Chapter 7

Who Wants My Product? Affinity-Based Marketing

Euler Timm

viadee IT-Consultancy, Münster/Köln (Cologne), Germany

Acronyms

CRISP-DM - Cross-Industry Standard Process for Data Mining
DWH - Data Warehouse
BI - Business Intelligence

7.1 Introduction

A bank introduces a new financial product: a type of current (checking) account with certain fees and interest rates that differ from those in other types of current accounts offered by the same bank. Sometime after the product is released to the market, a number of customers have opened accounts of the new type, but many others have not yet done so. The bank's marketing department wants to push sales of the new account by sending direct

mail to customers who have not yet opted for it. However, in order not to waste efforts on customers who are unlikely to buy, they would like to address only those 20 percent of customers with the highest affinity for the new product.

This chapter will explain how to address the business task sketched above using data mining. We will follow a simple methodology: the Cross-Industry Standard Process for data mining (CRISP-DM) [1], whose six stages will be applied to our task in the following subsections in their natural order (although you would typically switch back and forth between the stages when developing your own application). We will walk step by step through the fictitious sample data, which is based on real data structures in a standard data warehouse design, and the RapidMiner processes provided with this chapter to explain our solution.

Thus, how can we determine whether a customer has a high affinity for our new product? We can only use an indirect way of reasoning. We assume those customers who have already bought the product (the buyers) to be representative of those who have a high affinity toward the product. Therefore, we search for customers who have not yet bought it (the non-buyers) but who are similar to the buyers in other respects. Our hope is that the more similar they are, the higher their affinity.

Our main challenge, therefore, is to identify customer properties that can help us to find similarities and that are available in the bank's data. Section 7.4 discusses what properties might be useful, and how to build data that reflect them. Assuming that we have good data, we can use a standard data mining method, namely binary classification, to try to differentiate between buyers and non-buyers. Trying to keep buyers and non-buyers apart in order to find their similarities may sound paradoxical; however, "difference" and "similarity" belong to the same scale. Therefore, it is crucial that our data mining algorithm be able to provide that scale. Fortunately, most algorithms can do that by delivering a *ranking* of customers, with higher-ranked customers being predicted to be buyers with higher confidence or probability than lower-ranked ones.

Thus, in what follows, a number of mining models will be developed that each deliver a ranking of non-buyers in which the top-ranked customers are those for which the model is most confident that they ought, in fact, to be buyers (if only they knew it!). We will also see how to decide which model is most useful. We can then serve the marketing department by delivering the top 20 percent of non-buyers from our final ranking. Finally, in the last part of this chapter, we will discuss how to apply the same methodology to other business tasks.

7.2 Business Understanding

The purpose of the CRISP-DM Business Understanding phase is to thoroughly understand the task at hand from the perspective of the business users: what terminology do they use, what do they want to achieve, and how can we assess after the project whether its goals have been met?

Thus we conduct a number of interviews with our bank's business experts and learn the following:

- The bank now has four types of current (checking) accounts on offer, differentiated by their internal names CH01 through CH04. CH04 is the new type that is to be pushed by our marketing action. Basically, each type of account comes with certain fixed monthly fees and interest rates for credit and debit amounts, but some customers can

have deviating rates or can be freed from the monthly fee due to VIP status or other particularities.

- A customer can have any number of current accounts, including zero, and any mix of types.
- Accounts have an opening and an ending date. When an account ends, its balance is zero and no further money transaction can occur. An open account whose ending date has not yet passed is called *active* and a customer with at least one active account is also called active.
- Each money transaction for every account is automatically classified by an internal text analysis system, based on an optional free-text form field that can be filled by the initiator of the transaction. There are many categories such as "cash withdrawal", "salary", "insurance premium", and so on, including an "unknown" category.
- For most customers, personal information such as date of birth or family status is known, but may not always be up to date.
- Customers can buy many other products from the bank, including savings accounts, credit cards, loans, or insurance. While data from these business branches is a valuable source of information, in our simplified example application, we do not include such data.

From this we quickly develop the idea of using information about the customers' *behavior*, derived from the money transaction data, to characterize our customers. We delve into this in Section 7.4 after taking a look at the available data sources in Section 7.3. For now we decide to exclude inactive customers from the analysis because their behavior data might be out of date. Thus, all active customers who have ever had a CH04 account are buyers, and all other active customers are non-buyers.

As stated above, eventually we want to send mail to about 20% of the non-buyers. In order to evaluate whether the mailing was successful, we can look at the sales rates of the CH04 account among recipients of the mail and other current non-buyers some time after the mailing. While this is an important part of our project, we will not discuss it further in this chapter as it involves no data mining-related techniques.

7.3 Data Understanding

Continuing our interviews with the bank's staff, we now turn to the IT department to learn about the available data. This phase of the CRISP-DM process is central to planning the project details, as we should never mine data whose structure and contents we have not fully understood. Otherwise, we run the risk of misinterpreting the results.

Not surprisingly, the bank has implemented a central data warehouse (DWH), that is, a data source that is separate from the operational information systems and provides an integrated view of their data, built specifically for analysis purposes. While data warehouses are not typically aimed at allowing data mining directly, data mining projects can benefit greatly from a well-designed data warehouse, because a lot of issues concerning data quality and integration need to be solved for both. From the perspective of a data miner, data warehouses can be seen as an intermediate step on the way from heterogeneous operational data to a single, integrated analysis table as required for data mining.

It is best practice [2] to use a dimensional design for data warehouses, in which a number of central fact tables collect measurements, such as *number of articles sold* or *temperature*, that are given context by dimension tables, such as *point of sale* or *calendar date*. Fact tables are large because many measurements need to be collected, so they do not store any context information other than the required reference to the dimension tables. This is the well-known star schema design.

We consider a particular star schema, depicted in Figure 7.1, for our example application; the data are provided as RapidMiner example sets in the material for this book. The central fact table holds money transactions, linked to three dimensions: calendar date, customer, and account.

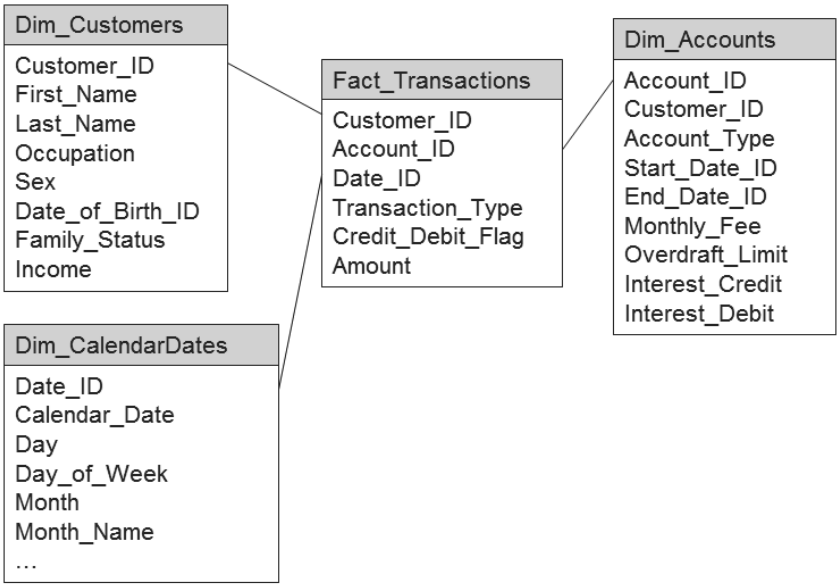

FIGURE 7.1: The source data schema.

