**Mining Binary Constraints in the Construction of Feature Models**

**A Feature-Oriented Approach to Mining Binary Constraints between Requirements**

Authors Name/s per 1st Affiliation *(Author)*

line 1 (of *Affiliation*): dept. name of organization

line 2: name of organization, acronyms acceptable

line 3: City, Country

line 4: e-mail: name@xyz.com

Authors Name/s per 2nd Affiliation *(Author)*

line 1 (of *Affiliation*): dept. name of organization

line 2: name of organization, acronyms acceptable

line 3: City, Country

line 4: e-mail: name@xyz.com

*Abstract*—Feature models provide an efficient way to organize and reuse requirements in a specific domain. A feature model consists of a feature tree and cross-tree constraints. Identifying features and then building a feature tree takes a lot of effort, and many automated or semi-automated approaches have been proposed to help the situation. However, finding cross-tree constraints is often a more challenging task which still lacks the help of automation. In this paper, we propose an approach to mining cross-tree binary constraints in the construction of feature models. Binary constraints are the most common and basic kind of cross-tree constraints which involve exactly two features and can be in turn classified into two sub-types, i.e. *requires* and *excludes.* Given these two sub-types, a pair of any two features in a feature model falls into one of the following classes: no constraints between them, a *requires* constraint between them, or an *excludes* constraint between them. Therefore we perform a 3-class classification on feature pairs to mine binary constraints from features. We incorporate a support vector machine as the classifier and utilize a genetic algorithm to optimize the classifier. We conduct a series of experiments on two feature models constructed by third parties, to evaluate the effectiveness of our approach under different conditions that might occur in practical use. Results show that we can mine binary constraints at a high recall (near 100% in most cases), which is important because finding a missing constraint is very costly in real, often large, feature models.

Keywords-Feature Model; Constraints; Support Vector Machine

# Introduction

Feature models provide an efficient way to organize and reuse requirements in a specific domain. Requirements are encapsulated in a set of *features*, and the features are often organized into a tree-like structure according to *refinement* relationships between them, and *constraints* are constructed to capture any additional dependencies between the features. The requirements are then reused by selecting a subset of features from the tree without violating the constraints.

In the construction of feature models, the first step is to identify features, and then organize them into a tree-like structure. The process needs a broad review of requirements documents of existing applications in a domain, which takes a lot of effort.

However, finding constraints among identified features is a more challenging task. There are mainly two reasons. First, the size of problem space of finding constraints is the square of identifying features. In other words, a certain feature may be constrained by any other features, and all the possibilities need to be checked to avoid any miss. Second, features are often concrete, which means that they can often be directly observed from an existing application or its documents [1]. By contrast, constraints are often abstract, which means that they often have to be learned from a systematic review of several similar applications.

Many automated or semi-automated approaches have been proposed to relieve human work in construction of feature models. However, these approaches focus on feature identification or refinement discovery. In this paper, we focus on automatically mining binary constraints. Binary constraints are the most basic kind of constraints. We focus on them for three reasons. First, they are adopted in every feature-oriented method therefore our approach has a wide applicability. Second, although there are more complicated kinds of constraints existed [15], binary constraints are the most common form of constraints in real feature models. Finally, they are simple so that they could be a suitable starting point for mining constraints in the constructions of feature models.

As binary constraints involve exactly two features, we can view the problem of mining binary constraints from a classification perspective. Consider a pair of features in a feature model, it falls exactly into one of the following classes: no constraints between the paired features, a *requires* constraint between them, and an *excludes* constraint between them. From such a perspective we transform the mining problem into a 3-class classification problemon feature pairs. The input of our approach is a feature model in construction, in which features and their descriptions are provided, and the feature tree may be constructed, and a few binary constraints have already been constructed. We first construct feature pairs from the input, and then train a classifier with previous known binary constraints. A support vector machine is incorporated to implement the classifier, and a genetic algorithm is utilized to optimize the classifier. The trained and optimized classifier is then check feature pairs with unknown class to find possible binary constraint candidates.

We conduct a series of experiments on two feature models constructed by third parties, to evaluate the effectiveness of our approach under different conditions that might occur in practical use. Results show that we can mine binary constraints at a high recall (near 100% in most cases), which is important because finding a missing constraint is very costly in real, often large, feature models.

