# Automated Feature Engineering on Relational Data

*Dimitrios Kalouris*

Thesis submitted in partial fulfillment of the requirements for the

*Masters' of Science degree in Computer Science and Engineering*

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor:
Prof. *Ioannis Tsamardinos*, Prof. *Vassilis Christophedes*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

The research work was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "First Call for H.F.R.I. Research Projects to support Faculty members and Researchers and the procurement of high-cost research equipment grant" (Project Number: 1941).

University of Crete
Computer Science Department

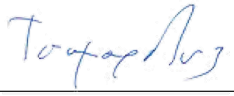
**Title**

Thesis submitted by
**Dimitrios Kalouris**
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: 

Dimitrios Kalouris

Committee approvals: 

Ioannis Tsamardinos
Professor, Thesis Supervisor, University of Crete, Greece

Vassilis Christophides
Professor, Thesis Supervisor, University of Crete, Greece

Nikos Komodakis
Assistant Professor, Committee Member, University of Crete

Departmental approval:

Polyvios Pratikakis
Associate Professor, Director of Graduate Studies

Heraklion, February 2022

# Automated Feature Engineering on Relational Data


## Abstract

Machine learning typically learns from a single table. However, in the age of big data it is often the case that data are distributed across many different tables in a relational database for efficiency. To work with relational data it is not rare for scientists perform feature engineering manually and intuitively. Additionally, many algorithms that produce a single table from a relational database have been proposed for this problem but none of them takes into account complex relational data schemas or they are limited in the paths they follow and the combinations of joins and aggregations they perform during feature generation. Moreover these algorithms, during feature generation, accumulate large number of features before performing feature selection and the feature selection algorithms are not optimized. To this end we created SRFGA a novel online feature engineering algorithm that performs joins and aggregations on the tables to create features and keeps only the most useful features, using the residuals calculated by a model to guide the feature selection. This algorithm can be used without any knowledge expertise, and it also unifies all the previous works in terms of visited paths and actions performed.

# Αυτοματοποιημένη Κατασκευή Χαρακτηριστικών σε Σχεσιακά Δεδομένα

## Περίληψη

Στην Μηχανική Μάθηση συχνά μαθαίνουμε μοντέλα από ένα μοναδικό πίνακα. Όμως, στην εποχή των μεγάλων δεδομένων στην οποία ζούμε πολύ συχνά τα δεδομένα είναι μοιρασμένα σε πολλούς διαφορετικούς πίνακες μέσα σε μια βάση για καλύτερη αποδοτικότητα. Για να δουλέψουν με τα σχεσιακά αυτά δεδομένα δεν είναι σπάνιο οι επιστήμονες να δημιουργούν ενα-ενα τα χαρακτηριστικά σε μια διαδικασία που είναι κυρίως ενστικτώδης και απαιτεί γνώσεις πεδίου. Στόχος είναι η δημιουργία ενός Αυτοματοποιημένου Αλγορίθμου Κατασκευής Χαρακτηριστικών, τον οποίο οποιασδήποτε χωρίς ειδικές γνώσεις μπορεί να χρησιμοποιήσει. Πολλοί αλγόριθμοι έχουν προταθεί που μετατρέπουν μια βάση από πολλούς πίνακες σε έναν, αλλά κανείς από αυτούς δεν λαμβάνει υπόψη τα πιο περίπλοκα σχήματα βάσεων. Επιπλέον αυτοί οι αλγόριθμοι, κατά την παραγωγή των χαρακτηριστικών, συσσωρεύουν ένα μεγάλο πλήθος από χαρακτηριστικά πριν εκτελέσουν κάποιον αλγόριθμο επιλογής χαρακτηριστικών ενώ ακόμη οι αλγόριθμοι επιλογής που χρησιμοποιούν δεν είναι βελτιστοποιημένοι. Για αυτό και εμείς δημιουργήσαμε έναν καινοφανή αλγόριθμο κατασκευής χαρακτηριστικών ο οποίος εκτελεί ενώσεις και αθροιστικές συναρτήσεις στους πίνακες για να παράξει χαρακτηριστικά, ενώ κρατάει μόνο τα πιο χρήσιμα από αυτά μέσα από ένα μοντέλο υπόλοιπα που υπολογίζει υπόλοιπα. Τέλος προτείνουμε έναν αλγόριθμο επιλογής χαρακτηριστικών που κλιμακώνει σε μεγάλου όγκου δεδομένα, ο οποίος μπορεί να βλέπει έναν πίνακα σε κομμάτια και μετά να αθροίσει την πληροφορία χωρίς μεγάλο κόστος στην απόδοση.

# Ευχαριστίες

Επειδή μια επιστημονική μελέτη δεν μπορεί να πραγματοποιηθεί μόνο από ένα άτομο, αισθάνομαι την ανάγκη να αναφερθώ και να ευχαριστήσω όλους όσους συνέβαλαν με τον τρόπο τους να ολοκληρωθεί η διπλωματική μου εργασία και στήριξαν αυτή μου την προσπάθεια.

Αρχικά θέλω να ευχαριστήσω το Πανεπηστήμιο Κρήτης και το Τμήμα Επιστήμης Υπολογιστών που με το προγραμμα μεταπτυχιακών σπουδών μου έδωσε αυτή την ευκαιρία και μου άνοιξε νέους δρόμους σκέψης και γνώσης.

Θελω να ευχαριστήσω την καθηγήτρια Γεωργία Καραλή οπου οι συζητήσεις μαζί της και οι συμβουλές της με ώθησαν αρχικά στο Τμήμα Επιστήμης Υπολογιστών, καθώς χωρίς αυτήν δεν θα είχα την ευκαιρία να ζήσω όλη αυτή την εμπειρία.

Θελω να ευχαριστήσω τους επόπτες καθηγητες μου Ιωάννη Τσαμαρδίνο και Βασίλη Χριστοφίδη για την βοήθεια και την καθοδήγηση που μου προσέφεραν σε όλη την δουλεία μου καθως και για την απεριόριστη υπομονή και κατανόηση που έδειξαν στις δύσκολες φάσεις της.

Ευχαριστώ ακόμα, τα μέλη του ερευνητικού εργαστηρίου mens X machina για τις γνώσεις και συζητήσεις που μοιραστήκαμε.

Τέλος θελω να ευχαριστήσω θερμά τους φιλους μου και γονεις μου Αντώνη και Ευαγγελία και τον αδερφο μου Ευάγγελο για την αμέτρητη αγάπη και συμπαράσταση που μου πρόσφεραν σε όλη αυτήν την προσπάθεια.

*Στους γονείς μου και τον αδερφό μου*

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

Over the last decade, data analytics has become an important trend in many industries including e-commerce, healthcare, entertainment and more. The reasons behind this are the availability of data, variety of open-source machine learning tools and powerful computing resources. Fig 1.1 shows the basic steps in a data analysis project. Although many state of the art algorithms exist for the Modelling step, most of them assume that the data are in batch form, inside a table. The problem with this is that most enterprises use databases to store their data across many different tables. This is done for efficiency and speed and also because data may come from many different sources, and databases make integration easier.

**Example 1.1.1** (Running example database)**.** The database example we will use follows the PKDD'99 Discovery challenge dataset. Financial database describes real anonymized bank data from a Czech bank including transactions, loans, accounts and other info. The original database contains 8 tables but we will use only 4 for simplicity. The tables we will use are CLIENT which describes characteristics of clients, DISTRICT that describes demographic info about each district, ACCOUNT that describes characteristics of individual accounts and DISP (disposition) which relates clients to accounts and describes the rights each client has to an account. The predictive task we will aim to solve is deciding the Gender of the Client. In this sense the Base Table of interest is Client and Figure 1.3 shows a sample of the Client relation. Finally in Figure 1.2 we can see the running example database.

A relational database schema consists of the tables (or relations) inside, the attributes they contain and the relationships between the tables which are denoted by arrows. Each relation has a unique set of attributes that uniquely identifies every row called Primary Key (PK). Relations can also reference the PKs of other relation by Foreign Key (FK) attributes. In general a relationship between two tables has a direction, denotes that they have a common attribute and that one


Figure 1.1: Main steps involved in data analysis problems.

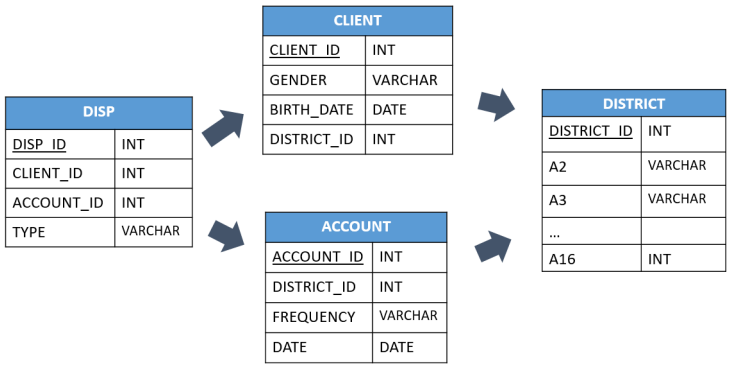



Figure 1.2: Relational Schema of the running example database. This database describes a subset of the relations that appear in the PKDD'99 Financial database challenge. Each table describes a relation, with the relation name on the top and then the attributes with their types. The primary key attribute(s) of each relation have their attribute names underlined.

table's attribute references the attribute of the other table. For example ACCOUNT.DISTRICT_ID references DISTRICT.DISTRICT_ID. In this case we say the relationship is 1-many with relation to District table because more than one accounts can share the same district, and likewise it is many-1 with relation to account. If for each account we have exactly one district then we say that the relationship is 1-1 both ways but usually 1-1 relationships are simplified into one table. In Fig 1.5 we can see that joining CLIENT and DISP tables results in increasing the rows of the CLIENT table as some DISP records share the same client. We will only focus on inner joins so tuples that dont exist for both tables ( Client_id=4 or 5) will be removed. What we want is to have one unique row per client so we group the records that have the same Client_id and then perform aggregation functions to summarize many values into one. Figure 1.6 shows the results of aggregating by Client_ID and performing the MAX, MIN, SUM aggregations on some of the attributes.

In order for the batch machine learning algorithms to work, the relational data, which are scattered across many tables need to be summarized in a single table which will include the attribute we want to model. The problem we are trying to solve is called Relational Feature Construction and takes as input a database schema as in Fig 1.2, a main table **S** and an attribute of interest **T** and outputs a batch dataset that includes features that we can use to model **T**. This is done by

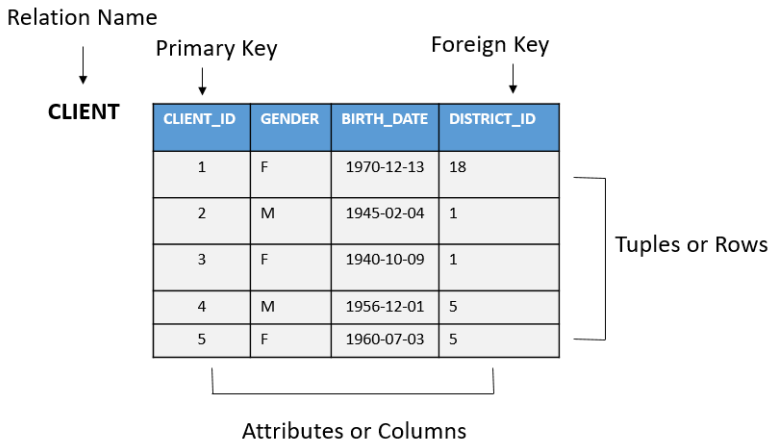



Figure 1.3: Schema describing the relation Client.

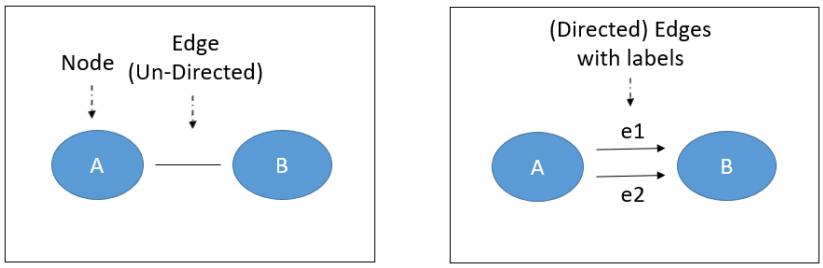
performing transformation, joins on the tables and aggregation functions. It is not rare that data analysts gather and create useful features manually. This process is apart from being manual, is highly intuitive, requires domain knowledge and takes a lot of time. In fact in previous Kaggle competitions experienced data scientists noted that feature engineering took most of their time for these tasks. This should not be the case as this data analysis should also be performed by people with no expertise in the given areas and also even for experienced users, it would be useful to have a process to try a few things quickly with low cost before diving more into the data.

To solve this many algorithms have been proposed that take a database as input and either return a batch dataset to be used for modeling (propositionalization algorithms) or return a relational model that can be applied directly on the database (relational classifiers). Propositionalization algorithms start from the base tables, perform joins and aggregations to create new features which they then store inside views. They have two main variation based on where they start producing features. The first approach we will call Forward because they start from S to reach other non-base tables. More specifically:

- start from the base table
- perform joins to reach other tables
- when they visit a 1-many edge they group-by the primary key of the base table and perform aggregations.

In our running example the feature creation order would be:

1. CLIENT
2. $CLIENT \bowtie DISP$




(a) Undirected Simple Graph

(b) Directed MultiGraph. Notice that each edge has a label.

Figure 1.4: Simple Graph and MultiGraph Example. Circles correspond to nodes and lines/arrows correspond to undirected/directed edges respectively.

3. GB CLIENT_ID ( $CLIENT \bowtie DISP$ )

where GB CLIENT_ID means group by Client ID and then perform aggregations.