Let us briefly examine the contents of these tables:

- Dim_Customers: One row for each customer. Includes personal data: first and last name, date of birth (as a reference to the Dim_CalendarDates table), occupation (one of ten internal codes, 00 through 09, whose interpretation is irrelevant for our application), sex (M or F), family status (single, married, or divorced, with a few missing values), and income. The income attribute is set to zero for young customers, but has many missing values for adults.

- Dim_Accounts: One row for each account, with a reference to the account owner (customer). Each account belongs to one of the types CH01 through CH04 and has a start date and an end date (both are references to the Dim_CalendarDates table). The monthly fee, the overdraft limit, and the interest rates for credit and debit amounts are stored individually with each account because, as we saw in the previous section, for some customers individual fees or rates may apply.

- Dim_CalendarDates: As recommended by [3], our data warehouse includes an explicit calendar date dimension rather than using date fields. In our case, it holds one row for each of the 50,000 days from January 1st, 1900 through November 22nd, 2036,

plus one row with ID 100,000 that represents "infinity" (December 31st, 9999). This last row is used for the end date of accounts whose end date is not yet fixed (active accounts). Because the `Date_ID` attribute numbers the days consecutively, we can use it to compute the number of days between two given dates directly without needing to join to this table, as long as we do not use the special date "infinity". The columns of this table can be used in reporting or other applications to qualify a given calendar date; we won't need most of them for our use case.

- Fact_Transactions: For each account, all money transactions that have ever been made are recorded in this table. A reference to the account owner (customer) and the transaction date is stored with each fact (transaction). The type of transaction holds categories like "salary" that are found automatically as explained in Section 7.2 (our sample data includes only a rather small number of categories). The credit/debit flag (values *CR* and *DR*) tells us whether money was taken from the account or booked into it, while the amount attribute gives the amount of money transferred, which is always a positive value.

We should be aware that the data for this example application is simplified, compared to real data, in at least the following ways:

- Our star schema does not accommodate "slowly changing dimensions" [2], a mechanism to keep old values of changed data together with a reference to the period of time during which these old values were current. For predictive data mining projects, such a mechanism can be valuable because it may be necessary to restore data values from past points in time, in order to learn from relationships between those values and subsequent developments. This is not needed for the application discussed here, however.
- Customers belonging to the same household are grouped together and analysed at the household level in many financial applications. In addition, a customer may not represent a natural person but a married couple or a corporate entity. These issues would need to be considered to find an appropriate definition of a mining example in a real application. However, our sample data includes only natural persons.
- Unlike a real schema, our star schema includes only a few, simple attributes without many distinct values.

These simplifications allow us to focus on the data mining issues of our task, yet still include some standard data preparation subtasks and a RapidMiner solution to them in the following section. You should easily be able to extend the methods that we discuss in this chapter to your real data.

7.4 Data Preparation

Virtually all data mining algorithms use a single table with integrated and cleaned data as input. Data Preparation is the CRISP-DM phase in which this table is created. The table should provide as much information about the examples (here, the customers) as is feasible, so that the modelling algorithm can choose what is relevant. Therefore, it is a good idea to bring background knowledge about the business into the data.

In this section we examine two RapidMiner processes; one to assemble the data from our

warehouse schema into a single example set (Section 7.4.1), and the second (Section 7.4.2) to perform some tasks that are specific to enabling data mining on the result.

7.4.1 Assembling the Data

Bringing data from the warehouse into a form suitable for data mining is a standard task that can be solved in many ways. Let us take a look at how to do it with RapidMiner, as exemplified by the first sample process from the material for this chapter, Ch7_01_CreateMiningTable. After opening the process in RapidMiner, you can click on Process, Operator Execution Order, Order Execution to display the order of operator execution; we will use these numbers for reference in the following.

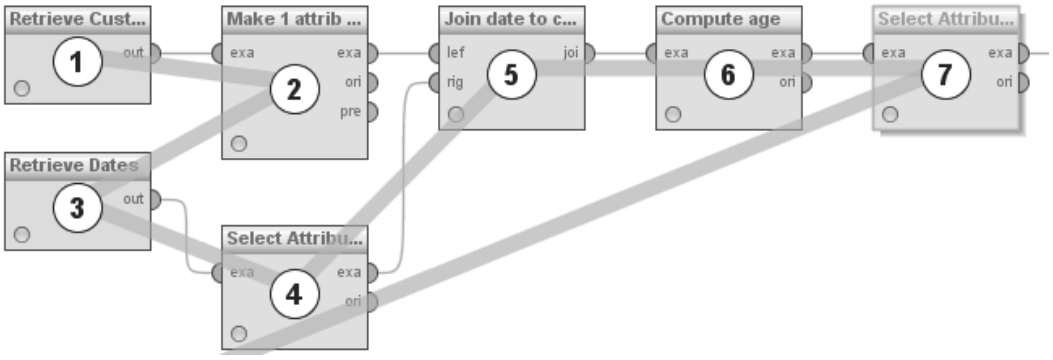

FIGURE 7.2: The substream in Ch7_01_CreateMiningTable for processing customer data.

Each of our four tables is used as one of the four inputs for the process. You could replace the four *Retrieve* operators (numbers 1, 3, 8, and 9) that read our sample data with operators like *Read Database* if you were using a real warehouse as your source. Each input is fed into a substream of our process consisting of a number of operators, and we will discuss the substreams in turn below. Figure 7.2 shows the customer substream. Figure 7.3 shows the last steps in this process, where our substreams are eventually combined into a single resulting example set.

One area of consideration in this phase is data types. RapidMiner associates each attribute with one of several types: real, integer, date, binominal, or nominal. The attributes in our four source datasets have an automatically determined type which may or may not be appropriate; ensuring that our resulting example set is correctly typed makes it easier for RapidMiner to support our data mining issues later on.

Substream: customer data: For example, the first operator we apply to our customer data (number 2 in the execution order, compare Figure 7.2) changes the type of the sex attribute to binominal, because we know (or can see from the example set statistics) that this attribute has exactly two distinct values, M and F. The operator *Nominal to Binominal*, which we use here, normally creates new, binominal attributes for each value of the input attribute, but in this case it recognizes that the input attribute already is binominal, so it just changes the type. (We have set the parameter *attribute filter type* to *single* because we only want to change one attribute.)

Next, we want to include a customer's current age into our data mining basis. In our application we assume that the data is current for December 31st, 2011, so we compute the current age as of this date. Because the customer table refers to the calendar date table in the birth date attribute, rather than using a date attribute, we join the calendar date table to the customer data using the *Join* operator (number 5) with the join type "left".