The remainder of this paper is organized in the following way. Section II gives some preliminaries on feature models. Our approach is presented in Section III. Experiments and the discussions are described in Section IV. Section V discusses threads to validity, and related work is presented in Section VI. Finally, Section VII concludes the paper and describes our future work.

# Preliminaries: Feature Models

In this section, we give a brief introduction on feature models, especially the binary constraints which we want to mine from feature models.

Figure 1 shows a minimal meta-model of feature models. We claim that it is minimal because its elements are common in all mainstream feature modeling methods. A feature model consists of a set of *features* and *relationships* between them. The concept of feature can be understood in two aspects: intension and extension. In intension, a feature denotes a cohesive set of individual requirements. In extension, a feature describes a software characteristic that has sufficient user/customer value.

There are two kinds of relationships between features, namely *refinements* and *constraint*s. Refinements organize features into a feature tree. A feature may have several child features, and it may have at most one parent feature. Besides, a feature may be *mandatory* or *optional* relating to its parent feature (i.e. the *optionality* of a feature). A mandatory feature must be selected when its parent feature is selected, while an optional feature can be selected or removed.

Constraints capture additional dependencies between features. The most basic kind is the binary constraint, which is in turn categorized into two subtypes, namely *requires* and *excludes.* Given two features *X* and *Y*, *X requires Y* means that if *X* is selected then *Y* must be selected as well, while *X excludes Y* means that *X* and *Y* cannot be selected at the same time.



1. A minimal meta-model of feature models.

# A Classification-Based Approach to Mining Binary Constraints

In this section, we present our approach to mining binary constraints. We give the approach overview first, and then describe the details.

## Overview of the Approach

The key technique in our approach is classification, which means we first learn a classifier from known data (i.e. training), and then use it to classify unknown data (i.e. test). Figure 2 gives an overview of our approach. The input is training and test feature models. They can be the same feature model, and in such cases, binary constraints in a fraction of the feature model are already known, and the rest of the feature model is tested. The first step is making feature pairs. The original feature pairs contain textual attributes, i.e. names and descriptions. The second step quantifies feature pairs so that they are represented by numeric attributes and can be viewed as vectors. The classifier is then trained and optimized by the training vectors. The final step is to use the classifier to classify the test data.



1. An overview of our approach.

## Step 1: Make Feature Pairs

Feature pairs used in our approach are *unordered*, that is, given two features *X* and *Y*, we do not distinguish between the pairs *(X, Y)* and *(Y, X).* If a feature pair *(X, Y)* is classified as *requires*, it means that *X requires Y* or *Y requires X,* or both. The reason for making feature pairs unordered is that *excludes* and *non-constrained* pairs are unordered by nature, so we treat *requires* as unordered pairs as well. Furthermore, given an unordered *requires* pair *(X, Y)*, it is often trivial for people to further identify whether *X requires Y,* or *Y requires X*, or *X mutual-requires Y*. Therefore it does not harm the benefit brought by our automated approach.

Given a feature model of *n* features, there are *(n2-n)/2* unordered feature pairs in total. However, not all of them are used for mining binary constraints. We keep *cross-tree­­* pairs only, that is, a feature pair is kept if the two paired features do not have ancestor-descendant relationships. The reason is that a *requires* constraint between them is implied by the refinements (i.e. a child *requires* its parent, and a parent *requires* its mandatory children), and an *excludes* constraint between them is an error in a feature model. Therefore we eliminate feature pairs consisting of an ancestor and a descendant from the data set.

## Step 2: Quantify Feature Pairs

The classification technique in our approach assumes that feature pairs contain only numeric attributes. However, as shown in Figure 2, input feature pairs have two textual attributes, i.e. *name* and *description.* According to the textual attributes, we derive four numeric attributes as Table 1 shows.

1. Numeric Attributes for Feature Pair *(X, Y)*

|  |  |
| --- | --- |
| *Attribute* | *Definition* |
| Feature Similarity | Similarity between *X* and *Y*’s descriptions. |
| Object Overlap | Similarity between objects and their adjective modifiers of *X* and *Y*’s descriptions. |
| Target (1) | Similarity between *X*’s name and *Y*’s objects and their adjective modifiers. |
| Target (2) | Similarity between *Y*’s name and *X*’s objects and their adjective modifiers. |

The rationale behind these attributes is that constraints reflect dependencies and interactions between features, and the attributes indicate possible dependencies and interactions. Feature similarity indicates that such features should have common words in their descriptions. Object overlap reveals that their function areas are overlapped. Target shows that a feature is directly affected by another.

