1): 264–267. Visited on August 2, 2013.

Damerau, F. J. 1964. "A technique for computer detection and correction of spelling errors". *Communications of the ACM* 7 (3): 171–176.

Dantzig, G. B., and J. H. Ramser. 1959. "The truck dispatching problem". *Management science* 6 (1): 80–91.

De Marchi, F., S. Lopes, and J. M. Petit. 2002. "Efficient algorithms for mining inclusion dependencies". *Advances in Database Technology—EDBT 2002:*199–214.

———. 2009. "Unary and n-ary inclusion dependency discovery in relational databases". *Journal of Intelligent Information Systems* 32 (1): 53–73.

Deutsch, A. 2009. "FOL modeling of integrity constraints (dependencies)". *Encyclopedia of Database Systems:*1155–1161.

Doan, A., P. Domingos, and A. Y. Halevy. 2001. "Reconciling schemas of disparate data sources: a machine-learning approach". *SIGMOD Rec.* 30 (2): 509–520. ISSN: 0163-5808.

Drexl, Michael. 2012. "Rich vehicle routing in theory and practice". *Logistics Research* 5 (1-2): 47–63. ISSN: 1865-035X, 1865-0368.

Fagin, R., P. G. Kolaitis, R. J. Miller, and L. Popa. 2005. "Data exchange: semantics and query answering". *Theoretical Computer Science* 336 (1): 89–124. ISSN: 0304-3975.

Freund, Y., and R. E. Schapire. 1995. "A desicion-theoretic generalization of on-line learning and an application to boosting". In *Computational Learning Theory,* edited by P. Vitányi, 23–37. Lecture Notes in Computer Science. Springer Berlin Heidelberg. ISBN: 978-3-540-59119-1, 978-3-540-49195-8.

Friedman, J. H. 2001. "Greedy function approximation: a gradient boosting machine". *Annals of Statistics:*1189–1232.

Friedman, J. H. 2002. ″Stochastic gradient boosting″. *Computational Statistics & Data Analysis* 38 (4): 367–378.

Geurts, P., D. Ernst, and L. Wehenkel. 2006. ″Extremely randomized trees″. *Machine learning* 63 (1): 3–42.

Guyon, I., and A. Elisseeff. 2003. ″An introduction to variable and feature selection″. *The Journal of Machine Learning Research* 3:1157–1182.

Hastie, T., R. Tibshirani, and J. Friedman. 2001. *The elements of statistical learning: data mining, inference, and prediction.* Springer New York.

Hristidis, V., and Y. Papakonstantinou. 2002. ″Discover: keyword search in relational databases″. In *Proceedings of the 28th international conference on Very Large Data Bases,* 670–681. VLDB ′02. VLDB Endowment.

Huhtala, Y., J. Kärkkäinen, P. Porkka, and H. Toivonen. 1999. ″TANE: An efficient algorithm for discovering functional and approximate dependencies″. *The Computer Journal* 42 (2): 100–111.

Hunter, J. D. 2007. ″Matplotlib: A 2D graphics environment″. *Computing in Science & Engineering:*90–95.

Ilyas, I. F., V. Markl, P. Haas, P. Brown, and A. Aboulnaga. 2004. ″CORDS: automatic discovery of correlations and soft functional dependencies″. In *International Conference on Management of Data: Proceedings of the 2004 ACM SIGMOD international conference on Management of data,* 13:647–658.

Jones, E., T. Oliphant, and P. Peterson. 2001. *SciPy: Open source scientific tools for Python.*

Kang, J., and J. Naughton. 2003. ″On schema matching with opaque column names and data values″. In *International Conference on Management of Data: Proceedings of the 2003 ACM SIGMOD international conference on Management of data,* 9:205–216.

Kantola, M., H. Mannila, K.-J. Räihä, and H. Siirtola. 1992. ″Discovering functional and inclusion dependencies in relational databases″. *International Journal of Intelligent Systems* 7 (7): 591–607. ISSN: 1098-111X.

King, R. S., and J. J. Legendre. 2003. "Discovery of Functional and Approximate Functional Dependencies in Relational Databases". *Journal of Applied Mathematics and Decision Sciences* 7 (1): 49–59. ISSN: 1173-9126.

Kolaitis, P. G. 2005. "Schema mappings, data exchange, and metadata management". In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems,* 61–75. PODS '05. New York, NY, USA: ACM. ISBN: 1-59593-062-0.

Koltchinskii, V. 2001. "Rademacher penalties and structural risk minimization". *Information Theory, IEEE Transactions on* 47 (5): 1902–1914.

Kullback, S., and R. A. Leibler. 1951. "On information and sufficiency". *The Annals of Mathematical Statistics* 22 (1): 79–86.

Lenstra, J. K., and A. H. G. Kan. 1981. "Complexity of vehicle routing and scheduling problems". *Networks* 11 (2): 221–227.

Lenzerini, M. 2002. "Data integration: A theoretical perspective". In *Symposium on Principles of Database Systems: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems,* 3:233–246.