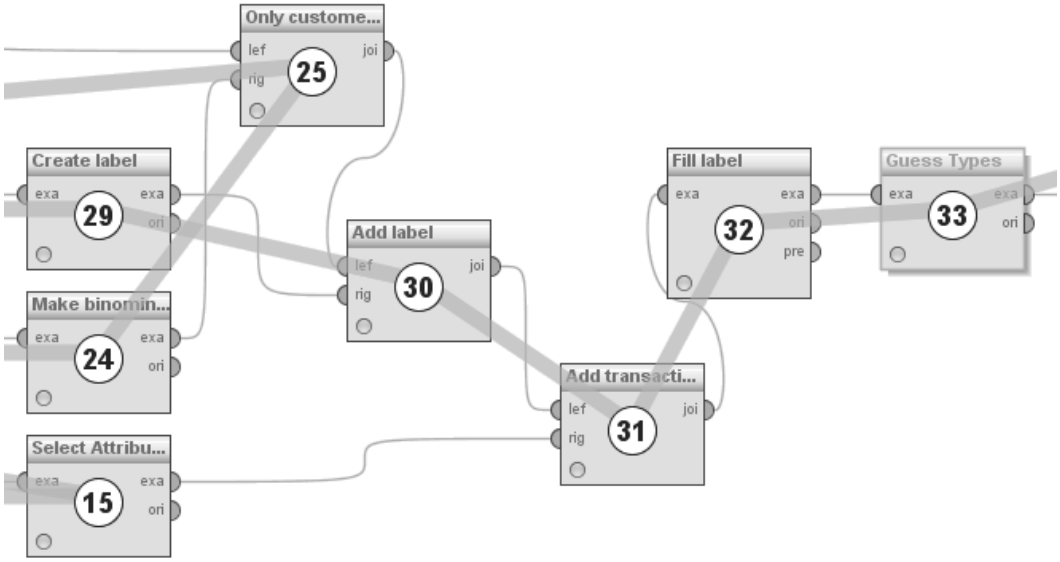

FIGURE 7.3: Last steps in the process Ch7_01_CreateMiningTable.

Our example sets have no attribute with the RapidMiner ID role yet, so we uncheck the option *use id attribute as key* in the *Join* operator; instead we choose the join key attributes explicitly. Note that in order to avoid taking over all the attributes from Dim_CalendarDates, we use the *Select Attributes* operator (number 4) before the join. After the join, we can compute a customer's current age (operator 6) by subtracting their year of birth from 2011: `year` is an attribute from the calendar data, so we set up the simple function expression `2011 - year` as a definition for the new attribute `current_age` in the *Generate Attributes* operator. Finally, we remove three attributes from the customer data (7), as they are useless for data mining because they nearly take on individual values for every customer: first and last name, and the birth date ID.

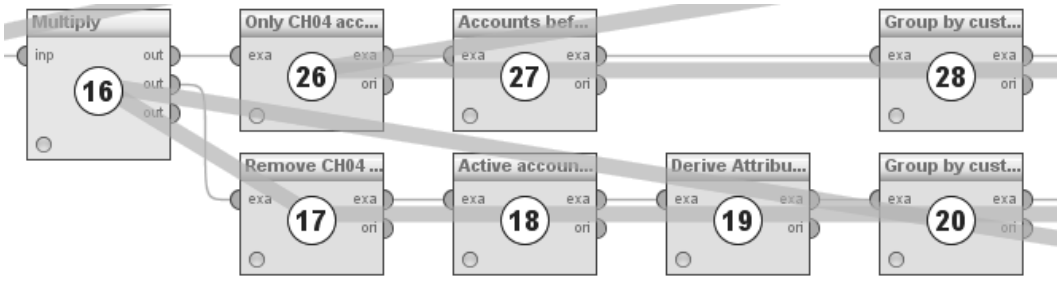

FIGURE 7.4: The first part of the substream in Ch7_01_CreateMiningTable for processing account data.

Substream: account data: Now we turn to the account data, see Figures 7.4 and 7.5. One important goal of this process is to label each customer as buyer or non-buyer; all owners of a CH04 account (as of December 31st, 2011) are buyers. We use the operator *Multiply* (number 16) to copy the input data into two streams, keeping only CH04 accounts (26) in one stream and dealing with all other accounts in the second stream, by applying the *attribute_value_filter*s `account_type = CH04` and `account_type != CH04`, respectively,

in two *Filter Examples* operators. Operator 27 leaves only accounts that have opened on or before 12/31/2011 in the stream (for simplicity, we use the ID of this date from our calendar dimension, which is 40907, directly for filtering). Note that we include inactive CH04 accounts as long as they were opened before 12/31/2011, because their owners have taken the decision to open a CH04 account, so they are buyers even if they have closed the account again in the meantime.

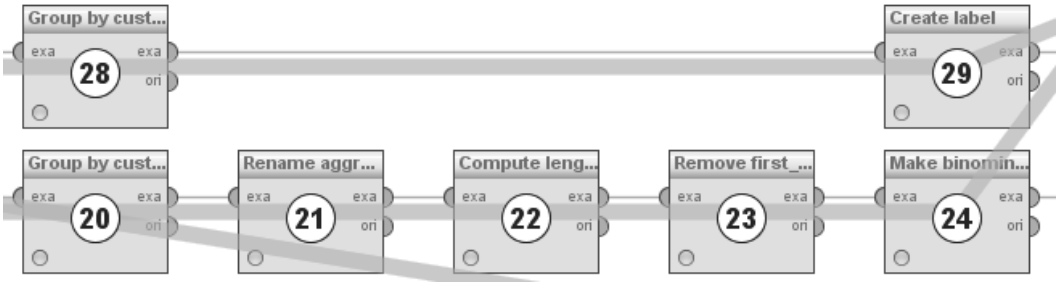

FIGURE 7.5: The second part of the substream in Ch7_01_CreateMiningTable for processing account data.

A customer can have more than one account, even more than one CH04 account, so we group by customer ID with the *Aggregate* operator (28). The resulting example set has only one attribute, the customer ID; we do not need any aggregation attributes since all customers in this substream are buyers. Thus we can now add the label attribute, `is_buyer`, by setting its value to *true* for all customers in this stream (operator 29, another *Generate Attributes* operator).

In the second stream that processes the account data, we handle all other account types (CH01 through CH03, operator 17), and here we keep only active accounts (18). The aim of this substream, as well as of the substream that deals with the transaction data, is to gather information about the customer's behavior (see Section 7.2).