We utilize *Stanford Parser* to find objects and their adjective modifiers in features’ descriptions. An object is an entity which is being affected by actions described in a feature, and adjective modifiers distinguish between different characteristics and status of a certain kind of objects. Figure 3 gives an example taken from real products. Stanford Parser works well with incomplete sentences (e.g. missing subjects), which is common in features’ descriptions in the real world.



1. An example of extracting objects and their adjective modifiers by Stanford Parser.

The similarity between two textual documents is defined as the cosine of their corresponding term vectors. First, the words in the documents are stemmed and stop words are removed. Then the words are weighted by the widely used TFIDF (term frequency and inversed document frequency) score, which is defined as follows:

, and

, where the number of documents equals to the number of feature names plus descriptions.

The term vector *D* for a document *d* is the vector of TFIDF weights of its distinct words, that is:

Given two documents *d1* and *d2,* the similarity between them is defined as the cosine of their corresponding term vectors *D1* and *D2*:

.

## Step 3: Train the Classifier

The classification technique used in our approach is support vector machine (SVM), which has shown promising results in many practical applications. In this subsection, we first introduce the concept of *maximal margin hyperplanes* which forms the basis of SVM. We then briefly describe how an SVM can be trained to look for such hyperplanes to perform a *binary* classification (the input data is divided into two categories). Finally, we describe how to extend SVM to classify feature pairs which is a *three-class* classification problem and how to handle the imbalanced class problem in our approach.

### Maximal Margin Hyperplanes

Consider a training set containing training data items that have two numeric attributes *A1* and *A2*. We represent each data item by a vector *(a1, a2)* in a 2-dimensional space, where *ai* isthe data item’s value of attribute *Ai*, for *i =* 1, 2.We draw the vectors in a diagram (see Figure 4), and we assume that the data items belong to two classes, represented by circles and squares. We also assume that the training set is linearly separable, i.e. we can find a *hyperplane* (or a *line* in 2D space) in the space such that all the circles reside on one side of the hyperplane and all the squares reside on the other side. In fact, there are infinitely many such hyperplanes in the space. We draw two of them (*H1* and *H2*) in Figure 4. Each hyperplane *Hi* is associated with a pair of hyperplanes, denoted as *hi1* and *hi2*, respectively. *hi1* is obtained by moving a parallel hyperplane away from *Hi* until it touches the closest square(s), whereas *hi2* is obtained by moving the hyperplane until it touches the closest circle(s). The distance between *hi1* and *hi2*is known as the *margin* of *Hi.*



1. Maximal margin hyperplane.

The basic idea of SVM is to find hyperplanes with *maximal margin*. The rationale is that wider margins have been proved to have fewer errors in prediction (i.e. classify data that do not belong to the training set). Furthermore, wider margins are preferable even they make training errors sometimes. Figure 5 shows a training set that is similar to Figure 4 except it has two new items, *P* and *Q*. Although *H1* misclassifies them, while *H2* classifies them correctly, *H1* should still be preferred over *H2*. The margin of hyperplanes like *H1* is called a *soft margin*.



1. Maximal margin hyperplane with soft margin.

### The Support Vector Machine

*Support vectors* are the vectors residing on the (soft) margin of the maximal margin hyperplane, i.e. the vectors residing on *h11* and *h12* in the previous examples. A support vector machine tries to find support vectors in a given training set. Formally, consider a binary classification problem consisting of *N* training examples. Each example has *k* attributes and is denoted by a *k-*dimensional vector **xi**= (*xi1, xi2, …, xik*), and we use the variable to denote its class label (*i* = 1, 2, …, *N*). The two parallel hyperplanes defining a separating hyperplane’s margin is formulated by:

(1)

(2)

The parameters are a vector **v** and constants *b1, b2*, where *b1 > b2.* The margin is then given by, where is the length of **v**.