On the other hand we have the Backward algorithms, which follow they backward direction starting from non-base tables to reach S. More specifically:

- start by joining non-base tables
- perform aggregations based on those tables primary keys
- only at the end join with S and group by the PK of S.

For example:

1. ACCOUNT
2. $ACCOUNT \bowtie DISTRICT$
3. GB DISTRICT_ID ($ACCOUNT \bowtie DISTRICT$ )
4. $CLIENT \bowtie$ (GB DISTRICT_ID $ACCOUNT \bowtie DISTRICT$)
5. GB CLIENT_ID($CLIENT \bowtie$ (GB DISTRICT_ID $ACCOUNT \bowtie DISTRICT$))

With the previous approaches the following problems exist:

- Relational classifiers, by using a single modeling algorithm, induce bias into the analysis and deprive the analyst of the option to try different models.
- A limitation of related work on Propositional Algorithms is that either they aggregate too early (as soon as they visit 1-many edge) for speed, or aggregate only by specific keys like on Forward that we aggregate only by PK of S which leads to missing some possible useful features as they do not try all join - groupby key combinations.

**CLIENT**

| CLIENT_ID | GENDER | BIRTH_DATE | DISTRICT_ID |
|---|---|---|---|
| 1 | F | 1970-12-13 | 18 |
| 2 | M | 1945-02-04 | 1 |
| 3 | F | 1940-10-09 | 1 |
| 4 | M | 1956-12-01 | 5 |
| 5 | F | 1960-07-03 | 5 |

**DISP**

| DISP_ID | CLIENT_ID | ACCOUNT_ID | TYPE |
|---|---|---|---|
| 1 | 1 | 23 | 1 |
| 2 | 1 | 2 | 3 |
| 3 | 2 | 92 | 1 |
| 4 | 2 | 2 | 3 |
| 5 | 3 | 5 | 3 |

**CLIENT x DISP**

| CLIENT_ID | GENDER | BIRTH_DATE | DISTRICT_ID | DISP_ID | ACCOUNT_ID | TYPE |
|---|---|---|---|---|---|---|
| 1 | F | 1970-12-13 | 18 | 1 | 23 | 1 |
| 1 | F | 1970-12-13 | 18 | 2 | 2 | 3 |
| 2 | M | 1945-02-04 | 1 | 3 | 92 | 1 |
| 2 | M | 1945-02-04 | 1 | 4 | 2 | 3 |
| 3 | F | 1940-10-09 | 1 | 5 | 5 | 3 |
| 4 | M | 1956-12-01 | 5 | - | - | - |
| 5 | F | 1960-07-03 | 5 | - | - | - |

Figure 1.5: CLIENT has a 1-many relationship with DISP so when we create CLIENT x DISP ( here we used outer join) for some clients we have more than 1 records from DISP.

- While Forward and Backward approaches have some common features, as the depth of the graph increases (depth>2) they start to differ , but not one of these approaches covers the other.
- To the best of our knowledge algorithms in related work track duplicate features using a list of visited tables. This of course fails to address cases when we have more than 1 edge between two tables or we have circles as there we can have tables occurring more than once.
- Feature Selection is done at the end of feature generation and does not account for correlation or redundancy between the features.

## 1.2 Thesis Contributions

The purpose of this work is introduce Supervised Relational Feature Generator Algorithm (SRFGA), a novel automated feature generation framework on databases that fully captures all the information available even in complex database schemas; that also encapsulates a highly scalable feature selection algorithm to make the memory impact of the algorithm minimal. The contributions of this work are:

| CLIENT_ID | MAX( TYPE) | MIN(DISP_ID) | SUM(ACCOUNT_ID) |
|---|---|---|---|
| 1 | 3 | 1 | 25 |
| 2 | 3 | 3 | 94 |
| 3 | 3 | 5 | 5 |

Figure 1.6: Resulting table from grouping Client⋈Disp by Client ID and performing aggregations MAX, MIN, SOME on a subset of the columns.

- We translate the relational feature generation problem into a Graph Search problem by encoding all the necessary information for feature generation inside search nodes. The actions we selected are Join and Aggregate but easily more actions can be inserted. By using this abstraction we can then navigate the relational schema graph and produce all combinations of join and group-by operations and thus encompassing both the previous approaches. We show that SRGFA was on par or outperformed state of the art algorithms in terms of AUC performance in all the real database experiments.

- We demonstrate the usefulness of the produced features by showing that in all real worth datasets SRGFA produced features that led to a significant increase in performance, +20% AUC on average, in comparison to using only the information from the base table.

- We developed the algorithm to work with complex graphs instead of only graphs that have single edges like DFS does. We show that in some case this leads to an improvement of 30% in AUC of the predictions as the SRGFA is able to capture most of the important information from the database.

- We use an online version of an off-self feature selection algorithm and show that this algorithm can retain performance as if we had seen all features at once. Using this algorithm we only keep the most important features at each step, thus trading computer memory for execution time which is needed in order to scale to big databases where not all features can fit into memory at once.

## 1.3 Organization of the chapters

- In chapter 2: We present the Notation and Background Knowledge needed for this thesis.

- In chapter 3: Describes our work Supervised Relational Feature Generator Algorithm

- In chapter 4: Shows the experiments and comparison with related work

- In chapter 5: Presents the Related work.
- In chapter 6: We finalize with the discussion and future work.

# Chapter 2

# Background & Notation

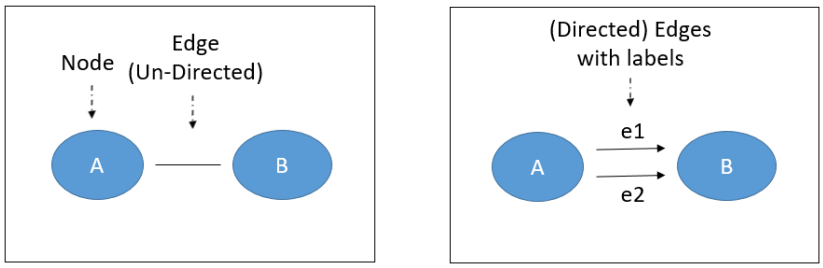
In this section we will introduce the basic background knowledge and notation that will be needed to follow the rest of the thesis. The algorithm that we will use takes us input a relational database and returns a set of features after performing an intelligent search. The schema of the database follows a graph so in Section 2.1 we introduce the basic notions of a Multi-Graph and graph paths. Section 2.2 describes relational data, how it is stored in the database, what operators can be performed by a Relational Database Management System and introduces a running example database. Finally Section 2.3 describes the notions of intelligent agents and Search problems and introduces and Search problem in the context of relational data and our running example database.

## 2.1 Graphs and MultiGraphs

Graphs or Simple Graphs are data structures that are used to model relationships between objects. These objects are called nodes or vertices and the connections between them are called edges or links. Graphs are can either have directed or undirected edges, called directed or undirected graphs respectively, based on if we are interested to model the direction of the relationship or only its existence.

**Definition 2.1.1** (Simple Graph)**.** A Simple Graph or Graph G can be defined as $G = (V, E)$ where V is the set of vertices and $E \subset \{(x, y) \in V^2, x \neq y\}$ the set of edges which can be unordered or ordered based on the type of graph edges. If we also want to allow loops which is a vertice connected with itself we remove the $x \neq y$ restriction from the definition of E[4].

Figure 2.1(a) shows an example of an Un-Directed Simple Graph, if the edge had a direction, for example from A to B, then the graph would be directed. Because graphs can only model the relationship between two vertices, if we are interested in more than one edge between vertices MultiGraphs are needed which are a generalization of Graphs.




(a) Undirected Simple Graph

(b) Directed MultiGraph. Notice that each edge has a label.

Figure 2.1: Simple Graph and MultiGraph Example. Circles correspond to nodes and lines/arrows correspond to undirected/directed edges respectively.

**Definition 2.1.2** (Multi Graph)**.** MultiGraphs are defined as $G = (V, E, \phi)$ where V is the set of vertices, E is a finite set of edges and $\phi$ is a function $\phi : E \rightarrow V^2$ that maps edges to pairs of vertices[4]..

In this notation the edges E can be though of as extra labels or enumeration upon the connections so that if we have two connections between two vertices we can differentiate them by the labels. In Figure 2.1(b) we can see an example of a Directed Multi-Graphs, notice that here in comparison with the Simple graph because we have two edges connecting nodes A and B, we have labels to discriminate them.

**Definition 2.1.3** (Multi Graph Path)**.** Let $G = (V, E, \phi)$ be a MultiGraph and $e_1e_2...e_{N-1}$ be a sequence of edges in E and $v_1, v_2, ..., v_N$ elements of N such that $\phi(e_i) = (v_i, v_{i+1}), i = 1, .., N-1$. The sequence $e_1e_2...e_{N-1}$ is called a path P in G and $v_1, v_2, ..., v_N$ is called the vertex sequence of the path.

If we have a path P with vertex sequence $v_1, v_2, ..., v_N$ then we say that P is a path from $v_1$ to $v_N$. We say that a path P contains a cycle if $\exists v_1, v_2 : v_1 = v_2$ inside the vertex sequence of P.

**Definition 2.1.4** (SubGraph)**.** Let $G = (V, E, \phi)$ be a MultiGraph then a graph $G' = (V', E', \phi')$ is a subgraph of G if $V \subseteq V', E \subseteq E'$ and $\phi$ is the restriction of $\phi'$ on $E'$.

## 2.2 Relational Data

### 2.2.1 Relational Schema and Relations

We assume that the data are stored in relations $\mathcal{R}$:$R_1, R_2, ..., R_N$. Each attribute of a relation Aj $\in \mathcal{A}(Ri)$) has a name and domain (type) written as dom(Aj) from where its values are drawn from. The schema of each relation is uniquely identified

by the relation name and the set of attributes it contains $(R_i, \mathcal{A}(Ri))$ i=0,..,N. We will assume that the order of attributes does not matter. We denote with $A \subseteq \mathcal{A}(Ri))$ the set of attributes of a relation $Ri$. To refer to a subset of attributes $A$ by the notation $Ri[A]$ or simply $Ri.Aj$. A row of a table is mapping from the attributes to their domains. A row is called a record or a tuple and to reference a tuple in Ri we use the letter $t$ for example $t \in Ri$ to denote it belongs in Ri and $t[A]$ to refer to specific attributes $A \subseteq \mathcal{A}(Ri)$ of the tuple [1].

### 2.2.2 Relational Constraints and Dependencies

The tuples of the relations sometimes have to meet specific constraints. Functional Dependencies are a type of constraint on a relation Ri between two sets of attributes X,Y $\subseteq \mathcal{A}(Ri)$ denoted as X $\rightarrow$ Y which states that for t1 ,t2 in Ri: $(t1[X] = t2[X]) \Rightarrow (t1[Y] = t1[Y])$. More specifically the values of X uniquely identify the values of Y.
A primary key constraint for a relation Ri denoted as PK(Ri) is a minimal set of attributes so that PK(Ri) $\subseteq \mathcal{A}(Ri))$ and PK(Ri) $\rightarrow \mathcal{A}(Ri)$. Note that the minimal set is not unique and the solutions to this problem are called candidate keys from which we choose one as the primary key of the relation. For the Client relation Client_ID is the PK and it uniquely identifies every row. Lastly PK values can also be used to reference a specific tuple from other relations.
Foreign key constraints are constraints that involve two relations, the parent relation and the referenced relation . Tuples in the referencing relation Ri have attributes called foreign key attributes (FKs(Ri)) that reference the primary key attributes PK of the referenced relation Rj (PK(Rj)). More formally a subset of attributes FK of Ri is a foreign key referencing the primary key of Rj ( PK(Rj) ) if for every ti in Ri there is a tuple tj in Rj such that ti[FK]=t2[K2]. Thus we have that $dom(Ri[FK]) \subseteq dom(PK(Rj))$ i.e. the values of the FK(Ri) either has to match an existing primary key value in PK(Rj) or be NULL. Note here that the referenced tuple does not have to be unique in the referenced relation. In Figure 1.3 attribute CLIENT.DISTRICT_ID is a FK that refers to PK(DISTRICT).

### 2.2.3 Relational Databases and the Join Graph

**Definition 2.2.1** (Relational database)**.** Let $\mathcal{R}$ denote the set of all relations $(R_i, \mathcal{A}(Ri))$ and $\mathcal{C}$ denote the set of all key-foreign key relationships which consists of tuples $(Ri, Fj, Rj, PK(Rj))$ where Fj $\in \mathcal{A}(Ri)$ refers to PK(Rj). A relational database is defined as a set of relations $\mathcal{R}$ along with their relationships $\mathcal{C}$.

In Figure 1.2 where we have the running example database, we can see that the relational schema of the database can be modelled by a Multi-Graph because two tables can be connected with more than one edges. For this reason we will model it using the Join Graph which consists of the relations and their relationships but the edges are labeled based on the KFK join conditions.
These relationships are created when a table Ri references another table Rj using a

foreign key, symbolised as $Ri \rightarrow Rj$ it creates a binary key-foreign key relationship between these two tables. If $Ri \rightarrow Rj$ we say that the relationship is many-1 with relation to table Ri because many values of the foreign key can refer to a single value of PK(Rj). Likewise we say that the relationship is 1-many with relation to table Rj. If there is a 1-1 mapping from PK(Rj) to to the foreign key referencing it then we say that the relationships is 1-1 with relation to both tables.

### 2.2.4 Relational Database Management Systems (RDMS)

All RDMS use Structured Query Language (SQL) to query the data inside the database. SQL is a comprehensive database language which offers statements for data definition, querying and updating the relational database.