Excursion: Creating the label: There is an important point to be discussed here. Why do we not include (active) CH04 accounts in the data from which we derive our behavior indicators? After all, customers display their behavior in all four types of accounts. But the CH04 accounts make a customer a buyer, so there may be some behavior information in them that betrays the label. Our overall goal is to predict which non-buyers could be most easily persuaded to become buyers, so when we apply our data mining model to the non-buyers, no CH04-based information will be available. Therefore, our model should be trained on data that does not include CH04-based information either (other than the simple buyer/non-buyer distinction itself). Otherwise it might discover biases in the data that result from the fact that the buyers already have bought a CH04 account, and such biases cannot hold for the non-buyers. Indeed, if you are going to implement a similar project using real data, you should be extremely careful to avoid including information in your mining base that is only available because the label is already known for the training data; it can be quite difficult to discover the hidden links between the label and other data sources due to the complexity of your business. The problem is that even evaluating your model on test data does not necessarily tell you that it is too good to be true, if these effects are small.

Indicators for customer behavior: We now return to the question of how to represent customer behavior in our mining base. Candidate attributes are the number of active accounts, the average duration that accounts have been active, whether a customer seems to be willing to pay a fee for their accounts, and, of course, the type and volume of money transactions that have been shifting money from and to their accounts. In a real business,

you should be able and willing to think of many more such "derived" attributes that may indicate, however indirectly, what kind of customer someone is. You can let the data mining algorithm decide whether an attribute is useful, but if you do not construct the attribute in the first place, you may lose an opportunity to improve your model. In fact, including more information in your data mining basis is probably the single most promising approach if you do not get good results in your first data mining runs.

ExampleSet (2663 examples, 0 special attributes, 14 regular attributes)

account_id	customer_id	fee_indicator	ch01_indicator	ch02_indicator	ch03_indicator	a
63	59	0	0	0	1	
65	61	0	0	0	1	
66	61	1	0	1	0	
67	62	1	0	1	0	
68	63	1	1	0	0	
69	63	1	1	0	0	
70	64	0	0	0	1	

FIGURE 7.6: Indicator attributes at the account level.

The next step in our example process, therefore, is to compute a few attributes at the account level (operator 19) before aggregating them to the customer level (20). In particular, we compute binominal indicator attributes (values 0 and 1) to determine whether an account is associated with a monthly fee or if it is of type CH01, CH02 or CH03; see Figure 7.6. We use the *Generate Attributes* operator (19) with if-expressions (whose second argument is returned if the first argument, a Boolean condition, is true, while the third argument is returned otherwise) to do so. For example, the function expression

```
if(account_type == "CH01", 1, 0)
```

creates the binominal attribute `ch01_indicator` shown in Figure 7.6.

ExampleSet (2385 examples, 0 special attributes, 8 regular attributes)

customer_id	maximum(fee_indicator)	sum(ch01_indicator)	sum(ch02_indicator)	sum(ch03_indicator)
59	0	0	0	2
61	1	0	1	1
62	1	0	1	0
63	1	2	0	0
64	0	0	0	1

FIGURE 7.7: Indicator attributes at the customer level (after applying the *Aggregate* operator).

Summing up these indicators to the customer level (Figure 7.7) gives us, for instance, the counts of each type of account, while computing the maximum preserves the binominal character of the indicator, like in our fee indicator. The *Aggregate* operator is used for this, with `customer_id` as the only *group by attribute*. (The RapidMiner types of these generated attributes must be explicitly set, for which the operators *Numerical to Binominal* (24) and *Guess Types* (33) are used).

To compute the account duration (19), we exploit the fact that the date IDs in our calendar dimension are consecutively numbered, so that the difference between two IDs yields

the number of days between the corresponding dates. The number of years a customer has been with our bank now—the duration of the customer relationship—is computed similarly (22), using the earliest start date of any account as the start of the relationship.

Substream: transaction data: Our last data source is the transaction data (see Figure 7.8). The first operator applied to it (10) implements an application-specific decision to consider only transactions from the last year (2011 in our example) as a basis for behavior-indicating attributes, for fear of older data being no longer representative of a customer's behavior. The second operator (11) sorts the transaction amounts into new attributes according to their transaction type. For example, the function expression

```
if(transaction_type=="CreditCard", amount, 0)
```

is used to compute the attribute `creditcard_last_year`, which holds the value of the `amount` attribute for all rows in which the transaction type is *CreditCard*, and 0 for other rows. This step is simply a preparation for the aggregation (operator 12) to the customer level, because we want to have separate attributes with the number of transactions, or the (total and average) amount of money involved in such transactions, for our relevant transaction types. From a business perspective, these attributes are very important because we hope that they reflect how the customers handle their money. As mentioned above, many more such attributes can and should be invented in a real application.

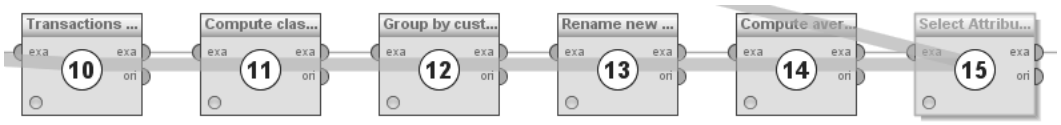

FIGURE 7.8: The substream in Ch7_01_CreateMiningTable for processing transaction data.

Joining the substreams: See also Figure 7.3. Joining the active accounts to the customer example set by an inner join (25) leaves only active customers in our stream. The next join is a left join to add the label to our data (30); it results in missing values in the label attribute `is_buyer` for all non-buyers, which is remedied by the operator *ReplaceMissingValues* (operator 32, called "FillLabel" in our process), with *false* as the replenishment value. By left joining our active customers with the transaction-based data (31), we complete our assembly of a single example set with diverse, information-rich attributes, so we store the result in our RapidMiner repository (34).

7.4.2 Preparing for Data Mining

Having illustrated how RapidMiner can be used for data assembly, we now turn to data mining-specific issues in our next example process, Ch7_02_PrepareDataForMining. It uses the output of the previous process and applies a simple chain of operators to it that make the data ready for modelling.

Most importantly, using the operator *Set Role* we tell RapidMiner that our attribute `is_buyer` contains the label that is to be predicted. We also declare `customer_id` to be an ID attribute, making RapidMiner ignore it during modelling, since no similarities or patterns can be based on an attribute that takes an individual value for each example.

Data cleaning: Modelling operators usually cannot handle missing values, so we must decide on a strategy to deal with the missing values in the `income` and `family_status` attributes. We choose to fill them with a standard value, which is the new value *unknown* for the family status and simply zero for `income` (Operator *Replace Missing Values*). A simple strategy like this is sufficient in many applications, but also provides one of the many

options we have for trying something different if our modelling results are unsatisfactory. Other strategies include deleting customers with missing values, filling up with the most frequent or average value, and training extra models that predict how to fill, using the RapidMiner operator *ImputeMissingValues*.

We include the operator *Remove Useless Attributes* in our process; it is a useful cleaning operator that can remove attributes whose values are the same for almost all examples. Its parameters must be adjusted to the data at hand. In our case it would remove any nominal attribute whose most frequent value occurs in less than 1% or more than 99% of examples, because we set the parameters *nominal useless above* and *nominal useless below* to 0.99 and 0.01, respectively, but no such attributes exist in our sample data.