Given the two parallel hyperplanes, there are infinitely many separating hyperplanes between them, with the same amount of margin. The trick of SVM is that it always chooses the middle one, i.e. with the same distance from the two parallel hyperplanes. Suppose the separating hyperplane chosen by SVM is:

(3)

Then we can rewrite equation (1) and (2) as:

(4)

(5)

Finally, we scale the parameters in equations (3) to (5) and the margin by dividing them by. We get the simple form of the separating hyperplane and its marginal hyperplanes as:

(6)

(7)

(8)

Margin = (9)

According to the definition of margin, all the training data reside on or outside the marginal hyperplanes. Therefore for all training data, the following inequality holds:

(10)

(11)

Equations (10) and (11) can be expressed in a more compact form as follows:

(12)

In summary, the simplest form of SVM is defined by equation (9) and (12) as follows.

**Definition 1 (SVM: Linearly Separable Case).** Given a training set of *N* items {**x1, x2, …, xN**} that belong to two classes labeled as , training an SVM is equal to solving the following constrained optimization problem:

Training sets in practice are often non-linear and non-separable. For non-linear case (i.e. the boundary between two classes is not a hyperplane), a functionis used to transform original vectors (**xi**) into a much higher dimensional space (sometimes even an infinite-dimensional space) so that the boundary become a hyperplane in that space. For non-separable case, a few training errors are inevitable, so the penalty of making the errors must be taken into account. In addition, the margin is slacked by a positive amount i for each training example **xi** (i.e. the soft margin mentioned before). Formally, the most general form of SVM is defined as follows.

**Definition 2 (SVM: General Case).** Given a training set of *N* items {**x1, x2, …, xN**} that belong to two classes labeled as , training an SVM is equal to solving the following constrained optimization problem:

An off-the-shelf classifier such as LIBSVM can compute the **w**,, *b* and to solve the optimization problem. However, the penalties (or weights) of the classes, and, must be set manually before training and using the classifier, and their values significantly affect the effectiveness of the classifier. Our solution is to use an optimization algorithm to find optimized weights of the classes. (We will discuss this later in Section III.E.)

### Extend SVM to Three-Class Classification

The SVM introduced before only supports classification of two classes. To extend it on three classes in our approach, we incorporate the *one-against-one* strategy as follows.

We denote *non-constrained*, *requires,* and *excludes* by *y1*, *y2, y3*, respectively. The classifier runs three times, where in each time it distinguishes between two classes (*yi, yj*), for . Training examples that do not belong to either *yi* or *yj* are ignored when training the classifier on classes (*yi, yj*). When making predictions, a feature pair’s class is determined using a voting strategy: when classifying against (*yi, yj*), the corresponding class receives a vote after the feature pair is classified. Then we predict the feature pair is in the class with the most votes. In case that two or more classes have identical votes, we simply select the class with smaller index. This simple tie-breaking strategy gives good results according to [1].

## Step 4: Optimize the Classifier

The classifier needs to be optimized on a given training set to find optimized parameters. Table 2 lists the parameters to be optimized. Each parameter has a default value that is used as the initial value at the beginning of optimization. The optimized parameters are selected within their ranges.

For the weights of classes, we use the *non-constrained* class as a baseline and always set its weight to 1. Then we compute two ratios:

and we use them as default values for the *requires* and *excludes* class, respectively. The rationale is that a correct classification of a rare class should have greater value than a correct classification of a majority class. Therefore the weight of classes has a close connection to the number of instances in these classes.

A parameter is used in the function mentioned in Definition 2. The authors of LIBSVM suggested that the default value of should be set to (1 / number of attributes), which is 1 / 4 here. They also suggested that needs to be optimized in practice.

We define the ranges based on a factor of 10, that is, start from 1 / 10 of a parameter’s default value, and end with 10 times larger than its default value. The weight is an exception. It is always larger than 1 (if *Ri >* 1) or smaller than 1 (if *Ri* < 1).

1. Classifier Parameters to be Optimized

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | Meaning | Default Value | Range | Step |
| Creq | The weight of the *requires* class. | R1 | (1, 10R1) or  (1 / 10R1, 1) | 0.5 |
| Cexc | The weight of the *excludes* class. | R2 | (1, 10R2) or  (1 / 10R2, 1) | 0.5 |
|  | A parameter in the function (see Definition 2) used by LIBSVM. |  |  | 0.01 |