#### 2.2.4.1 Aggregation and Transformation Functions

Most Relational Database Management systems allow the use of statistical functions that take as input an attribute or more attributes and produce new attributes or a scalar. We will focus on transformation functions $f : \mathbb{R}^N \rightarrow \mathbb{R}^N$ that take an attribute perform a simple transformation and return an attribute of the same length e.g. $f(X) = \frac{X}{100}$ .The aggregation functions we will use are functions $f : \mathbb{R}^N \rightarrow \mathbb{R}$ that take an attribute perform an operation and return a scalar e.g. $f(X) = SUM(X)$. We will use $\mathcal{F}_{\mathcal{T}}$ and $\mathcal{F}_{\mathcal{A}}$ for the set of transformation and aggregation functions respectively.

#### 2.2.4.2 Relational Operators

We will focus on the operators (equi) Join, Select, Project and Group By.

- Select: Let Ri be a relation with attribute set $\mathcal{A}(Ri)$, Aj $\in \mathcal{A}(Ri)$, and x a value in the domain of Aj. Selecting all tuples that satisfy the condition $Ri.A = x$ written as $\sigma_{A=x}(Ri)$ which is the set: { t | t in R and $t[A] = x$ }. For example to select the only the female Clients from Fig 1.3 we use the notation $\sigma_{GENDER=F}(CLIENT)$.
- Project: Let Ri be a relation with attribute set $\mathcal{A}(Ri)$, A $\subseteq \mathcal{A}(Ri)$. Projecting on the attribute set A of Ri, written as $\pi_A(Ri)$ which is the set: { t | t has only attributes in A and $\exists$ tuple s in R s.t. $s[A] = t[A]$ }. For example to select only the Client_Id and the Gender of each Client we use $\pi_{CLIENT\_ID,GENDER}(CLIENT)$.
- (Equi) Join: The join operation takes two relations Ri, Rj and a join condition Ri.A = Rj.B and produces a single relation that is the Cartesian product of the two relations, but keeping only the tuples that satisfy the join condition, written as $Ri \bowtie_{Ri.A=Rj.B} Rj$ which is the set {t | t is a tuple with attributes

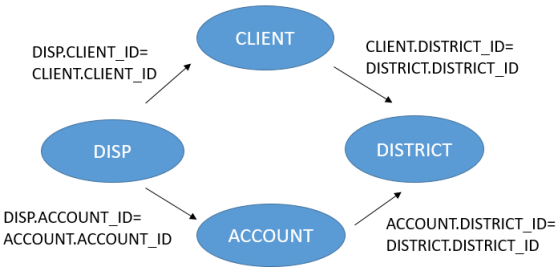



Figure 2.2: Join graph is a Multi-Graph where nodes correspond to relations and edges correspond to relational join conditions. Each edge here is labeled based on the join condition it follows.

$\mathcal{A}(Ri) \cup \mathcal{A}(Rj)$ and ∃tuples ti in Ri and tj in Rj s.t. $t[Ri] = ti[Ri]$ ,t[Rj]=tj[Rj] and ti(A)=tj(B) }. If we look at Fig 1.2 one possible (equi)join could be $CLIENT \bowtie_{CLIENT.CLIENT\_ID=DISP.CLIENT\_ID} DISP$.

- Group by: While transformation function can be used on the attributes without previous operation, aggregation functions are most useful when used on subgroups created from the original attributes. This works by partitioning the relation R into groups of tuples. Each group consists of the tuples that have the same value on some selected attribute set A, called the grouping attributes. After creating the groups we can then apply aggregation functions f∈ $\mathcal{F}_{\mathcal{A}}$ on any non-group attribute $B \subset \mathcal{A}(\mathcal{R}) \setminus A$ to produce new summary values for each group. This can be written as $\pi_{A,f(B)}(R)$ which means group by attribute set A and apply aggregation function f on the subgroups created on attribute B. For example to calculate the number of different accounts a client has we would calculate $\pi_{CLIENT\_ID,COUNT(ACCOUNT\_ID)}(R)$ where R is the relation produced by joining the Client and Disp relations as in the example above.

#### 2.2.4.3 Views and Materialized Views

**Definition 2.2.2** (View)**.** A view V is a single table that is derived from other tables which can either be the starting tables Ri or other previously created views Vi, i=1,..,N. The view does not necessarily exist in physical form inside the database, it can have the form of a virtual table. If it is in virtual form then its tuples are not stored inside the database but are retrieved when querying the view [10].

While having virtual tables saves storage space in the database, if we want to query a specific view frequently then its most efficient to create a Materialized View, which is a view that has its tuples stored inside the database.

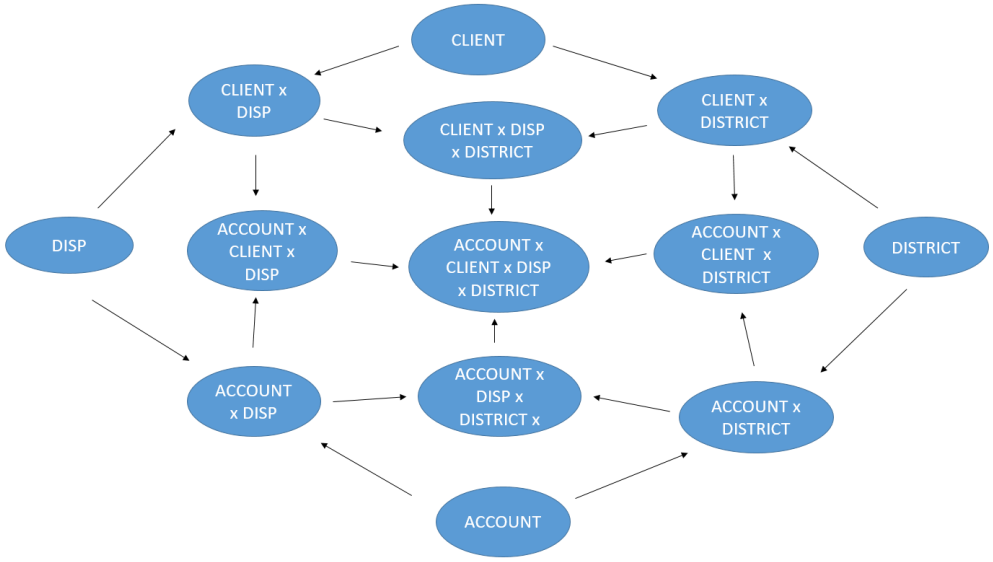



Figure 2.3: The state space created from the relations of our running example database. Here "x" denotes the join between tables and because the join operator has the associative and commutative property, the order of the joins does not matter so we name them by ordering the names. Here we restrict to joining with each table once.

## 2.3 Search Problems

### 2.3.1 Intelligent Agents

**Definition 2.3.1** (Intelligent Agent)**.** As an intelligent agent or simply agent we can define an entity that can perceive , make decisions and perform actions based on experience and perceptions [16].

We will focus only on goal-based agents which are agents that try to find a sequence of actions that lead to desirable states. The process of finding this sequence of actions is called search. Once the solution is found we can move on to the execution phase to reach the desired state. The intelligent agent in our example will be performing joins between tables by navigating the join graph in Fig 2.2.

### 2.3.2 Problem Formulation

One of the most difficult stages in solving problems by searching is turning the original vague problem into a specific stated problem that an agent can understand and solve. This process is not easy and requires removing details that do not help us in finding a solution.
A problem can be formalized by:

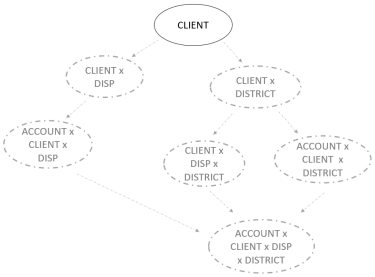
- The **Initial State** from where the agent starts. In our example this can be the state that has only Client table.
- The possible **Actions** that an agent can take in each state. In the example the only action we can perform is Join using KFK dependencies[1]. Notice that after we join two tables the resulting table has the union of attributes of the joined tables and thus has the union of their PK and FKs. As a consequence the resulting table can be seen as having all the KFK dependencies that the original tables had.
- A **Successor Function** Successor(s) that for each state s, returns all the states that are reachable from this state by a single action. For our example the Successors of a state correspond to all the results of joining the table of the current state with one of the connected tables.
- A **Goal Function** that tests whether a given state is a goal state. The goal function can either have a specific set of goal states and check if a given state is inside this set or the goal states can be described by an abstract property. An example of a Goal Function with specific states can be reaching the state ACCOUNT x CLIENT x DISP x DISTRICT. The same Goal Function can be described by the specific property is: goal state is any state where we have joined all 4 tables.
- A **Path Cost** that describes the cost of reaching a given state s. This is described by a cost function which tailored to the performance metric(s) we have chosen for the agent. For our running example the path cost can be number of joins we have performed.

The initial state, Actions and Successor function together create a **state** space of the problem which shows all states reachable by the initial state by any sequence of actions. The state space can be modeled by a directed simple Graph where nodes correspond to states and each state is connected with directed edges with its Successors. Figure 2.3 shows the State Space for the running example database. Solution(s) is any sequence of states that leads to a goal state while an optimal solution(s) is a solution with the minimum cost. In the example we have many ways to get to the goal of joining all 4 tables and if we use the number of joins as the cost, then all solutions are optimal solutions because they all have 3 steps.

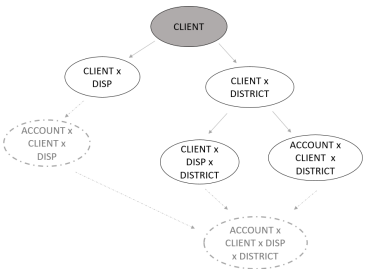
### 2.3.3 Searching For solutions

Once we have formalized the problem the final step is navigating the state space and finding the solution(s). To do this search algorithms have to take into account different action sequences and guide the search to the ones that lead to solutions.

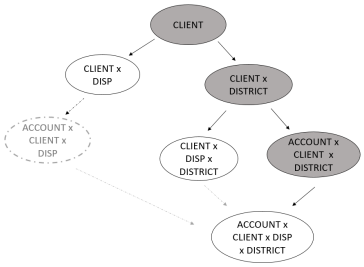
[1]When we navigate the Join graph edges either way, so they can be seen as undirected. However, the orientation of the edge will matter later when we care about the type of relationship between the tables




(a) Initial State




(b) After expanding the initial state.




(c) After completing the search.

Figure 2.4: First two Search Graphs and the final Search Graph for the running example. Nodes that have been expanded are shaded while nodes that have been generated but not yet expanded are outlined in bold. Finally nodes that have not yet been generated are shown in faint dashed lines

**Definition 2.3.2** (Search Graph)**.** The possible actions starting from the initial state create a Search Graph where nodes correspond to states in the state space and edges are actions.

As it can be seen in 2.4(a) in the root of the problem we have the initial state (here CLIENT) and then we consider various actions. Here the available actions are either joining with DISP or CLIENT. By expanding the initial state we generate the two new states respectively called child states (see 2.4(b) ). Generated notes that have not been expanded are inside the frontier set, in our example these are white node states. These states are also leaf nodes, meaning nodes without children. To continue the search we pick one of the newly generated states and repeat the same process. Exactly which one depends on the search algorithm we have chosen as we will see below[2]. An example of a complete Search Graph can be seen in 2.4(c). When and if we reach a goal state then we stop, in this case it is the last generated state. When we search for solutions it is possible that more than 1 states can lead to the same state in the example we can see that tree states lead to the final goal state (ACCOUNT x CLIENT x DISP x DISTRICT). These states are called redundant states and we can avoid them if we keep a record of all

[2]In this case we have used Depth First Search algorithm.

**Algorithm 1** Graph Search Algorithm

```
function GRAPH-SEARCH(problem) returns solution or failure
    loop do
    if frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if node contains a goal state then return corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    but only if not in the frontier or explored set
end function
```

**Algorithm 2** Depth First Search Algorithm

```
function DEPTH-FIRST-SEARCH(problem) returns solution or failure
    return RECURSIVE-DFS(MAKE-NODE(problem.INITIAL-STATE),problem)
end function

function RECURSIVE-DFS(node, problem) returns solution or failure
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else
        for each action in problem.Actions(node.STATE) do
        child ← CHILD-NODE( problem, node, action)
        result ← RECURSIVE-DFS( child, problem)
        if result ≠ failure return result
    return failure
end function
```

states that have been created. This is stored in the explored set and in the example this includes all nodes except the the ones with faint dashed lines. An informal description of the Graph Search algorithm can be seen in Algorithm 1. Finally if the algorithm has returned one or more solution(s) plan(s) then the final step is generate the solution(s) to the problem by following the steps in the paths.

#### 2.3.3.1 The structure of the search nodes and Depth First Search

Search Graphs need a data structure to be able to save all the generated nodes and also to be able to generate a solution from a returned solution path. The structure of the Graph requires for a node N at least the following:

- N.STATE the state inside the state space that corresponds to this node.
- N.PARENT: the node that preceded this node
- N.ACTION: the action performed on the parent node to reach this node.
- N.COST: the cost to reach this node, this cost the addition of the cost of parent plus the cost to reach node N.

Algorithm 2 shows the structure of the Depth First Search Algorithm. The idea is that it expands first the node with that was generated last/is the deepest inside the Graph. This is the main algorithm we will use for graph traversing, but other graph traversing algorithms could also be used. The example in Figure 2.4 shows

the exactly the first two and the last step of the Depth First Search algorithm on the running example database.