Discretization: The remaining steps discretize some of the numeric attributes. They are optional in the sense that modelling can be done without them, as long as the modelling algorithm can deal with numeric data, but they are useful for bringing some background knowledge into the data, which can improve modelling results.

A classic example that we also use here is the discretization of the age attribute into age groups like kids, adults, or pensioners. The age intervals for these groups are based on real-world knowledge or business rules, rather than statistic criteria as might be employed by an automatic discretization method. Similarly, in our example process we discretize income and the duration of the customer relationship by user-specified intervals. The operator *Discretize by User Specification* accepts an arbitrary number of intervals, each specified by its upper limit, which is included in the interval, implicitly including "minus infinity" as the lower limit for the first class.

In contrast, the operator *Discretize by Binning* that we apply to our monetary attributes automatically creates a number of intervals that are specified by a parameter, by equally dividing the range of values. You can easily follow this behavior, as well as that of other discretization operators, by looking at the meta data view of our attributes before and after the discretization.

In general, however, it would be difficult to give recommendations as to whether discretization should be used at all, or which type of discretization would be most helpful in a particular application. It is worth tinkering with discretization if the modelling results do not meet your expectations, but there is no guarantee that this will lead to improvement.

7.5 Modelling and Evaluation

Our data is prepared for data mining now, so let us prepare ourselves too, by briefly discussing a couple of methodological issues.

7.5.1 Continuous Evaluation and Cross Validation

The CRISP-DM process mentioned in the introduction formally distinguishes between the process stages *modelling* (other terms are learning, training, mining, building a model) and *evaluation* of the model. However, in practice an unevaluated model is useless, so the two stages are intertwined. We will evaluate each model that we build until we have found a model with satisfactory quality. Of course, the datasets used for modelling on the one hand and evaluation on the other must always be distinct.

One standard way of evaluating/validating a model is cross validation, which is directly supported by the RapidMiner operator *X-Validation*. It takes the available data and ran-

domly divides it into a parameter-specified number of distinct subsets. Each subset then serves as a test set for a model that has been trained using the remaining subsets. Thus, as many models are created as there are subsets, and each is evaluated on one subset. Afterward, the evaluation results are averaged. This scheme ensures that we do not overestimate the quality of our model, which might happen if we accidentally choose a single test set with favourable data.

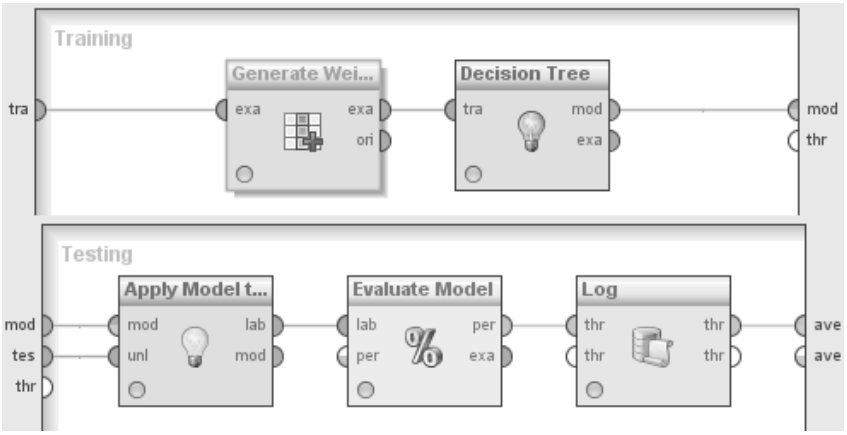

FIGURE 7.9: Inside the RapidMiner *X-Validation* operator.

The *X-Validation* operator is a nesting operator as two RapidMiner sub-processes can be embedded into it, one for training the model and one for testing/evaluating it. Double-clicking on the operator shows the two sub-processes and their expected in- and output (see Figure 7.9): labelled data as input, a model as output of the training sub-process and input of the testing sub-process, and a performance evaluation result as the testing output. Each sub-process is executed by the operator as many times as is specified by the parameter *number of validations*. More details will follow below.

7.5.2 Class Imbalance

Our application displays a rather common characteristic. The distribution of the values of our label attribute `is_buyer` is unbalanced: there are many more non-buyers than buyers; only around 7% of our customers are buyers. This can mislead many data mining algorithms into disregarding the buyers, because a high accuracy (ratio of correct predictions) can be achieved by simply predicting all customers to be non-buyers. There are, at least, two possible remedies for this: weighting and re-balancing.

Weighting introduces a special attribute into the data that RapidMiner recognizes by its role. It is a numeric attribute that associates each example, in this case the customer, with a weight factor, allowing more weight to be given to the minority label, in this instance the buyers. Some, but not all modelling operators can account for this weight factor and change their prediction behavior, accordingly. The RapidMiner operator *Generate Weight (Stratification)* can be used for the automatic creation of a weight attribute, which is demonstrated in the example process Ch7_03_XValidateWeighted (Figure 7.9). It distributes a given total weight over the examples in the input, such that for each label, the sum of the weights of all examples with this label is equal. In our example, both the buyers and the non-buyers receive a total weight of 5,000 because we set the parameter *total weight* to 10,000 (have a look at the new weight attribute in the output of this operator).

Re-balancing creates a special subset of the data in which both labels (buyers and non-buyers) occur with roughly equal frequency. This subset is used only for training/creating the model, but not for testing or deploying it. This approach is shown in the example process Ch7_04_XValidateBalanced. The only change is that the *Sample* operator replaces *Generate Weight (Stratification)* in the training sub-process. We use it to randomly draw 10% from the non-buyers in the training data (i.e., a fraction of 0.1, to be set as ratio for the class *false*), but all buyers (a fraction of 1.0, the ratio for *true*), resulting in a roughly even distribution of buyers and non-buyers.

Good results have been achieved in real applications with both approaches, so it is usually worth trying which works best. On our fictitious, randomly created sample data, re-balancing gives better evaluation results with a decision tree model while the author has frequently found the weighting scheme to be more successful in his real projects.

7.5.3 Simple Model Evaluation

Let us now take a first look at model evaluation: inside our *X-Validation* operators, the models that are returned by the training sub-processes are used to make predictions on the test data by applying the operator *Apply Model* (compare Figure 7.9). This operator creates (among others which are discussed below) an attribute with the predicted label, called `prediction(is_buyer)` in our case (insert a breakpoint after *Apply Model* if you want to see it). In order to see how well the predicted label matches the true label in our test data, we include the operator *Performance (Binominal Classification)*, which is specialized for evaluating applications with a binominal label like ours. It automatically creates a contingency table and several useful metrics, depending on the selected parameters.

Multiclass Classification Performance ○ Annotations

◉ Table View ○ Plot View

accuracy: 72.70% +/- 10.62% (mikro: 72.70%)

	true false	true true	class precision
pred. false	1661	111	93.74%
pred. true	540	73	11.91%
class recall	75.47%	39.67%	

FIGURE 7.10: A RapidMiner contingency table.