In order to find optimized parameters, we need to measure the effectiveness of the classifier under a given set of parameters. We run a *k-fold cross-validation* on the training set and compute the *error rate* of classification. The *k-*fold cross-validation is performed as follows. First, we divide the training set into *k* equally sized subsets. We then do the classification *k* times, and in each time, a distinct subset is chosen for testing and other subsets are used for training. Therefore in the end each instance is tested (classified) exactly once. Finally, we compute the error rate of classification as:

We incorporate a genetic algorithm for optimization, and it is sketched in Algorithm 1. A *solution* is a tuple of the three parameters, i.e. (*Creq, Cexc,*). The main operations in a genetic algorithm are *mutation* and *crossover*. A *mutation* takes a solution and makes a little change on a random number (1 to 3) of its parameters. The little change here means randomly increases or decreases the parameter by a predefined step (see Table II). A *crossover* takes two solutions and combines a random number of parameters from one solution with the rest parameters from another solution. Both operations produce a new solution.

The first step of the algorithm is to generate the initial set of solutions. We define a *seed solution* by the default values of the parameters. We then perform a mutation on the seed solution to produce a new solution in the initial set. The procedure is repeated until the initial set is full.

The algorithm then repeats an evolution step for a given number of times. In each evolution step, we train a classifier for each solution, with the parameters in it. Then we do *k-*fold cross-validations for each classifier and a certain amount of solutions that give the lowest error rate (known as the *elites*) are kept, and the rest solutions are eliminated from the solution set. New solutions are then produced by randomly performing mutation or crossover on randomly selected elites. These new solutions are added to the solution set until the set is full again.

Finally, the overall optimized solution is the best solution in the last solution set. We use the parameters in this solution to train the classifier, and thus we get an optimized classifier for a given training set.

|  |
| --- |
| Algorithm 1: Find optimized parameters for the classifier |
| optimize (seed: Solution, m: int, e: double, p: double): Solution  solution\_set {seed}  repeat  Add a mutation of seed to solution\_set  until solution\_set is full  repeat m times  Train a classifier and do cross-validation for each solution  elites {The best solutions in solution\_set}  solution\_set elites  repeat  r a random number between 0 and 1  if (r < p)  x a randomly selected solution from elites  Add a mutation of x to solution\_set  else  (x1, x2) two randomly selection solutions from elites  Add a crossover of (x1, x2) to solution\_set  end if  until solution\_set is full  end repeat  return the best solution in solution\_set  end |

# Experimental Evaluation

We conduct a series of experiments to evaluate the classifier. Input data are constructed by third parties and they are completely independent to our classifier. We evaluate the performance of the classifier in different scenarios, including:

* Training data and test data come from the same or different domains. These experiments check that whether a classifier trained by a feature model of some domain can be applied to a feature model of some totally unrelated domain.
* Training is supervised or semi-supervised. These experiments evaluate the effect of different training strategies.
* Feedback is enabled or disabled during testing. In practice, human analysts may use our approach to get constraint candidates and then give feedback on correctness of the candidates. The procedure may repeat several times until all constraints are found. We also design experiments to simulate the above scenario and show the effect of human feedback on the classifier.

## Data Preparation

We use two feature models from the SPLOT repository[[1]](#footnote-1) for the experiments. Table III shows the basic information about the feature models. To view the feature models, follow the “Edit” link in the “Feature Model Repository” page, and select a feature model according to its name. The authorship information provided by the SPLOT repository shows that the two feature models are created by experts in this field (Don Batory and the pure-systems Corp., respectively), so they can be trusted inputs for our experiments.

1. The Feature Models for Experiments

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Features | Feature Pairs | Constraint Pairs |
| Weather Station | 22 | 196 | 6 *requires*  5 *excludes* |
| Graph Product Line | 15 | 91 | 8 *requires*  5 *excludes* |

A major problem is that feature models available online or in publications do not contain feature descriptions. Our solution is to search the features in Wikipedia and copy the first paragraph of their definitions as their descriptions. The rationale is that most features in the two feature models are domain terminologies which can be clearly defined, and by convention, the first paragraph in a Wikipedia page is a short summary of the term which is close to a real description that would appear in a feature model. Figure 6 illustrates some features and their descriptions completed in this way. A few features are not defined in Wikipedia, and in such cases, we leave their descriptions as blank.