# Chapter 3

# Supervised Relational Feature Generation Algorithm

The problem we are trying to solve is having a relational database with tables $R_1, R_2, ..., R_N$ where one table Ri is the base table of interest that has the target attribute T we wish to model. We consider the records of T as identically and independently distributed, or i.i.d. for short because this is the form many machine learning algorithms assume the data are in. The goal of the algorithm is to produce features for the base table (denoted as S) useful for modelling target attribute T. To do this we have to gather data from other tables inside the database to extend the features the base table has. To produce features from the relational database we have to navigate the join graph $G_{join}$ where each edge between two relation nodes corresponds to a join condition we can use to join these relations. A path $R_1 \rightarrow R_2 \rightarrow \cdots \rightarrow R_N$ in $G_{join}$ corresponds to the join of relations $R1 \bowtie R2 \bowtie ... \bowtie R_N$ (Edges can be traversed both ways). When we visit 1-many relationships with relation to S i.e. $R_1 \rightarrow S$ we may have for a given t1 $\in S$ : $|\{t \in S \bowtie R1 : \pi_S(t) = t1\}| \geq 1$. This results in losing the i.i.d property and we cannot create model on the resulting table. What we can do instead is group-by PK(S) and then aggregate to produce new features that are unique for each t $\in$ PK(S) and thus can be used for modeling T.

To navigate the join graph and generate features we have modelled the problem as an AI Search problem and generate features by solving this Search Problem instead to find the solution. The generation and storing of the features will be done using a RDMS using queries which we will then store as views. After generating the features the next step will be performing feature selection to only select a subset of features that can maximize the performance of a classifier modelling T. For this matter we have developed two scallable feature selection algorithms s_gomp and p_gomp that generalize the gomp algorithm[20]. These algorithms work by inspecting the data in chunks and thus can scale to large datasets. Finally when the feature selection is finished we either can return a model trained on the data or return the constructed features and leave the modelling choice to the user. The

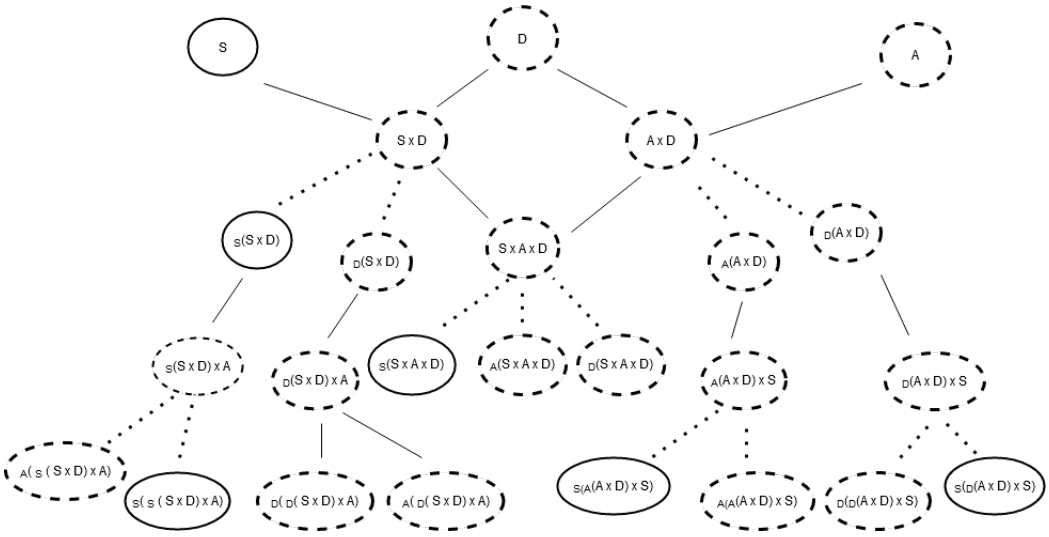



Figure 3.1: Search Space of the running example without the Disp relation. For simplicity A denotes Account, S Client and D for District. Strait line nodes correspond to nodes we can model, straight line edges are used to denote the Join action was performed while dotted line edges are used to denote a combination of Group-by and Aggregate operators was performed. By the subscript we denote name the table whose PK was used for creating the groups.

organization of this chapter is as follows, Section 3.1 describes the modelling choices we made for representing the relational feature generation problem as an AI Search Problem and how to create the Search space. Section 3.2 describes conditions that can prune this Search space to remove redundant states. Finally 3.3 describes the main algorithm we will use for streaming feature generation and feature selection.

## 3.1 Relational Feature Construction as a Search Problem

To perform the feature generation we will present the problem as a search problem. The features will be stored using views in a database which correspond to states in the state space and all the other info that is useful for describing the state and the graph traversal will be stored in nodes.
Lets recall the example database in Example 1.1.1. In the example the base table is Client and the goal is to create features useful for modeling Client.Gender. To create these features we have to perform joins between tables following the Join graph in Figure 2.2. In Figure 2.3 we looked as to how to create a search space from the Join graph but only using the join operator. As we have argued above,

only performing joins cannot produce all possible useful features [1] for modelling T and we need to also introduce the group by and aggregation operators. This means that we will still start from the Join graph and use it to guide our search but the state space has to be expanded. A new search space taking into account aggregations, but for simplicity without the District relation, can be seen in Figure 3.4, and we can already see that even for 3 tables the state space is extensive (25 states with aggregations and only 6 without). Previous algorithms performed aggregations immediately after after joining two tables, performed only based on specific primary keys, or considered only one path from the base table to other tables. We will extend these works by allowing all combinations of join and group by and aggregations operators using any available key(s) are grouping attribute(s).

### 3.1.1 Node Structure

In order to make the state space we need to extend the structure of a node to include additional information useful for the generation of the state space and later for the materialization of the nodes we are interested in. Because only nodes useful for modeling T are materialized and we will need additional bookkeeping to guide the search. Throughout this thesis we use aggregate unless otherwise noted to also mean group-by and aggregate for simplicity.

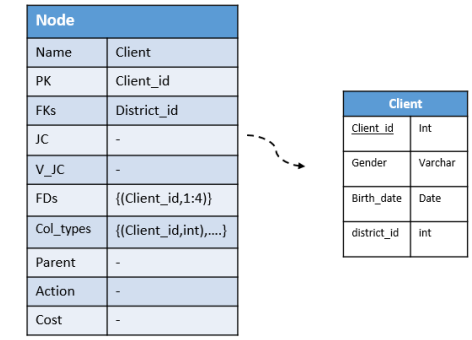
The basic structure that a node has node **N** until now was (see Fig 3.2 for an example of the structure of the node):

- the Name or State a short string summarising the operations that resulted in this node
- Parent, a string specifying the node name of the parent of this node
- Action, a string with the action that was performed on parent node to create this node.
- Cost, integer indicating the number of actions we have performed to get to this node.

The additional structure for the generation of the search space is:

- primary key PK, a string tuple that uniquely identifies every row. For the relations Ri in the database this is their primary key. When we join two nodes we get the union of their primary keys while for aggregation the primary key becomes the key that we performed the aggregation. PK is used to find Join Successors.
- foreign keys FKs, a list of strings, the attributes that refer to some PK(Ri): Ri, i=1,..,n. For nodes of the relations in the database this contains their FKs.

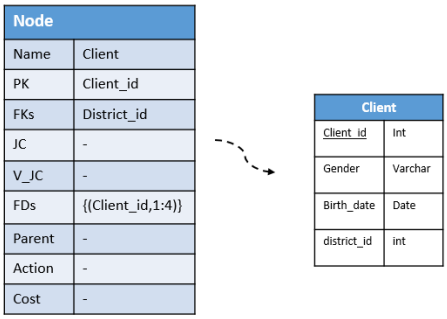
[1] In the case of joins that do not increase the Base Table's number of rows we could avoid performing aggregations.



| Node | |
|---|---|
| Name | Client |
| PK | Client_id |
| FKs | District_id |
| JC | - |
| V_JC | - |
| FDs | {(Client_id,1:4)} |
| Col_types | {(Client_id,int),....} |
| Parent | - |
| Action | - |
| Cost | - |

| Client | |
|---|---|
| Client_id | Int |
| Gender | Varchar |
| Birth_date | Date |
| district_id | int |

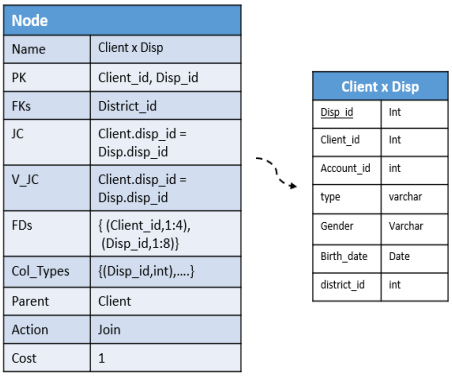
(a) Node for Client relation.



| Node | |
|---|---|
| Name | Client |
| PK | Client_id |
| FKs | District_id |
| JC | - |
| V_JC | - |
| FDs | {(Client_id,1:4)} |
| Parent | - |
| Action | - |
| Cost | - |

| Client | |
|---|---|
| Client_id | Int |
| Gender | Varchar |
| Birth_date | Date |
| district_id | int |

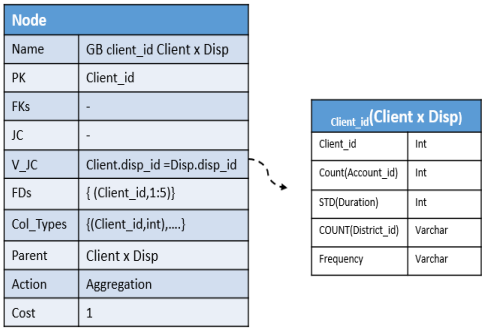
(b) Node for Disp relation.



| Node | |
|---|---|
| Name | Client x Disp |
| PK | Client_id, Disp_id |
| FKs | District_id |
| JC | Client.disp_id = Disp.disp_id |
| V_JC | Client.disp_id = Disp.disp_id |
| FDs | { (Client_id,1:4), (Disp_id,1:8)} |
| Col_Types | {(Disp_id,int),....} |
| Parent | Client |
| Action | Join |
| Cost | 1 |

| Client x Disp | |
|---|---|
| Disp_id | Int |
| Client_id | Int |
| Account_id | int |
| type | varchar |
| Gender | Varchar |
| Birth_date | Date |
| district_id | int |

(c) Node for Client x Disp.



| Node | |
|---|---|
| Name | GB client_id Client x Disp |
| PK | Client_id |
| FKs | - |
| JC | - |
| V_JC | Client.disp_id =Disp.disp_id |
| FDs | { (Client_id,1:5)} |
| Col_Types | {(Client_id,int),....} |
| Parent | Client x Disp |
| Action | Aggregation |
| Cost | 1 |

| Client_id(Client x Disp) | |
|---|---|
| Client_id | Int |
| Count(Account_id) | Int |
| STD(Duration) | Int |
| COUNT(District_id) | Varchar |
| Frequency | Varchar |

(d) Node after grouping by client_id and performing aggregations on Client x Disp .

Figure 3.2: Search Node structure for relations Client (a) and Disp (b) if they were initial states. For the FDs we used the numbering of attributes instead of the attribute names. For view Client x Disp (c) we assumed that the initial state was Client node. Finally the node that results from grouping and performing aggregations on Client x Disp view.

When we join we get the union of the foreign keys while when we aggregate this becomes empty because only PKs are grouping attributes. FKs are used along with PKs to find Join Successors.

- active join conditions JC, a list integers, of all the IDs of the join conditions that we have used in the path to create this node since last aggregation . This is initially empty, it gets populated when we perform joins and becomes empty when we aggregate. Visited join conditions list is used to avoid visiting the same edges on $G_{Join}$ again[2].

- visited join conditions V_JC, a list of integers of IDs of all join conditions

[2]Other works use visited relations list instead but this only works for Simple Graphs without loops

that we have used in the path to create this node. This is the same as join conditions with the difference that it does not become empty when performing aggregations but remains the same.

- Functional Dependencies FDs, a list of pairs of strings which correspond to attribute names. Each pair corresponds to a Functional Dependency constraint: $\{(key, cols) \mid key \in N.PK, cols \subseteq \mathcal{A}(N.V), key \rightarrow cols\}$. More specifically FDs show us for each element of the primary key the set of attributes that it uniquely identifies. This is used when we aggregate to avoid performing aggregations on attributes that have functional dependency with the grouping key because the features generated this way are either redundant or not useful for modelling T (for more details see Section 3.2). For relations Ri the FDs is the pair (PK(Ri), $\mathcal{A}$ (Ri)) because PK(Ri) by definition has functional dependency with all other attributes. When joining two relations Ri,Rj with $Ri \rightarrow Rj$ we have that Ri has a foreign key FK referencing the primary key of Rj. We join using the join condition Ri.FK=PK(Rj) and because we know that $PK(Rj) \rightarrow \mathcal{A}(N.V))$ thus we get that $Ri.FK \rightarrow \mathcal{A}(N.V))$ so we append these attributes to the attribute list of Ri.FK while for PK(Rj) they remain the same. When we perform aggregation on node N using grouping attribute A then the resulting FDs is A and a list of all the features generated from the aggregations since they are uniquely identified by A.

- column_types a list of pairs (A,B) where A is the name of an attribute of view V and B is the type of the attribute. For relations Ri this corresponds to their original schema. For join successors this becomes the union of the column_names of the nodes and for Aggregation we get the primary key along with its data type and a list with all the aggregated attributes and their data types accordingly.

Lastly these fields are used for the materialization step:

- view V, a string specifying the query that materializes this node.

- nest_q, a list of strings, the names views that this view is dependent upon. This is because views are not created from scratch but progressively to save up time since generating all the query at once can be costly and also parts of the query can be used by other nodes. This is initially empty and gets populated with the node names of all the ancestors of the node.