Compare the contingency tables created by the two example processes Ch7_03_XValidate-Weighted and Ch7_04_XValidateBalanced (the latter is shown in Figure 7.10); each is the result of adding up all contingency tables created by the *X-Validation* operator in each process. Similarly, other performance metrics yielded by *X-Validation* are averages of its individual runs. Inspecting these individual runs can be done with the help of a *Log* operator as shown in our processes (Figure 7.9). The *Log* operator is executed in each testing sub-process; it adds the values listed in the *log* parameter to the end of the specified file in each run. The elements of the contingency table have to be specified individually within the *Log* operator using the pre-defined categories *false_positives*, *false_negatives*, and so on—note that these categories are only available in the *Log* operator if they have been checked in the *Performance* operator.

Two important evaluation metrics for applications like ours are recall and precision of the prediction of buyers (i.e., of the positive class). They can be read directly from the contingency table (column "true true" and line "pred. true"). Typically, we cannot get high

values for both of them. Instead, as we find models that increase one of these values, the other one is usually decreased. So which metric is more important? The answer depends on your application. In our marketing application, the goal is to find non-buyers that are similar to the buyers. So the set of customers predicted to be buyers should include a lot of real buyers, and must not be too small, which it is likely to become if we aim for high precision on real data. Therefore, we should put more emphasis on achieving a high recall value; at the same time, the precision value must remain significantly higher than it would be if our prediction was random, which means that it must remain higher than the overall ratio of positive labels (7% in our sample data). With real data, we might not be able to achieve precision values higher than two or three times this ratio, depending on the application, at hand. This would still mean that our predictions are two or three times as good as random predictions, and would not prevent a useful deployment of the model.

7.5.4 Confidence Values, ROC, and Lift Charts

But we can do deeper evaluations of our models by looking beyond recall and precision, to obtain metrics that can directly be translated into business terms like costs and expected returns of our deployment. To see how, insert a breakpoint after the *Apply Model* operator and look at the special attributes `confidence(true)` and `confidence(false)` that it creates. They reflect a numeric confidence or probability (depending on the type of model) with which our model makes its predictions. You see that our prediction in `prediction(is_buyer)` is *true* whenever `confidence(true)` takes a value of 0.5 or higher.

You might have the idea of changing this threshold from 0.5 to a lower or higher value. You can manually do so using the RapidMiner operators *Create Threshold* and *Apply Threshold*, which we will discuss below in Section 7.6. As you increase the threshold, the set of customers predicted to be buyers becomes smaller and vice versa. If our model is any good, a higher threshold should also result in a higher precision.

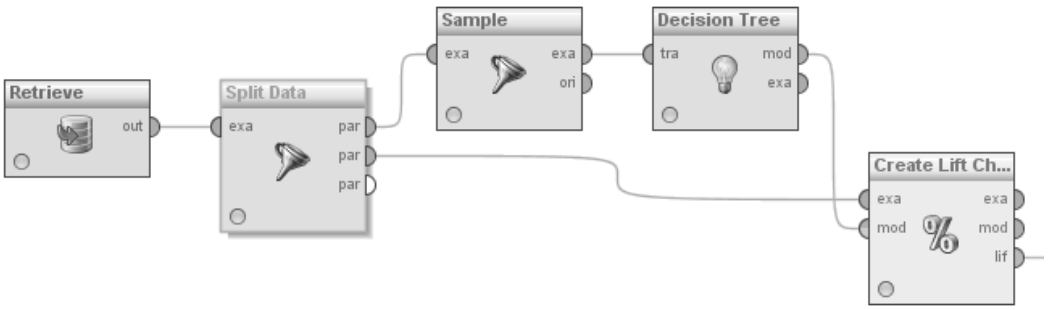

FIGURE 7.11: A RapidMiner process for creating lift charts.

Lift charts: Lift charts quantify this relation by showing the increase in the number of customers that are predicted to be buyers, and also the increase in the real buyers among them, as the threshold is decreased—see Figure 7.12. From the material for this book, check the process Ch7_05_LiftChart, shown in Figure 7.11, to see how to apply the RapidMiner operator *CreateLiftChart*, which was used to create Figure 7.12. In this process, the data is split evenly into a training set and a distinct test set to which *CreateLiftChart* is applied using a model that is built on the training set (the *Sample* operator is used to make the training data balanced, see Section 7.5.2). The chart is drawn by creating a number of bins for the confidence values associated with the specified *target class* (label), so we enter the value *true* for this parameter (we want to examine the lift for buyers). The bars indicate

for each bin the number of examples whose confidence value is higher than the threshold for this bin *and* which belong to the target class. The count to the right of the slash gives the full number of examples whose confidence exceeds the value for the bin, while the thin line shows the cumulated counts.

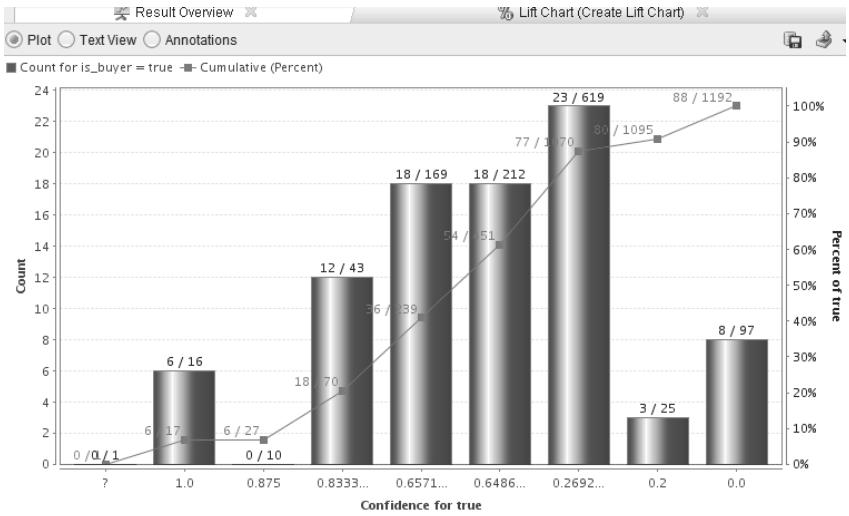

FIGURE 7.12: A lift chart created with RapidMiner.

The lift chart is useful for response-oriented marketing applications because the cumulated counts give us an idea of how many customers must be targeted in a campaign in order to receive a certain number of responses. For example, based on the chart in Figure 7.12, 451 customers would have to be addressed in order to find 54 buyers among them—note that this ratio of buyers, thanks to our model, is higher than the 7% we could easily get without data mining, by randomly choosing 451 customers (though the increase in this ratio is not great on our fictitious data). However, in this scenario, we wish to address only non-buyers, so we use the lift chart mainly to confirm that our model behaves as expected. In contrast, if we were to target both buyers and non-buyers because we wanted to sell a different, brand-new product that nobody has bought yet, and if we hoped that the buyers would tend to buy more readily again, the lift chart would be more useful. This corresponds to the classic scenario where previous responses to campaigns are used to predict future responses [4].