1. Example features in the experiments.

## Measure the Performance of the Classifier

The error rateintroduced in Section III.E is a standard measurement of the overall performance of a classifier. In addition, since we focus on finding constraints, we compute the *confusion matrix* for *requires* and *excludes* class, and calculate their *precision, recall* and *F2-measure.* A confusion matrix shows the number of instances predicted correctly or incorrectly (see Table IV). The counts tabulated in a confusion matrix are known as True Positive (TP), False Negative (FN), False Positive (FP), and True Negative (TN). The precision, recall and F2-measure are then computed as:

-measure receives a high score if both precision and recall are reasonably good. The parameter is a positive integer which means recall is times more important than precision, and it is often set to 1 or 2 (i.e. F1- or F2-measure). We use F2-measure because for human analysts, finding a missing constraint is much harder than identifying a false constraint; in other words, the classifier should strive for high recall on binary constraints.

1. A Confusion Matrix

|  |  |  |
| --- | --- | --- |
|  | Predicted Positive | Predicted Negative |
| *Actual Positive* | True Positive (TP) | False Negative (FN) |
| *Actual Negative* | False Positive (FP) | True Negative (TN) |

## Compare Different Data Set Selecting Strategies

The first group of experiments is designed to compare the effect of different data set (i.e. training set and test set) selecting strategy. Given two feature models of unrelated domains, one of them is known as the *training feature model*, and the other is called the *test* *feature model,* there are three possible strategies:

* Cross-Domain Strategy. The training set is the training feature model, and the test set is the test feature model. If this strategy wins, it means that in practice, analysts may directly apply a trained classifier to their work, and the accumulated knowledge from different domain may further benefit the classifier.
* Inner-Domain Strategy. The training set a fraction of the test feature model, and the test set is the rest of the feature model (the training feature model is not used here). If this strategy wins, it means that in practice, analysts have to manually classify a fraction of feature pairs in advance, and then incorporate a fresh classifier to work.
* Hybrid Strategy. The training set is the training feature model plus a fraction of the test feature model, and the test set is the rest of the test feature model.

Weather Station and Graph Product Line are used as the training and the test feature model, respectively. Their roles are then exchanged. Therefore there are classification settings. For each setting, a train-optimize-test procedure of the classifier is repeated 20 times, and the average performance of the classifier is presented here. Common experiment parameters for the classification settings are shown in Table V.

1. Common Experiment Parameters

|  |  |
| --- | --- |
| Parameter | Value |
| The fold of cross-validation | 10 |
| The size of solution set | 100 |
| The proportion of elite solutions | 20% |
| The probability of mutation (crossover) | 30% (70%) |
| The number of iterations | 200 |
| The fraction used for training in the inner-domain and hybrid strategy | 1 / 5 |

A 10-fold cross-validation is performed for estimating the effectiveness of classifier during the optimization. The number 10 is recommended in most data mining tasks. Table V also shows the parameters of the genetic algorithm (row 2 to row 5). The average error rate of 20 runs of optimization, given by 10-fold cross-validation on training set, is shown in Table VI.

Table VI clearly reflects the need and benefit of optimization. The average error rate of default parameters (i.e. before optimization) is somewhat random, ranging from 2.89% to about 73%. The optimization effectively decreases the error rate, especially for cross-domain and inner-domain strategies. The worst optimized error rate is only 12.95%.

1. Average Accuracy on Training Set (10-Fold CV)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Feature Model | Training = WS, Test = GPL | | | Training = GPL, Test = WS | | |
| Strategy | Cross | Inner | Hybrid | Cross | Inner | Hybrid |
| Avg. Error % (Default) | 18.2 | 72.89 | 2.89 | 16.17 | 64.68 | 12.97 |
| Avg. Error % (Optimized) | 0.82 | 12.95 | 2.40 | 8.83 | 4.70 | 11.01 |

For the inner-domain and hybrid strategies, 1 / 5 of the test feature model is used for training. We randomly divide the feature pairs in test feature model into 5 equally sized subsets. We then run the classifier 5 times, and each time we use a distinct subset for training and optimizing and others for testing. We treat the above process as a whole 5-fold train-optimize-test process. For the cross-domain strategy, the training set and test set do not change, so it is a 1-fold train-optimize-test process. The above processes are again repeated 20 times, and the average performance of finding binary constraints is shown in Table VII, under the columns marked as “L”.