Levenshtein, V. 1966. "Binary codes capable of correcting deletions, insertions and reversals". In *Soviet physics doklady,* 10:707.

Li, W.-S., and C. Clifton. 1993. "Using field specifications to determine attribute equivalence in heterogeneous databases". In *, Third International Workshop on Research Issues in Data Engineering, 1993: Interoperability in Multidatabase Systems, 1993. Proceedings RIDE-IMS '93,* 174–177.

———. 1994. "Semantic integration in heterogeneous databases using neural networks". In *Proceedings of the International Conference on Very Large Data Bases,* 1–1.

———. 2000. "SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks". *Data & Knowledge Engineering* 33 (1): 49–84.

Lopes, S., J.-M. Petit, and F. Toumani. 2002. ″Discovering interesting inclusion dependencies: application to logical database tuning″. *Information Systems* 27 (1): 1–19. ISSN: 0306-4379.

Maier, D., and J. D. Ullman. 1983. ″Maximal objects and the semantics of universal relation databases″. *ACM Transactions on Database Systems (TODS)* 8 (1): 1–14.

Maier, D., J.D. Ullman, and M. Y. Vardi. 1984. ″On the foundations of the universal relation model″. *ACM Transactions on Database Systems (TODS)* 9 (2): 283–308.

McKinney, W. 2011. *pandas: Powerful Python Data Analysis Toolkit.*

Meinshausen, N., and P. Bühlmann. 2010. ″Stability selection″. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 72 (4): 417–473.

Mell, P., and T. Grance. 2009. ″The NIST definition of cloud computing″. *National Institute of Standards and Technology* 53 (6): 50.

Mitchell, T. M. 1997. *Machine Learning.* Boston: McGraw-Hill.

Mohri, M., A. Rostamizadeh, and A. Talwalkar. 2012. *Foundations of Machine Learning.* Cambridge, Massachusetts: The MIT Press.

Naumann, F., C. T. Ho, X. Tian, L. Haas, and N. Megiddo. 2002. ″Attribute classification using feature analysis″. In *Proceedings of the International Conference on Data Engineering,* 271–271.

Oliphant, T. E. 2007. ″Python for scientific computing″. *Computing in Science & Engineering* 9 (3): 10–20.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. ″Scikit-learn: Machine Learning in Python″. *Journal of Machine Learning Research* 12:2825–2830.

Perez, F., and B. E. Granger. 2007. ″IPython: a system for interactive scientific computing″. *Computing in Science & Engineering* 9 (3): 21–29.

Perrey, R., and M. Lycett. 2003. ″Service-oriented architecture″. In *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings,* 116–119.

Puranen, T. 2011. ”Metaheuristics meet metamodels: a modeling language and a product line architecture for route optimization systems”. PhD thesis, University of Jyväskylä.

Quinlan, J. R. 1986. ”Induction of decision trees”. *Machine learning* 1 (1): 81–106.

———. 1993. *C4.5: Programs for Machine Learning.* Morgan Kaufmann. ISBN: 978-1-558-60238-0.

Rahm, E., and P. A. Bernstein. 2001. ”A survey of approaches to automatic schema matching”. *The VLDB Journal* 10 (4): 334–350. ISSN: 1066-8888, 0949-877X, visited on June 9, 2014.

Rostin, A., O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. 2009. ”A machine learning approach to foreign key discovery”. In *12th International Workshop on the Web and Databases (WebDB).*

Schapire, R., and Y. Freund. 2012. *Boosting: Foundations and Algorithms.* MIT Press.

Sekhavat, Y.A., and J. Parsons. 2012. ”Semantic Schema Mapping Using Property Precedence Relations”. In *2012 IEEE Sixth International Conference on Semantic Computing (ICSC),* 210–217.

Silberschatz, A. 2006. *Database system concepts.* 5th ed. Boston: McGraw-Hill. ISBN: 0-07-295886-3.

Sra, S., S. Agarwal, and S. J. Wright. 2012. *Optimization for Machine Learning.* MIT Press.

Toth, P., and D. Vigo, editors. 2002. *The Vehicle Routing Problem.* SIAM.

Transaction Processing Performance Council. 2013. ”TPC Benchmark H”.

Valiant, L. G. 1984. ”A Theory of the Learnable”. *Commun. ACM* 27 (11): 1134–1142. ISSN: 0001-0782.

Vapnik, V. 2006. *Estimation of dependences based on empirical data.* Springer.

Vapnik, Vladimir N., and A. Chervonenkis. 1971. "On the uniform convergence of relative frequencies of events to their probabilities". *Theory of Probability & Its Applications* 16 (2): 264–280.

Wand, Y., and R. Weber. 1990. "Mario Bunge's Ontology as a formal foundation for information systems concepts". *Studies on Mario Bunge's Treatise, Rodopi, Atlanta:*123–149.

Zhang, M., M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. 2010. "On Multi-Column Foreign Key Discovery". *Proceedings of the VLDB Endowment* 3 (1-2): 805–814.

– Groucho Marx