### 3.1.2 Modelling Nodes

We define Modelling Nodes as nodes that are uniquely identified by the base table S primary key, PK(S). It is obvious that this includes all nodes N that $PK(S) = N.PK$ but it also includes nodes that have PK(S) as a candidate key and to find this we find the transitive closure of the functional dependencies between the primary keys, meaning that we find all tables that we can reach from the base

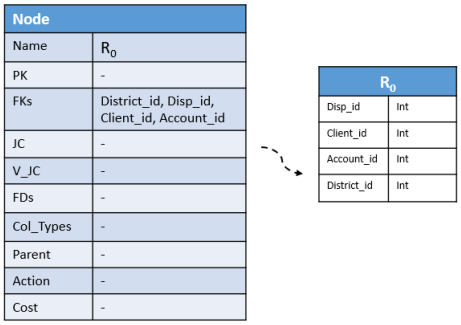



Figure 3.3: The empty relation $R_\emptyset$ that we use as an initial state in the problem instance. Notice that it only contains the references to the PK of other tables and does not have a PK on its own.

table using only many-1 edges. For example in 2.2 if we select Client as the base table, joining with District table does not increase the rows of Client since the relationship is many-1. If we had selected the Disp as base table then since there is a path following only many-1 edges from the base table to any other table, any join that we perform will not increase the number of rows of Disp and thus all nodes containing combinations joins would be modelling nodes.

### 3.1.3 Successor Function

Each node N has two types of successors: join successors and group-by and aggregate successors.

- Join successors correspond to joining node view N.V with a relation Ri that is connected via KFK relationships with N.V. using join condition E which is of the form (N.V).A = Ri.B for some attributes A and B. This is either a many-1 relationship if A is an element of N.FKs or a one-many if A is an element of N.PK. As mentioned above, to avoid visiting the same relations more than once and create redundancy we have the visited Join Conditions list.

- Groupby-Aggregate or simply Aggregate Successors correspond to grouping-by a key $\in N.PK$ and then performing aggregations on a subset of the remaining attributes. To remove redundancy here we can avoid performing aggregations on Attributes $A \in \mathcal{A}(N.V) \mid key \rightarrow A$ (see section 3.2).

### 3.1.4 Problem Instance

Now that the Successor function is defined we can define the search problem. For starting state we have two options. We can start from a relation Ri and continue with its successors based on tables connected to Ri. Otherwise we can start from

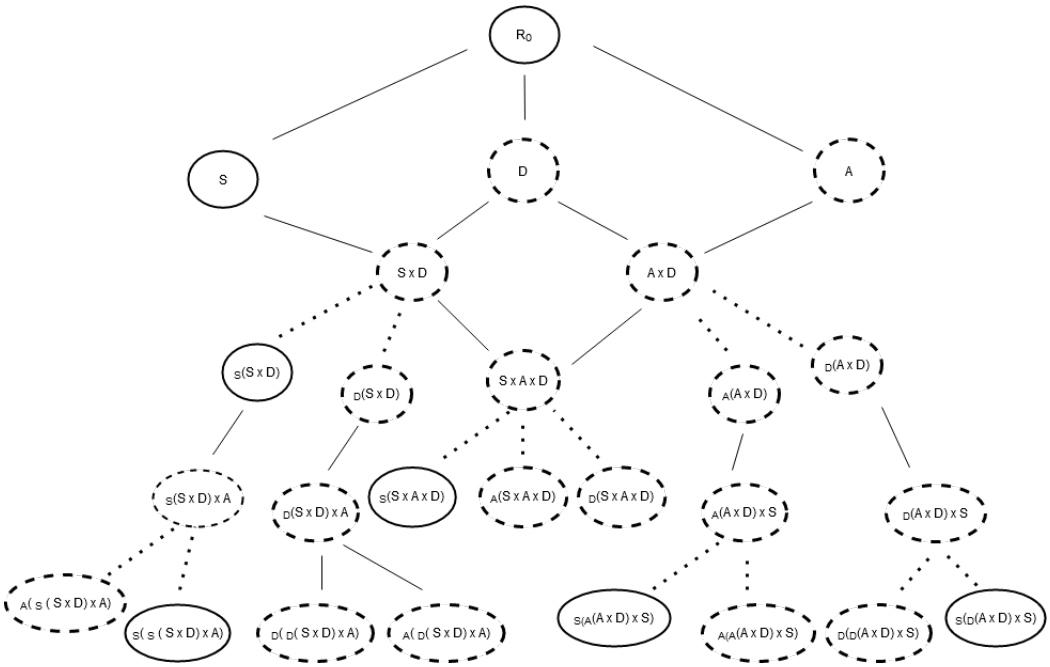



Figure 3.4: Search Space of the running example starting from $R_\emptyset$, without the Disp relation. For simplicity A denotes Account, S Client and D for District. Strait line nodes correspond to nodes we can model, straight line edges are used to denote the Join action was performed while dotted line edges are used to denote a combination of Group-by and Aggregate operators was performed. By the subscript we denote name the table whose PK was used for creating the groups.

the "empty relation" $R_\emptyset$ that we suppose is connected to all other relations (or a subset of them) via referencing their PKs with FKs (see Fig 3.3). The second option is the generalization of the first where essentially we have all or some of the base relations as starting states and this way we can produce all their successors and any combination of joins and aggregations. All the nodes are saved inside the search graph $G_{search}$ which is the output of the first part of the algorithm which is the SQL query generation.

- Initial state: Ri, where Ri any base relation or $R_\emptyset$
- Base_table where we will perform the modeling and
- Base_target the attribute for modeling
- Aggs, a list of available aggregation functions
- Goal state/test : None or Residuals based on modeling **T**
- Actions the set of available actions to derive child nodes which consists of Join, Aggregate.

- M_edge_v Maximum number of times we can visit the same edge (we can only visit the same edge only after aggregating because otherwise we create redundancy).

An example of a state space that uses $R_\emptyset$ can be seen in Figure 3.4.

## 3.2 Pruning the Search Space

Now that we have defined $G_{search}$ and produced all the available successors, the goal is to prune the graph and discard redundant nodes that bring no new information or are not useful for modeling T. By pruning these nodes we also prune their successors so we cut branches of the search tree effectively saving up execution time and space.

- **Avoid aggregate successors from keys that have FD with all the other attributes of the node:**
  For aggregation successors of a view N.V or V for simplicity, when grouping by key V.K avoid performing aggregations on attributes that are uniquely identified by V.K. If an attribute V.A is uniquely identified by V.K then when we group by V.K, the resulting groups on V.A will have size of one. The result of each aggregation function respectively will be:

  - $\pi_{K,SUM(A)}(V) = \pi_{K,A}(V)$ this is because:

$$\begin{aligned}\pi_{K,SUM(A)}(V) &= \{(k, sum(\{t(A) \mid t \in V, t(K) = k\}), k \in V.K\} \\ &= \{(k, sum(a_k)), k \in V.K, a_k = \pi_A(\sigma_{K=k}(V))\} \\ &= \pi_{K,A}(V)\end{aligned}$$

  - $\pi_{K,MAX(A)}(V) = \pi_{K,A}(V)$.
  - $\pi_{K,MIN(A)}(V) = \pi_{K,A}(V)$
  - $\pi_{K,MEAN(A)}(V) = \pi_{K,A}(V)$
  - $\pi_{K,COUNT(A)}(V) = \{(k, 1), k \in V.K\}$
  - $\pi_{K,STDEV(A)}(V) = \{(k, 0), k \in V.K\}$
  - $\pi_{K,VAR(A)}(V) = \{(k, 0), k \in V.K\}$

  It is obvious the attributes produced by the aggregation will either already be covered by the presence of A, or be constant attributes which do not have any predictive power (for the last 3 aggregation functions). In both of these cases we can safely avoid performing aggregation functions on these attributes. In the case that the grouping key V.K has functional dependency with all the other attributes (is actually a candidate key of the view V) then we can safely avoid generating the aggregate successor.

- **Don't visit the same edges again:** We use visited Join conditions list to limit the number of Join Successors of a node. We do not allow a node to visit the same edge more than once because that will not provide any new attributes because they will already be included inside the node view. However if we aggregate then visiting the same edges once more does provide new information since the attributes have now been aggregated. This is why after aggregating we empty the active Join Conditions list and we keep a separate visited Conditions list that keeps track of all Join Conditions regardless of aggregations.

- **Don't create equivalent nodes:**
  Let tables A1,A2,..,AN and $A1 \bowtie (...(A2 \bowtie A3)... \bowtie AN)$ be one of their natural joins. Due to the associative property of the natural join operator we can skip the parentheses and use the abbreviation A1 A2 ... AN. Moreover since the operator is also symmetric any permutation of A1, A2, ... , AN will lead to the same result. In contrast we can use the set { A1 , A2, ..., AN } to represent all the possible permutations. By taking advantage of this we can produce only one of the N! joins of size N between the tables. Taking this one step further only the table names cannot identify all joins between different tables for example if there exist circles in the join graph $G_{Join}$ or two tables can be joined by more than one conditions then this set will only contain one of the join results. To avoid this we will use the full join conditions instead of the table names, this way the identification of the tables as well as the join conditions are unique and all different joins are accounted. As an example for tables A,B,C and attributes A1, A2, B1, B2, C1 of tables A, B, C respectively, we can have the set of join condition $\{(A.A1 = B.B2)(A.A2 = B.B2)(B.B1 = C.C1)\}$. Lastly if one aggregation has been performed, then we have to compare the pair of group by key and the join conditions set before the aggregation, while if we have more join and group by operations then we compare all the pairs of group by key and join condition sets from first to last (assuming that after each group by we perform the same set of aggregation functions on the appropriate attributes as discussed above).

- **Stop generating nodes that cant lead to modeling nodes:**
  If the node cannot lead to modeling node, stop generating successors. For every node N.V we define the following problem by placing N.V. inside the join graph $G_{Join}$. The set of vertices **V'** is all the original relations with the addition of N.V. The set of edges **E'** consists of the result union of the original edges connecting the relations with the edges connecting N.V with all tables that either have foreign keys that reference PK(N.V) or their primary is referenced by a foreign key inside FKs(N.V) but subtracting the ones already visited for the creation of N.V which is N.visited_JC.

  – G'(V',E') as mentioned above.

  - If there doesn't exist a path P=N.V→ ... →Base_Table then stop generating Successors.

  For finding a path between N.V and Base_Table we can use any uninformed search algorithm.

- **Don't use all Ri that are intermediate nodes in graph first traversal from base table S as successors of** $R\_\emptyset$. Order the graph using Breadth-First search from the Base table and only keep the root and leaf nodes as successors of $R\_\emptyset$ since the features created from intermediate nodes will already be inside.

## 3.3 Feature Selection and Generation Pipeline

The main idea of the algorithm is that it sees the features in batches, performs feature selection and maintains only the selected features before getting the next batch. Like FBED [7] it does revisit the features more than one time to give them more opportunities to enter selection. This is important when a feature X becomes necessary for optimal prediction only in combination with some other feature Z. So, if X and Z are in different batches, X may be filtered out before Z is examined. So, we need to give X another chance to enter.There are two versions of the algorithm (see Algorithm 1 and 2) sequential and the parallel (or distributed) . Both version of the algorithm see the available data in batches based on both samples and attributes. They start with an empty set of selected features and model and the first batch of samples. Then they perform feature selection for every feature batch. The GETDATA function we can decide if we have seen enough samples or we need more to make a decision. The difference between the two versions is that in the sequential version we carry features we have selected in the previous feature batch to the next taking them into account for feature selection while in the parallel version feature selection between batches is independent of the others, and they are all gathered together at the end and a further feature selection is performed on this feature set. At the end for both versions, starting now with the selected features of the previous run, this process is repeated as long as we find new useful features to insert.

**Algorithm 3** Sequential Dynamic Feature Selection (with dropping)

**function** SeqDynFS(Initial feature subset $S$)
    //function GetFeatures, function GetData, function InitGetFeatures, function FS should be considered as provided.

    //Initialize current model, current feature set, dataset, and outcomes.
    $M, F, D, y \leftarrow \emptyset$
    $Sorig \leftarrow S$
    InitGetFeatures

    **while** the end of time (no more features) **do**
        //Get the next batch of features. The function GetFeatures should maintain an internal state to know which feature batch is next. The state is initialized by InitGetFeatures.

        $F \leftarrow$ GetFeatures
        **if** $F = \emptyset$ **then**
            //We are done with this call of the algorithm. However, if we have been pruning and the selected features have changed, we need to repeat the process and give a chance to pruned feature to enter again.
            **if** $Sorig \neq S$ **then**
                **return** SEQDynFS(S)
            **else**
                //Otherwise, we are really done
                **return** $\langle S, M \rangle$
            **end if**
        **end if**

        //Get the next batch of samples on both the selected features and the new batch of features. The model $M$ is also a parameter so that the function can determine how many samples are "enough"
        $\langle D, y \rangle \leftarrow$ GetData$(M, S, F)$
        //Perform standard feature selection starting from $S$ and updating the selected features and the model.
        $\langle S, M \rangle \leftarrow \mathrm{FS}(S, D, y, base = S)$
    **end while**
**end function**

**Algorithm 4** Parallel Dynamic Feature Selection (with dropping)

**function** ParDynFS(Initial feature subset $S$)
  $M, F, D, y \leftarrow \emptyset$
  $Sorig \leftarrow S$
  InitGetFeatures
  //Partitioning the feature set using the same exact function as before
  $F_1 \leftarrow$ GetFeatures, $i \leftarrow 1$
  **while** $F_i \neq \emptyset$ **do**
    $i \leftarrow i + 1, F_i \leftarrow$ GetFeatures
  **end while**
  $i \leftarrow i - 1$
  //Feature selection on individual feature groups, done independently and **should be parallel**
  **for** $F_i$ **do**
    $\langle D_i, y \rangle \leftarrow$ GetData$(M, S, F_i)$
    $S_i \leftarrow \mathrm{FS}(S, D_i, y)$
  **end for**
  //Partial solutions merged and a feature selection on the seleted features is performed
  $S \leftarrow \cup_i S_i$
  $\langle D, y \rangle \leftarrow$ GetData$(M, S)$
  $S \leftarrow \mathrm{FS}(S, D, y)$

  **if** $Sorig \neq S$ **then**
    **return** ParDynFS(S) //Give features one more chance to enter
  **else**
    **return** $\langle S, M \rangle$ //We are really done
  **end if**
**end function**

# Chapter 4

# Evaluation

Both Supervised Relational Feature Generation Algorithm (SRFGA) and the two version of gomp were implemented using R 4.1.0 and the experiments were run on a Windows 10 Desktop with 4-core Intel i7 processor and 32GB memory. The database we used for feature generation is Oracle. In this chapter we evaluate the performance of our proposed feature selection algorithm with two variations, p-gomp and s-gomp and also compare the performance of SRFGA to the state of the art algorithms and show how our approach covers them in terms of features generated. The outline for this capter is: Section 4.1 describes the performance experiments with the feature generation algorithm, in Section 4.2 we evaluate the performance of SRFGA and lastly in 4.4 we explain why the pruning methods we chose are effective.