ROC charts: A similar argument holds for ROC charts [5], which are provided by the RapidMiner operator *Performance (Binominal Classification)* if the *AUC* parameter is selected. Figure 7.13 shows a ROC chart obtained on real data. The horizontal axis shows the ratio of non-buyers wrongly predicted as buyers to all non-buyers (*false positive rate*). So, as you go from left to right, more and more non-buyers are predicted to be buyers, which can only happen if the confidence threshold is lowered, which is indeed shown by the blue line (named "(Thresholds)") for which the vertical axis indicates the threshold value. The red line uses a different vertical axis (but the same horizontal one), namely, the recall for the positive class (buyers), also called *true positive rate*. As the threshold is lowered, more and more true buyers are in the set of customers predicted as buyers, so the recall increases; but the earlier it does so, the better the model. A random model would lead to a red line that follows the diagonal from lower left to upper right, as you can indeed (almost)

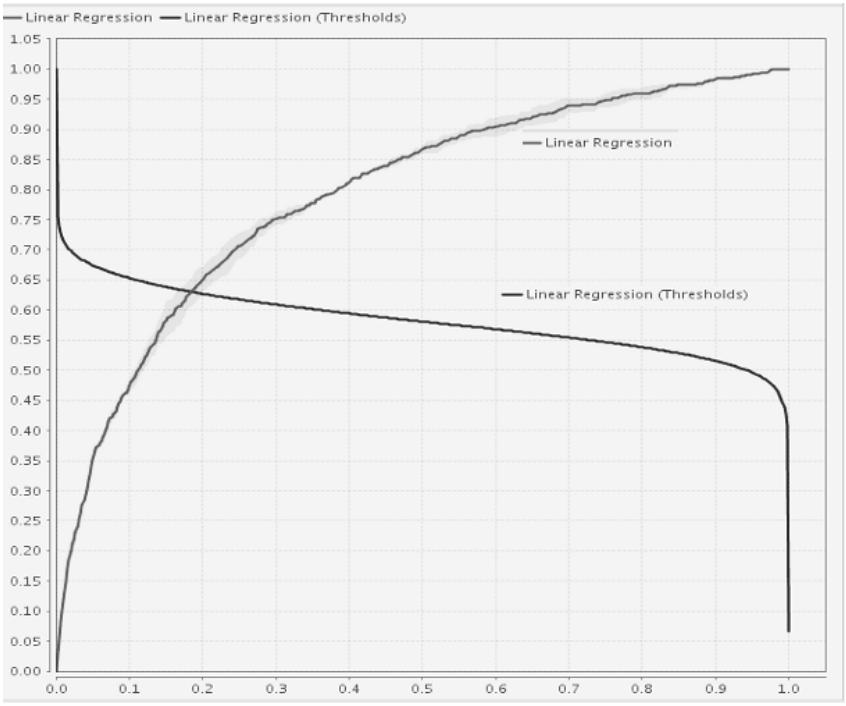

FIGURE 7.13: A ROC chart created with RapidMiner on real data.

see in ROC charts created on our fictitious, randomly created data. So the more the red line deviates from the diagonal, the more powerful our model. This deviation is measured by the area between the line and the diagonal, a metric called area-under-the-curve (AUC). It is a useful metric for comparing the qualities of different models, a task to which we turn next.

7.5.5 Trying Different Models

So far we have left the core of data mining aside in our discussion—the actual creation of a model. You noticed, of course, the *Decision Tree* operator in the last few example processes that we discussed. It was applied to the training datasets, and created a decision tree that we used as input for *Apply Model* in order to get actual predictions and confidence values. This operator may well be used in a real application, but for us it serves mainly as a placeholder for the large number of data mining operators that RapidMiner provides, many of which could have been used instead. For more information about decision trees and other data mining algorithms see, for example, [6].

Type compatibility: Some mining operators come with restrictions regarding the type of data they can be applied to. For example, most of the function fitting and neural net operators need numeric data, i.e., they cannot be applied to data with nominal attributes. You can see this in RapidMiner in the operator information under "capabilities". In addition, RapidMiner warns you if you attempt to apply an operator to incompatible data, which is why we ensured correct data types in our data preparation process (see Section 7.4). But it is usually not difficult to transform the data so that its attributes have the right types, and

in fact, frequently RapidMiner can automatically determine what kind of transformation to make and suggest it as a quick fix.

To see an example for type conversion, take a look at the example process Ch7_06_CompareMiningMethods. Its only top-level operator, *Compare ROCs*, performs a cross validation on each of its inner data mining operators and draws a ROC curve for each of them into the same chart. Thus it serves to compare any number of mining methods on the same data at one glance. However, if some of these methods are incompatible with the given data, type conversions have to be done. In our example, the operator *Nominal to Numerical* is applied inside *Compare ROCs*, to prepare the data for *Linear Regression* and *Logistic Regression*, two mining operators that need numeric data. With default settings, *Nominal to Numerical* replaces a nominal attribute with an integer attribute, using integers to code the nominal values. For example, a nominal attribute `colour` with values *red*, *green*, and *blue* is replaced by a like-named numeric attribute with values 0, 1, and 2. You can change this behavior to dummy coding by setting the parameter *coding type* accordingly. Dummy coding creates new attributes with the values 0 and 1 for each value of each nominal input attribute, so it would create the attributes `colour=red`, `colour=green`, and `colour=blue` in our example. Their values are 1 for all examples where the input attribute `colour` takes the value corresponding to the respective new attribute, and 0 otherwise. Effect coding uses -1 instead of 0.

There are several other types of conversion operators available in RapidMiner for various purposes, providing the flexibility for setting up mining schemes on rather diverse data.

Data mining parameters: Almost all data mining operators have parameters that specify details of their learning behavior. For example, the decision trees that the operator *Decision Tree* produces can be constrained to a maximum depth (parameter *maximal depth*), or to have a minimum number of examples in any leaf (parameter *minimal leaf size*). Further, how a tree is built can be influenced by parameters like *minimal size for split* or *minimal gain*. It is the main goal during modelling and evaluation to decide not only on a suitable data mining operator, but also to find settings of its parameters that yield good results. This is supported by the *Optimize Parameters* operators, which automatically search through different settings, as described elsewhere in this book.

7.6 Deployment

In the previous sections we have described how to prepare our data and how to try and evaluate different preparation methods, data mining models, and parameter settings. In a real project, probably quite a few variants of our processes would have been tested until a decision on a final preparation process, mining method, and parameter settings was made, based on the recall and precision or AUC values that could be achieved. Now we finally want to find the 20% of non-buyers that are the most promising target for our marketing campaign. This is the last CRISP-DM phase, called deployment.

Our example process Ch7_07_Deployment assumes that we decided on the balancing scheme to deal with class imbalance and on the *Decision Tree* operator with certain parameter settings as our mining algorithm. You can see that we do not take one of the decision trees constructed during our evaluation experiments, but that we build a final decision tree for deployment, using all the available data for training it, rather than keeping some test data apart for evaluation. Note that while our balancing scheme reduces the amount of data used for training, in general we can use all data for the final model. The tree is stored

in the RapidMiner repository for reference and documentation purposes, using the *Write Model* operator. It is then applied to all our non-buyers (and only the non-buyers). The *Performance* operator shows us that in this way, some 40% of non-buyers are predicted to be buyers.