## 4.1 Gomp Experiments

In this section we explain the experiments performed for both versions of the algorithm to access whether splitting the dataset in many parts and performing feature selection loses accuracy compared to when we would use all available data. The two versions of the feature selection algorithm we will use, sequential and dynamic feature selection, are shown in Alg 3 and Alg 4. Our assumption is that the accuracy loss will be small and this enables the algorithms to scale to big data as we have the option to give the data in parts. We will use the following specifications for the algorithms:

- We assume the data are stored locally in a file.
- INITGETFEATURES has only one part and initially returns all available features from each dataset.
- GETDATA returns all samples of the dataset. Further studying this function is left for future work.

| Dataset | n | p | P(T = 1) | Type | Domain |
|---|---|---|---|---|---|
| sylva | 14394 | 216 | 0.94 | Mixed | Forest Cover Types |
| madelon | 2600 | 500 | 0.5 | Integer | Artificial |
| gina | 3568 | 970 | 0.51 | Real | Handwritten Digit Recognition |
| hiva | 4229 | 1617 | 0.96 | Binary | Drug discovery |
| gisette | 7000 | 5000 | 0.5 | Integer | Handwritten Digit Recognition |
| arcene | 200 | 10000 | 0.56 | Binary | Mass Spectrometry |
| nova | 1929 | 16969 | 0.72 | Binary | Text classification |
| dexter | 600 | 20000 | 0.5 | Integer | Text classification |
| dorothea | 1150 | 100000 | 0.9 | Binary | Drug discovery |

Figure 4.1: Binary classification datasets used in the experimental evaluation, n is the number of samples, p is the number of predictors and P(T=1) is the proportion of instances where T=1.

- The FS algorithm we will use is gomp extended with the base argument that allows it to start from a given base of features and start building from there.

For simplicity we will call sequential feature selecton with gomp as s-gomp and parallel feature selection with gomp as p-gomp. Each dataset was split in train, test 0.8, 0.2 using stratified splitting based on the appropriate target feature. Then we used the training set for performing feature selection and evaluated its performance on the test set using the Random Forest algorithm with default parameters from 'randomForest' R package. To get statistically significant results, this process was repeated 5 times for each dataset and each time the dataset's rows and columns were randomly permuted.

### 4.1.1 Datasets

Figure 4.1 shows 9 datasets that were selected from [7] because they have more than 100 features and have been previously used as feature selection benchmarks. No preprocess was needed because the data were numerical and had no missing values. Notice also that they cover a range of different domains so they are ideal for testing the feature selection algorithm on real world situations.

| | P(T=1) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 25 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **arcene** | 0.56 | 0.72(0) | 0(0.1) | -0.03(0) | 0(0.1) | 0.01(0.1) | 0.03(0) | 0.01(0.1) | 0.03(0) | 0.0(0.1) | 0.05(0.1) | 0.04(0.1) | 0.06(0) |
| **dexter** | 0.5 | 0.89(0) | **-0.05(0)** | -0.07(0.1) | -0.04(0) | -0.03(0) | -0.02(0.1) | -0.03(0.1) | -0.06(0) | -0.03(0.1) | -0.04(0) | -0.03(0) | -0.01(0) |
| **dorothea** | 0.9 | 0.91(0) | 0(0) | 0(0) | 0(0) | -0.01(0) | -0.01(0) | -0.01(0) | -0.01(0) | 0(0) | 0(0) | 0(0) | 0.01(0) |
| **gina** | 0.51 | 0.93(0) | -0.01(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0.01(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |
| **gisette** | 0.5 | 0.96(0) | 0(0) | 0(0) | 0(0) | 0.01(0) | 0.01(0) | 0.01(0) | 0.01(0) | 0.01(0) | 0.01(0) | 0(0) | 0.01(0) |
| **hiva** | 0.96 | 0.96(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |
| **madeleon** | 0.5 | 0.59(0) | 0.04(0.1) | 0.04(0.1) | 0.03(0.1) | 0.07(0.1) | 0.08(0.1) | 0.03(0.1) | 0.06(0.1) | 0.03(0) | 0.05(0) | 0.07(0) | 0.08(0.1) |
| **nova** | 0.72 | 0.88(0) | 0(0) | 0(0) | 0(0) | 0.01(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0.01(0) |
| **sylva** | 0.94 | 0.99 | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |

Table 4.1: Mean (standard deviation) accuracy for each dataset (rows) and different number of splits (columns). First column corresponds to the probability of the second class and the second column is the accuracy of s-gomp on all features which is the base. The rest of the columns are calculated using the formula: accuracy(dataset,split)= accuracy(dataset,1) -accuracy(dataset,split).

| | P(T=1) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 25 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **arcene** | 0.56 | 0.77(0) | -0.02(0) | -0.02(0) | -0.01(0.1) | -0.01(0.1) | 0.02(0.1) | 0.01(0.1) | 0.01(0.1) | 0(0.1) | 0.03(0.1) | 0.01(0.1) | 0.08(0.1) |
| **dexter** | 0.5 | 0.95(0) | -0.04(0) | -0.07(0.1) | -0.04(0) | -0.03(0) | -0.02(0.1) | -0.03(0.1) | -0.06(0) | -0.03(0.1) | -0.04(0) | -0.03(0) | -0.01(0) |
| **dorothea** | 0.9 | 0.77(0) | 0.02(0) | 0.03(0) | 0.01(0) | -0.02(0) | -0.01(0.1) | -0.01(0.1) | -0.01(0.1) | 0.03(0.1) | 0.02(0.1) | 0.05(0.1) | 0.04(0) |
| **gina** | 0.51 | 0.98(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |
| **gisette** | 0.5 | 0.99(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |
| **hiva** | 0.96 | 0.68(0.1) | -0.01(0) | -0.01(0) | -0.01(0) | -0.02(0.1) | -0.02(0.1) | -0.01(0) | -0.03(0) | -0.02(0.1) | -0.01(0) | 0(0) | 0.01(0) |
| **madeleon** | 0.5 | 0.62(0) | 0.06(0.1) | 0.06(0.1) | 0.05(0.1) | 0.1(0.1) | **0.11(0.1)** | 0.04(0.1) | **0.09(0.1)** | 0.05(0) | 0.08(0) | 0.1(0) | 0.12(0.1) |
| **nova** | 0.72 | 0.88(0) | -0.01(0) | -0.01(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0.01(0) | 0(0) | 0.01(0) | 0.02(0) | 0.03(0) |
| **sylva** | 0.94 | 1 | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |

Table 4.2: Mean (standard deviation) auc of s-gomp for each dataset (rows) and different number of splits (columns). First column corresponds to the probability of the second class and the second is the auc of s-gomp on all features which is the base. The rest of the columns are calculated using the formula: auc(dataset,split)= auc(dataset,1) - auc(dataset,split).

### 4.1.2 Results

Table 4.1 shows the accuracy of s-gomp in comparison to the number of splits. We can see that the accuracy does not significantly change by splitting into 2 parts or more, for example in gina and hiva even splitting into 50 parts does not change the performance. In some case we can also see that it achieves greater accuracy (dexter dataset), but the differnce is very small. These results can also be confirmed by Table 4.2 that shows us the AUC of the Random Forest model on the same datasets. On the second table we can see that for datasets hiva and nova we have some diffence but it again not statistically significant. One thing we have to note for AUC is that in the case of madeleon dataset, which is artificially generated, for number of splits greater than 6 there are some significant deviations (0.11 for num_splits=6 and 0.12 for num_splits=50). This shows the ability of s-gomp to scale for big datasets without sacrificing the robustness of the predictions, but there are cases when we can see a difference in performance and ideally we should use the the least amount of splits as possible to reduce this effect.

Next we have the p-gomp experiments on the same datasets. Table 4.3 shows the accuracy of p-gomp in comparison to the number of splits. Similarly the accuracy does not significantly change by splitting into 2 parts and in fact it is almost always 0 for most of the splits and when it is not 0 the difference is very small (maximum 0.05 on arcene split 3). These findings can also be confirmed by Table 4.4 that shows us the AUC of the model on the same datasets, here we can see that even in the madeleon dataset when the previous algorithm did not perform well with increased number of split, pgomp even with 50 number of splits loses only 0.05 AUC which for very large datasets can have a very large computational speed

| | P(T=1) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 25 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **arcene** | 0.56 | 0.69(0.1) | 0.03(0.1) | 0.05(0.1) | 0.01(0.1) | 0.04(0.1) | 0.02(0) | 0.04(0.1) | 0.03(0.1) | -0.01(0.1) | 0.02(0) | 0.03(0.1) | -0.01(0.1) |
| **dexter** | 0.5 | 0.89(0) | 0.02(0) | 0(0) | 0(0) | 0.02(0) | 0.01(0) | 0.01(0) | 0.01(0) | 0.02(0) | 0.02(0) | 0.01(0) | 0.02(0) |
| **dorothea** | 0.9 | 0.91(0) | 0(0) | 0.01(0) | 0(0) | 0(0) | 0.01(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | -0.01(0) |
| **gina** | 0.51 | 0.93(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |
| **gisette** | 0.5 | 0.96(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0.01(0) |
| **hiva** | 0.96 | 0.97(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |
| **madeleon** | 0.5 | 0.61(0) | 0(0) | -0.01(0) | -0.01(0) | -0.01(0) | -0.01(0) | -0.01(0) | 0(0) | 0(0) | -0.01(0) | 0.03(0.1) | 0.03(0) |
| **nova** | 0.72 | 0.88(0) | 0.01(0) | 0.01(0) | 0.01(0) | 0(0) | 0.01(0) | 0.01(0) | 0(0) | 0.01(0) | 0.02(0) | 0.01(0) | 0.01(0) |
| **sylva** | 0.94 | 1 | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |

Table 4.3: Mean (standard deviation) accuracy for each dataset (rows) and different number of splits (columns). First column corresponds to the probability of the second class and the second is the accuracy of p-gomp on all features which is the base. The rest of the columns are calculated using the formula: accuracy(dataset,split)= accuracy(dataset,1) -accuracy(dataset,split).

| | P(T=1) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 25 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **arcene** | 0.56 | 0.78(0) | -0.02(0) | -0.02(0) | 0(0) | -0.01(0.1) | -0.02(0.1) | 0.01(0.1) | -0.01(0.1) | 0(0) | -0.01(0) | 0.01(0.1) | 0(0.1) |
| **dexter** | 0.5 | 0.95(0) | 0.01(0) | 0(0) | 0(0) | 0.01(0) | 0(0) | 0.01(0) | 0.01(0) | 0.01(0) | 0.01(0) | 0.01(0) | 0.01(0) |
| **dorothea** | 0.9 | 0.84(0.1) | 0.01(0) | 0(0.1) | -0.01(0) | 0(0.1) | 0.01(0.1) | 0.01(0) | -0.01(0) | 0.01(0) | 0.02(0) | -0.01(0) | 0.01(0) |
| **gina** | 0.51 | 0.98(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |
| **gisette** | 0.5 | 0.99(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |
| **hiva** | 0.96 | 0.66(0) | 0(0) | 0.01(0) | 0(0) | 0(0) | 0.01(0) | 0.02(0) | 0.01(0) | 0.02(0) | 0(0) | 0.02(0) | 0.02(0) |
| **madeleon** | 0.5 | 0.64(0) | -0.01(0) | 0(0) | 0(0) | 0(0) | -0.01(0) | 0(0) | 0(0) | 0(0) | 0(0) | **0.06(0.1)** | **0.05(0.1)** |
| **nova** | 0.72 | 0.86(0) | 0.03(0) | 0.01(0) | 0.03(0) | 0.01(0) | 0.03(0) | 0.02(0) | 0.01(0) | 0.02(0) | 0.03(0) | 0.03(0) | 0.04(0) |
| **sylva** | 0.94 | 1 | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) | 0(0) |

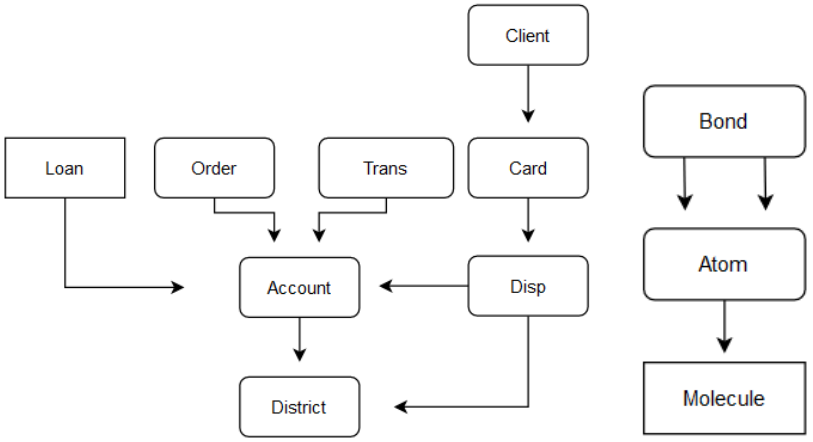
Table 4.4: Mean (standard deviation) auc of p-gomp for each dataset (rows) and different number of splits (columns). First column corresponds to the probability of the second class and the second is the auc of p-gomp on all features which is the base. The rest of the columns are calculated using the formula: auc(dataset,split)= auc(dataset,1) - auc(dataset,split).

benefit.

We have to note that for both s-gomp in some cases p-gomp the algorithm took as much as twice amount of time when splitting to two or more parts, but this is not a limitation as the goal is for the algorithms to be able to run feature selection on datasets that are too large to be able to be seen as a whole and can only be seen in parts. If we compare the two versions of the algorithms we notice that p-gomp was on par or outperformed s-omp in all datasets which is to be expected since p-gomp keeps all the useful features and then decides from the union which ones to keep, while s-gomp only keeps the best so far which in some cases can lead to removing features that might be useful only in a context of other features that have not been seen yet.

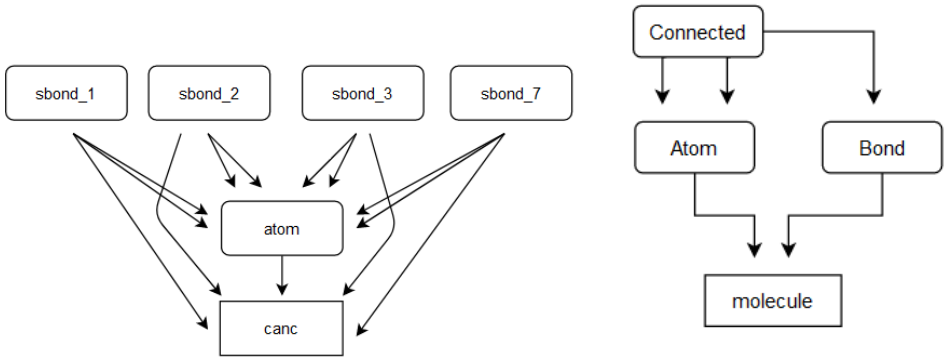
## 4.2 Cover Experiments

We will use 5 algorithms for this part of the experiment. For our implementations SRFGA will denote only feature generation without the use of feature selection algorithm and p-gomp, s-gomp will be the algorithms with feature selection as mentioned above. For DFS we will use the implementation provided in the 'feature-tools' R package. For the forward algorithms AFEM and OneBM we will evaluate them as a direct subset of our algorithm denoted as Forward algorithm. Note that our implementation of them is possibly an extension of these two algorithms as to the best of our knowledge they cannot cover multiple edges, multi-paths or circles. For each algorithm we record the time and memory during feature generation and then take the output dataset and use it as input for Jadbio for automated model




(a) Financial Database

(b) Mutagenesis Database.




(c) Carcinogenesis Database.

(d) Toxicology Database.

Figure 4.2: Relational schemas of the Databases used in the Experiments. Squares edges denote the target table, and arrows denote foreign key reference.

selection and training, and report the final AUC. The pipeline for this experiment can be seen in Figure 4.3.

### 4.2.1 Relational Databases

Fig 4.2 shows the schemas of the relational databases we will use. All four datasets were chosen because they have been previously used for relational feature engineering benchmarks and their schema is a complex Multi-Graph possibly with circles. For all datasets the target to predict is binary. More specifically:

- Financial: predict if a loan will be successful.
- Mutagenesis: predict if a given molecule is mutagenic.

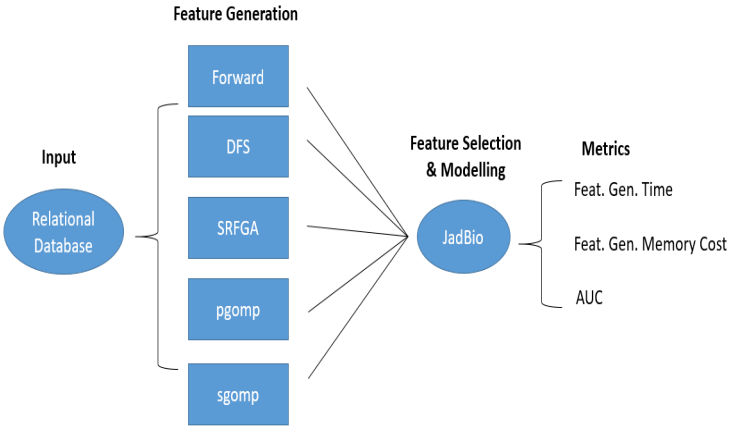



Figure 4.3: Comparison pipeline. It takes as input relational database, performs feature generation based on the selected algorithm and all the generated features are used as input to Jadbio that performs Automated feuture selection and modelling.

- Carcinogenesis: predict for a given molecule if it is carcinogenic or not.
- Toxicology: predict for a given molecule if it is carcinogenic or not.

### 4.2.2 Results

Table 4.5 reports the performance of the algorithms for the given datasets. In all databases all algorithms are able to improve the performance of the base table. It is noteworthy also that in all of them our 3 algorithms are either on par or outperform Forward and DFS. Especially in Carcinogenesis where the graph is the most complex we can see that our algorithms performed a lot better than DFS which cannot handle multiple edges between tables. The difference we have with Forward is that it only groups by PK(S), in most cases this was enough to get most of the performance but on Toxicology we see a small difference.

Tables 4.6 and 4.7 show the total time and maximum memory used by each algorithm during feature generation. From the algorithms DFS was by far the fastest, while P-gomp was always the longest, especially in Financial the difference is significant. In all datasets S-gomp and P-gomp take at least 50% more time than SRFGA and this is due to the fact that they revisit the dataset more than once. As expected P-gomp and S-gomp significantly reduce the memory impact of the algorithm and always have the lowest cost from all the algorithms. Essentially with P-gomp and S-gomp we trade-off running time for performance and memory impact so if time is not a constraint these are the versions of the algorithms that should be used.

| | Molecule | Financial | Toxicology | Carc. 20k |
|---|---|---|---|---|
| **Base Table** | 0.92 | 0.67 | 0.5 | 0.5 |
| **Forward** | 0.95 | **1** | 0.53 | 0.82 |
| **DFS** | 0.96 | **1** | 0.57 | 0.52 |
| **SRFGA** | 0.94 | **1** | 0.57 | 0.82 |
| **P-gomp** | **0.96** | **1** | **0.69** | 0.57 |
| **S-gomp** | **0.96** | **1** | **0.69** | **0.86** |

Table 4.5: AUC of the final model selected by Jadbio.

| | Molecule | Financial | Toxicology | Carc. 20k |
|---|---|---|---|---|
| **Forward** | 8s | 1.6h | 3m | 38m |
| **DFS** | 1s | 10s | 10s | 2s |
| **SRFGA** | **10s** | 2h 35m | 9m | 40m |
| **S-gomp** | **10s** | 3h 10m | 23m | **50m** |
| **P-gomp** | **10s** | **3h 15m** | **31m** | **51m** |

Table 4.6: Total time for feature generation.

| | Molecule | Financial | Toxicology | Carcinogenesis 20k |
|---|---|---|---|---|
| **Forward** | 177 | $29 \cdot 10^3$ | $5 \cdot 10^3$ | $7 \cdot 10^3$ |
| **DFS** | 173 | $1.2 \cdot 10^3$ | 800 | 255 |
| **SRFGA** | **727** | $70 \cdot 10^3$ | $18 \cdot 10^3$ | $4.6 \cdot 10^3$ |
| **S-gomp** | 10 | 8 | 11 | 30 |
| **P-gomp** | 8 | 8 | 11 | 50 |

Table 4.7: Maximum memory used by each algorithm in Killobytes during feature generation.

| | Molecule | Financial | Toxicology | Carcinogenesis 20k |
|---|---|---|---|---|
| **Base Table** | 0.5 | 0.5 | 0.5 | 0.5 |
| **Forward** | 0.95 | 0.97 | 0.98 | 0.92 |
| **DFS** | 0.98 | 0.86 | 0.75 | 0.66 |
| **SRFGA** | 0.96 | **0.97** | 0.98 | 0.94 |
| **P-gomp** | 1 | **0.97** | **1** | **0.96** |
| **S-gomp** | 0.93 | **0.97** | **1** | **0.96** |

Table 4.8: Simulations: AUC of the final model selected by JadBio.

| | Molecule | Financial | Toxicology | Carcinogenesis 20k |
|---|---|---|---|---|
| **Ideal** | 1 | 0.67 | 0.75 | 0.5 |
| **Forward** | 0.5 | 0 | 0.12 | 0 |
| **DFS** | 0 | 0 | 0 | 0 |
| **SRFGA** | 0.33 | **0.33** | 0.4 | 0.5 |
| **P-gomp** | 0 | **0** | **0.67** | **1** |
| **S-gomp** | 0 | **0** | **0.67** | **1** |

Table 4.9: Simulations: Precision (True Features Found/ Total Features Found) of the final model selected by JadBio.

## 4.3 Simulation Experiments

In this section we show that some of the features we produce cannot be approximated by other algorithms. For this task we selected features that we produce but DFS doesn't and generated an outcome using logistic regression generator. As Ideal we denote the classifier that is build using the true predictor features only. In most cases DFS and Forward were able to successfully approximate these features, in Table 4.8 we show an example of cases where approximation was more difficult. In most datasets the difference is not significant but in Carcinogenesis it is clear that the limitation that DFS has for Simple Graphs made the approximation impossible. It is interesting to inspect the results together with Table 4.9 that shows that even though the algorithm could not find the exact features in most cases, the performance wass close to perfect (For Example P-gomp and S-gomp in Financial) and even performed better than Ideal classifier that had all the true predictive features in Carcinogenesis.

## 4.4 Pruning Experiments

In this section we test the pruning methods and prove that we don't lose important nodes but only remove redundant ones, which shows that using more pruning methods can only be beneficial. We show the results only for the Financial databases but the results are similar for the other databases.

| Method | Total Nodes | Model Nodes | Time |
|---|---|---|---|
| **Without Pruning** | 109.000 | 4.557 | 336 |
| **With Pruning** | 14.091 | 4.557 | 20 |

Table 4.10: Results of pruning nodes that will not lead to model nodes on Financial database. Time column is in minutes.

| Method | Total Nodes | Model Nodes |
|---|---|---|
| **Without pruning** | 10.000 (210) | 4976 (89) |
| **With duplicate pruning** | 2895 (2895) | 718 (718) |

Table 4.11: Results of pruning nodes that will tha are duplicate using Node names on Financial database. We limited both approaches to only 10.000 total nodes.

In the first experiment we tested the performance of Path Pruning that removes any nodes that do not have a path that leads to model nodes. From Table 4.10 it is obvious that the pruning is successful as it leads to less total nodes as we expected and less total time, and we don't lose any model nodes in the process.

The goal of the second experiment is to measure the performance gain of removing duplicate nodes using the name of the nodes. This pruning method is also successful (Table 4.11) as not using it creates significant redundancy. Only 210 out of 10.000 total nodes and 89 out of 4976 model nods are unique. In comparison when using the pruning we capture 2895 total unique nodes and 718 unique model nodes that also include the 89 nodes of the previous method.

It is important to note that the two pruning methods are independent as they focus on different ways to prune the graph, so they can easily be used together without compromising modelling nodes.

# Chapter 5

# Related Work

## 5.1 Inductive Logic Programming

Inductive Logic Programming sees tables as entities which consist of facts. For example Customers(1,M,23/1/1950,3/2/20) states the fact that the customer with ID=1 which is a Male with birth Date of 23/1/1950 joined the Database on 3/2/20. These approaches work by making induction based on the facts as in: $connected(A, B) \leftarrow exists\_link(B, A)|exists\_link(A, B)$, which States that if there is a link for A to B or from B to A then A and B are connected. The induction of facts is done using first order logic operations like $\cup, \cap, \in, \exists, \neg$ but some works extend this set to include more operations.

FOIL [18] is one of the first ILP systems to use top-down induction, guided by an information based criteria to construct a Prolog clause. TILDE [6] is a first order upgrade of propositional decision trees that uses a top-down induction to create a first order logical decision tree which expands the previous approach by adding existential and universal quantifiers through the use of negation. Following Tilde, Tilde-RT [9] was created that can handle regression problems in addition to classification to which Tilde was limited .

As ILP based approaches are generally limited to the binary existence quantifier, it can only tell if the row being predicted has a related instance in the other table which satisfies the predicate. For example, it could have a feature quantifying if the total price of product was 5 with value 0 or 1 but not a feature having a real time value of the price. Moreover ILP algorithms work only with binary data and the learning phase is tightly coupled with the attribute generation phase which makes it incompatible with most of the existing learning techniques.

## 5.2 Multi Relational Data Mining

Multi-relational Data Mining (MRDM) techniques help in finding patterns and applying learning techniques on databases which store information in multiple tables [12]. In literature it is also sometimes called Relational Data Mining (RDM).

| Dataset | #tuples in target table | #tables | target class distribution |
|---|---|---|---|
| Financial 234 | 234 | 8 | 203:31 |
| Financial 682 | 682 | 8 | 606:76 |
| Mutagenesis | 188 | 3 | 125:63 |
| Thrombosis | 770 | 5 | 695:75 |

Figure 5.1: An example of common datasets used for comparison of algorithms in Relational Classifiers

The need for MRDM arose with the widespread need to store data in using the relational database model. In the relational database model, information is spread across many different tables, so propositional techniques that require all the data to fit in a 2-D table cannot work, hence the need for MRDM. Many methods have been proposed which are described below.