Those 40% are more than we want, but remember that behind these predictions there is a default threshold value of 0.5 that is applied internally to the ranking of customers that is induced by the confidence values produced by *Apply Model* (Section 7.5.4). We can easily choose a different confidence threshold. The process Ch7_08_Deployment_Thresholds shows a variant of the previous process in which the threshold is set manually. In this variant, the parameterless operator *Apply Threshold* is applied to the output of *Apply Model*, using the threshold created by a *Create Threshold* operator. We need to tell the *Create Threshold* operator which values of our label correspond to the confidence values 0 (parameter *first class*, which must be set to *false* in our process) and 1 (*second class*, to be set to *true*).

We can try different thresholds to find one that comes as close to our desired result as possible, given the model. The final example set returned by this process lists all non-buyers, with a ranking of who should be most likely to open a new CH04 account in the `confidence(true)` attribute.

7.7 Conclusions

In this chapter, we have developed a marketing application for data mining with the aim of advertising a new product to those customers with a high affinity for it. We have seen how to find and assemble suitable data, how to test and evaluate different mining algorithms on it, and how to deploy a final model to our customer base. We have introduced several RapidMiner operators and processes that support all of these stages and provide a reference as well as starting points for your own applications and experiments. In addition we have clarified a number of methodological issues in the hope of enabling you to confidently realize your own data mining projects in due course.

We have used the buyers of our product of interest as "model customers" for finding similar customers among the non-buyers. This is a common approach in other use cases as well. For example, in churn prediction, customers who have churned—discontinued their contract with our organisation—serves to find other would-be churners in a similar way. Analogous to our use case, churners and non-churners form two groups for binary classification, based on which ranking of non-churners can be created that reflects their propensity to churn. In fact, one would be able to directly re-use several of the processes presented in this chapter for churn prediction, which illustrates the generality and flexibility of our methods and the RapidMiner software.

Glossary

Accuracy In supervised learning, the ratio of the number of correctly predicted examples to the number of all examples in a test set.

Area Under the Curve (AUC) The area between a ROC curve and the diagonal line in a ROC chart (see ROC).

Contingency Table A table that displays the number of correctly and incorrectly predicted examples, for each label, from a single model application to a test dataset. For binary classification, this results in four cells with the numbers of true positives, false positives, true negatives, and false negatives, respectively.

Cross Validation An evaluation scheme for supervised learning that uses all of the available labelled data. The data are divided into a given number of subsets and a model is tested on each subset after training it on the remaining subsets. In the extreme case of using as many subsets as there are examples, this is called leave-one-out. The individual evaluation results are averaged to get an overall result that is statistically more reliable than would have been a single result gathered from a single test set.

Data Cleaning Good data quality is a prerequisite for data mining, but even the best real-world data usually contain some omissions or errors. During Data Cleaning, which is one stage in the Data Preparation phase (see separate entry), erroneous data values are repaired where possible, and missing values are filled (unless it is decided to simply delete any examples with erroneous or missing values). For filling missing values, several methods exist, like inserting a distinct value or the most frequent or average value. As a more refined approach, a separate model can be learned for each attribute with missing values; such a model predicts how to fill the missing value based on the present values of other attributes, for each example.

Data Preparation Preparing data for mining involves a number of technical tasks: bringing the relevant data into a single table that has one row for each example, creating the label attribute (for supervised learning tasks), cleaning the data (see Data Cleaning), and converting the data types to make them suitable for the chosen data mining method. For some mining methods, for instance association rule mining, quite specific input formats are needed, whose creation is also part of Data Preparation. But apart from these technical tasks, there is also the challenge of finding or creating attributes that give as much information about the examples as possible, perhaps including some background knowledge from the business domain. For example, it may help to explicitly create attributes that reflect certain trends in the data, like increase/decrease in revenue, rather than hoping that the mining algorithm will discover or use such trends by itself.

Deployment The final phase in a data mining project when a trained model is applied to some dataset as part of a business process.

Example The representation in data of a real-world object that exemplifies the phenomenon of interest for a data mining application. A collection of examples is needed in order to find patterns or relationships among them.

False Negatives In binary classification, the number of positive examples in a test dataset that have been predicted to be negative.

False Positives In binary classification, the number of negative examples in a test dataset that have been predicted to be positive.

False Positive Rate In binary classification, the ratio of the number of negative examples predicted to be positive (false positives) to the number of all negative examples.

Precision The ratio of the number of *correct* predictions of a given label to the number of predictions of that label in a test dataset. In binary classification, there are two precision values, one for the positive and one for the negative label. Precision for the positive label is thus defined as the ratio of true positives to all examples predicted to be positive in a test dataset.

Recall The ratio of the number of *correct* predictions of a given label to the number of all examples with that label in a test dataset. In binary classification, there are two recall values, one for the positive and one for the negative label. Recall for the positive label is thus defined as the ratio of true positives to all positive examples in a test dataset.

Receiver Operating Characteristic (ROC) In binary classification, a chart that

plots model evaluations (one point in the chart for each evaluation), using the false positive rate for the horizontal axis and the true positive rate for the vertical axis (usually both expressed as values between 0 and 1). It is common to evaluate the same model based on different thresholds for the minimum confidence value that leads to a positive prediction, so that each threshold leads to one point on the ROC curve. Random predictions would lead to a ROC curve that is identical to the diagonal line from (0,0) to (1,1). The area between the ROC curve for a given model and this diagonal line is a measure for the model's quality called area under the curve (AUC).

Training The application of a data mining algorithm to a dataset in order to create (learn) a data mining model, like a decision tree, a neural net, or a regression function.

True Negatives In binary classification, the number of negative examples in a test dataset that have been predicted to be negative.

True Positives In binary classification, the number of positive examples in a test dataset that have been predicted to be positive.

True Positive Rate The same as recall for the positive label: the ratio of true positives to all positive examples in a test dataset (for binary classification).

Bibliography

[1] P Chapman, J Clinton, R Kerber, T Khabaza, T Reinartz, C Shearer, and R Wirth. CRISP-DM 1.0. Technical Report, The CRISP-DM Consortium, 2000.

[2] R Kimball, M Ross, W Thornthwaite, J Mundy, and B Becker. *The Data Warehouse Lifecycle Toolkit,*. 2nd edition. John Wiley & Sons, 2008.

[3] R Kimball and M Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling,*. 2nd edition. John Wiley & Sons, 2002.

[4] G Linoff and M Berry. *Data Mining Techniques: For Marketing, Sales, And Customer Relationship Management,*. 3rd edition. Wiley Publishing, 2011.

[5] T Fawcett. ROC Graphs: Notes and Practical Considerations for Data Mining Researchers. Technical Report, HP Laboratories. 2003.

[6] M North. *Data Mining for the Masses.* A Global Text Project. Global Text Project, 2012.