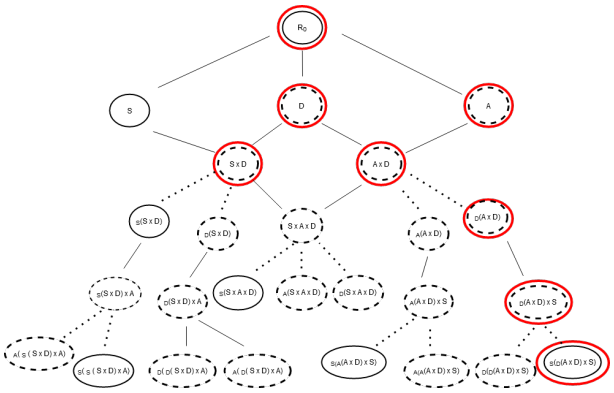
### 5.2.1 Relational Classifiers

Relational classifiers is an extension to ILP classifiers, which use the relational model to describe the data instead of first order logic. In [12] they propose MRDTL an extension of decision trees for classification that work on relational databases and MRDTL-2 [3] makes it more scalable and adress missing values in the database. Mr-Smoti [2] propose an upgrade of regression trees for relational data. Finally we have ensemble models that propagate target attribute to non-base tables and train a relational model based on an ensemble of models produced on each table in the database. More specifically MVC-2 [14] creates a clustering model using naive bayes and decision trees while Graph-NB [8] focuses on naive bayes and SDF [5] on decision trees.

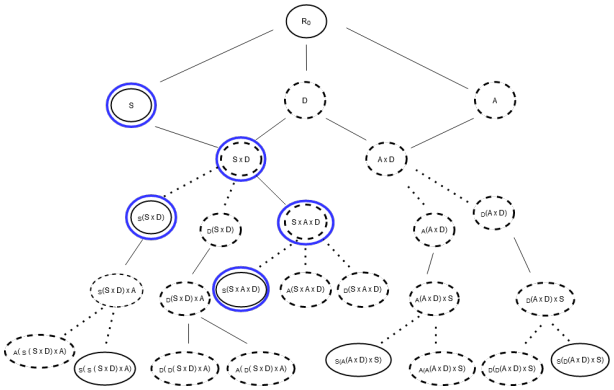
Relational Classifiers although they can work with non binary data, the features they produce consider all possible values of attributes and thus makes them impossible to scale for big data. Figure 5.1 taken from [8] depicts common datasets used for comparison of these works, and as we can see the data size is limited. Lastly they are also limited because they can only perform either classification or regression tasks not both and because they couple the feature generation with the modeling.

### 5.2.2 Propositionalization

Propositionalization algorithms gather data on one table so that traditional propositional algorithms can work. It divides the learning task in two parts,in the first part it summarises the information of all tables into one table using a combination of joins, aggregations or other transformation functions and secondly (optional)




(a) Backward Algorithms




(b) Forward Algorithms.

Figure 5.2: Search graph for the running example S is Client, D is District, A is Account. Straight circles correspond to modeling nodes. Outer bold circles note the path each family of algorithms will take.

pre-process the final dataset and return a model. There are two version of Propositionalization algorithms that differ on where they start performing joins. Forward we denote the algorithms start from S to reach the other tables while Backward are the algorithms that start from all the non-base tables to perform joins and reach the base table. Figure 5.2 shows Forward and Backward approaches on the state space of the running example.

DFS [11] introduced the Forward algorithms where they follow table relationships starting non-base tables performs joins and performs a number of predefined aggregators if last relationship visited is 1-many with relation to the last table inserted, or copies the attribute otherwise. In the join graph (Figure 5.2(a) ) it can be seen that because the relationship A-D is 1-many w.r.t D we group by PK(D) and perform aggregations, while when we join D with S the relationship is many-1 w.r.t S so we don't need to perform aggregations. Limitations of DFS is that it only works for numerical data, it performs simple aggregations like maximum, summary ,average and that it does not perform feature selection. Dataconda [19] is based

on the same idea, but additionally allows repeats of tables that are already inside but only of the form A – B – A where A-B have 1-many relationship (A and D in our example) and it also allows categorical and date attributes. Moreover it allows refinements (e.g. sum of price where price>50) which could create useful features but makes it impossible to scale for big data and optionally performs feature selection using Lasso at the end of feature generation. Likewise Predictor Factory [15] is an online tool for propositionalization that handles numeric, categorical, and Date features and performs feature selection at the end using chi-squared test.

From the Forward algorithms we have OBM [13]. These algorithms start from the base table and have one path for each of the other non base tables. If in this path they follow any non many-1 relationships, they group by PK(S) and perform aggregations. In 5.2(b) we can see that for D we only copy they features but when we reach A we need to groupby PK(S). OneBM extends DFS by working with also categorical, timestamps and freetext attributes, introducing new aggregators for these data types and finally performing feature selection but only at the end after all features are produced. Similarly, AFEM is also a Backward approach that [21] extends OBM by firstly by adding more supported attributes types, including social graph-based, spacial and representation-based attributes (produced from PCA, SVD) and extending the available aggregators for these types. Secondly it introduces the notion instance neighborhood which includes instances of one attribute that are close based on some metric for example a time interval, a geographic region or a set of people. Lastly it takes a step into performing feature selection along with the feature generation. More specifically it splits available attributes into groups based on their type e.g. numeric, categorical and produces features one category at a time, performing feature selection based on a model in between two categories. Lastly Wordification [17] is another tool that discretizes all values , considers each sample as a document with each attribute value being a word and uses Term Frequency-Inverse Document Frequency (TF-IDF) to weigh their importance and performs feature selection by removing the least frequent works.

When comparing Forward to Backward algorithms we can see that Forward algorithms always have the base table information and this is why the can perform feature selection at each step, while Backward algorithms only at the end reach a modeling node that has the base table and cannot perform feature selection earlier. These two works produce the same features for table that are directly connected with S like District in our example.

One of the limitations of the above works is that they produce a large number of features and only perform feature selection at the end, which leads to many redundant features and costing a lot of space. Redundancy is also introducing by not taking into account the predictor variable in the feature generation process in contrast with the relational classifiers. Finally, all these approaches can only take full advantage of simple graphs, and cannot fully capture all the information of of Multi-Graphs as they traverse each relation once and don't address circles in the graph.

**Algorithm 2** The generalised OMP algorithm

**Input**: Outcome values $y$, dataset $X$
**Hyper-Parameters**: Functions $f, Resid, Assoc, Stopping$
**Output**: A subset $\mathcal{S} \subseteq \mathcal{X}$ of selected features.
$\mathcal{S} \leftarrow \emptyset$
Initialize current model $M \leftarrow f(y, X_{\mathcal{S}})$
Initialize previous model $M' \leftarrow \emptyset$
Initialize residuals $r \leftarrow Resid(y, X_{\mathcal{S}}, M)$
**while** $Stopping(M, M', y, X)$ **do**
    $X_* \leftarrow \arg\max_{i \in \mathcal{X} \setminus \mathcal{S}} Assoc(r, X_i)$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{X_*\}$
    $M' \leftarrow M$
    $M \leftarrow f(y, X_{\mathcal{S}})$
    $r \leftarrow Resid(y, X_{\mathcal{S}}, M)$
**end while**
**return** $\mathcal{S}$

Figure 5.3: The generalized OMP algorithm taken from [20].

### 5.2.3 Feature Selection

Feature selection for predictive analysis is the problem of identifying a minimal-size subset of features that is maximally predictive for an outcome of interest. Feature selection helps us deal with the curse of dimensionality by rejecting useless or redundant features, making training of machine learning algorithms faster while at the same time increasing their performance. Especially in big data feature selection algorithms need to be able to scale to tens or hundreds of thousands of features and be able to deal with many different target outcomes.

We will focus on two algorithms gOMP [20] and FBED [7]. Algorithm gOMP (see Figure 5.3) is a generalized version of the Orthogonal Matching Pursuit (OMP) algorithm that is highly scalable and is able to handle many different types of predictors. It is a forward greedy algorithm that sequentially tries to find the best feature to minimize different types of residuals based on the variable outcome. FBED is a forward backward feature selection algorithm that performs early dropping, also after termination, the algorithm is allowed to run up to K additional times every time initializing the set of selected variables to the ones selected in the previous run and finally at the end performs a backward phase to discrard redundant features.

# Chapter 6

# Summary & Future Work

The purpose of this work was to create an algorithm to Automate Feature Construction for Relational Data Bases. We reviewed the literature and found two approaches, Relational Classifiers and Propositionalization algorithms. Classification algorithms [2, 3, 5, 8, 12, 14] are an extension of batch classifiers and produce models that can work directly on the database. The problem with relational classifiers is that they cannot efficiently scale to large volumes of data, usually work with either classification or regression data and they couple feature generation with modeling and do not allow the user to select the model. Propositionalization algorithms [11, 13, 15, 17, 19, 21] only focus on feature generation, and produce features on the database by performing joins and aggregations. From the works that had available code and documentation [11, 15, 17, 19] we performed experiments and noticed that only [11] can scale to big databases. We shifted our focus on works that can easily scale to big databases [11, 13, 21] and noticed that if we don't take into account different aggregation and transformation functions; the features produced are divided in two categories [11] and [13, 21] respectively based on where they start producing features and the group-by keys they produce. It was also evident that these works could not take full advantage of complex graphs like Multi-Graphs because they would only look at each relation once. We created Supervised Relational Feature Engineering Algorithm, that unifies the previous works in what features they produce, which also produces extra features on Multi-Graphs as it is able to capture more useful information. Finally, because Propositionalization works produce large amounts of features by nesting aggregation function, we coupled feature generation with a scalable feature selection algorithm that only sees features in batches and keeps the most useful so far and it also because it is greedy in this was it visits the features more than once to allow more useful features that were discarded a change to be included in the model. We showed that using this feature selection we can drastically reduce the memory cost of the algorithm and in some cases increase the accuracy by trading off computational time.

This work is only a first step in creating a system that can easily gather all the information inside a complex graph. Possible future work includes:

Increase scope of experiments:

- Find databases with different and more complicated topologies.
- Experiment with more modelling and feature selection algorithms.

Increase quality of solutions:

- Traverse circles more than once, since they could include useful features.
- Add operators (e.g. Filter operator) or transformation functions.
- Add aggregation functions to cover more types (e.g. Characters) .
- Add multivariate aggregation functions (take into account more than 1 attributes).

Scale algorithm to databases with large number of samples, tables or relationships:

- Optimize graph traversal and query materialization.
- Improve Pruning to remove possible duplicate features.
- Add more specified pruning that is guided by heuristics and does not hold in all cases.

# Bibliography


[1] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[2] Annalisa Appice, Michelangelo Ceci, and Donato Malerba. Mr-smoti: A data mining system for regression tasks tightly-coupled with a relational database. In *KDID*, pages 17–27, 2003.

[3] Anna Atramentov, Hector Leiva, and Vasant Honavar. A multi-relational decision tree learning algorithm–implementation and experiments. In *International Conference on Inductive Logic Programming*, pages 38–56. Springer, 2003.

[4] Edward A Bender and S Gill Williamson. *Lists, decisions and graphs.* S. Gill Williamson, 2010.

[5] Bahareh Bina, Oliver Schulte, Branden Crawford, Zhensong Qian, and Yi Xiong. Simple decision forests for multi-relational classification. *Decision Support Systems*, 54(3):1269–1279, 2013.

[6] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial intelligence*, 101(1-2):285–297, 1998.

[7] Giorgos Borboudakis and Ioannis Tsamardinos. Forward-backward selection with early dropping. *The Journal of Machine Learning Research*, 20(1):276–314, 2019.

[8] Hailiang Chen, Hongyan Liu, Jiawei Han, Xiaoxin Yin, and Jun He. Exploring optimization of semantic relationship graph for multi-relational bayesian classification. *Decision Support Systems*, 48(1):112–121, 2009.

[9] Luc De Raedt and Hendrik Blockeel. Using logical decision trees for clustering. In *International Conference on Inductive Logic Programming*, pages 133–140. Springer, 1997.

[10] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*, volume 7. Pearson, 2017.

[11] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE international conference on data science and advanced analytics (DSAA)*, pages 1–10. IEEE, 2015.

[12] Arno J Knobbe, Arno Siebes, and Daniël Van Der Wallen. Multi-relational decision tree induction. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 378–383. Springer, 1999.

[13] Hoang Thanh Lam, Johann-Michael Thiebaut, Mathieu Sinn, Bei Chen, Tiep Mai, and Oznur Alkan. One button machine for automating feature engineering in relational databases. *arXiv preprint arXiv:1706.00327*, 2017.

[14] Shraddha Modi. Relational classification using multiple view approach with voting. *International Journal of Computer Applications*, 70(16), 2013.

[15] Jan Motl. Predictor factory tool. `http://predictorfactory.com/doku.php`.

[16] P Russel Norvig and S Artificial Intelligence. *A modern approach*. Prentice Hall Upper Saddle River, NJ, USA:, 2002.

[17] Matic Perovšek, Anže Vavpetič, Janez Kranjc, Bojan Cestnik, and Nada Lavrač. Wordification: Propositionalization by unfolding relational data into bags of words. *Expert Systems with Applications*, 42(17-18):6442–6456, 2015.

[18] J. Ross Quinlan. Learning logical definitions from relations. *Machine learning*, 5(3):239–266, 1990.

[19] M Samorani. Dataconda: A software framework for mining relational databases,". In *DBKDA 2015, The Seventh International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 132–133, 2015.

[20] Michail Tsagris, Zacharias Papadovasilakis, Kleanthi Lakiotaki, and Ioannis Tsamardinos. A generalised omp algorithm for feature selection with application to gene expression data. *arXiv preprint arXiv:2004.00281*, 2020.

[21] Jianyu Zhang, Françoise Fogelman-Soulié, and Christine Largeron. Towards automatic complex feature engineering. In *International Conference on Web Information Systems Engineering*, pages 312–322. Springer, 2018.