The first observation is that the cross-domain strategy fails to find any *excludes* constraints. By looking at the features involved in *excludes* in both feature models, we find that descriptions of these features follow totally different patterns in different feature models. In the weather station feature model, the rationale of *excludes* is often beyond the description (or, it is actually a definition) of the features. For example, an *excludes* states that sending weather report via text message (the feature *Text Message*) does not support XML format (the feature *XML*). Regarding to the four numeric attributes of feature pairs, the two features seem to be totally unrelated. Their similarity is very low, and they do not share any objects or target on others’ names. By contrast, the rationale of *excludes* can be deduced from descriptions in the graph product line feature model. For example, two algorithms (each one is a feature) have the same effect on the same type of graphs, so that users must choose between them and therefore they are considered to be *excluded*. In this case, the two features are similar to each other, and they share many objects which indicate there is interference between their functional areas. The recall of *excludes* is improved by inner-domain and hybrid strategy, due to the training data from the part of test feature model.

The mining of *requires* receives a high recall for each strategy. The reason is that most *requires-constrained* feature pairs follow a similar pattern: one feature is targeted by another to some extent. For example, a certain kind of weather report (e.g. *Storm*) requires specific sensors (e.g. *Temperature* and *Pressure*); an algorithm (e.g. *Prim*) only applies to a specific kind of graph (e.g. *Connected* and *Weighted*). In summary, the occurrences of most *requires* can be deduced from the feature descriptions, which benefits the mining of *requires* in our experiments.

The precision of mining binary constraints is not stable in our experiments, and is highly dependent on the test feature models. However, there are still some interesting patterns. First, the cross-domain strategy gets the lowest score again, so it may not be a preferable strategy in practice. Second, a strategy that gives a higher precision in *requires* will give a lower precision in *excludes.* This phenomenon again conforms to the different patterns of the two kinds of constraints, as described before. Therefore a classifier that is good at recognizing one kind of pattern is somewhat bad at recognizing another. Finally, a larger test feature model (i.e. the weather station) tends to give higher precisions. The reason is that there are more training data from the test feature model for inner-domain and hybrid strategies, therefore the classifier can be better trained.

In summary, the cross-domain strategy may not be preferable in practice, and there is no significant difference between the inner-domain and hybrid strategies. The classifier gives a reasonably high recall on both kinds of constraints, although the precision depends on the being tested feature model. However in practice, finding a missing constraint in a large feature model is often much harder than reviewing a constraint candidate, therefore the high recall shows that our approach might be promising in practical use.

1. Comparison of Data Set Selecting Strategies and Training Strategies

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Strategy | Requires | | | | | | Excludes | | | | | |
| Precision % | | Recall % | | F2-Measure | | Precision % | | Recall % | | F2-Measure | |
| L | LU | L | LU | L | LU | L | LU | L | LU | L | LU |
|  | Training FM = Weather Station, Test FM = Graph Product Line | | | | | | | | | | | |
| Cross-Domain | 7.5 | 17.53 | 100 | 94.44 | 0.288 | 0.503 | N/A | N/A | 0 | 0 | N/A | N/A |
| Inner-Domain | 14.95 | 12.14 | 84.67 | 93 | 0.438 | 0.399 | 100 | 100 | 100 | 100 | 1 | 1 |
| Hybrid | 23.41 | 20.42 | 84 | 84.67 | 0.553 | 0.52 | 14.17 | 20.46 | 100 | 100 | 0.452 | 0.563 |
|  | Training FM = Graph Product Line, Test FM = Weather Station | | | | | | | | | | | |
| Cross-Domain | 66.67 | 50 | 100 | 100 | 0.909 | 0.833 | N/A | N/A | 0 | 0 | N/A | N/A |
| Inner-Domain | 92.67 | 86 | 100 | 94.67 | 0.984 | 0.928 | 22.14 | 2.68 | 80 | 100 | 0.525 | 0.121 |
| Hybrid | 73.06 | 74.07 | 93.33 | 100 | 0.884 | 0.935 | 35.14 | 22.17 | 66.67 | 80 | 0.565 | 0.526 |

## Compare Different Training Strategies

In the previous sub-section, the classifiers are trained with labeled data (i.e. data in the training set) only. It is also known as the *supervised* training strategy. Another type of training strategy, known as the *semi-supervised* training, is also widely adopted in machine learning. The idea of semi-supervised training is to train a classifier with both labeled and unlabeled (i.e. unclassified data in the test set) data. The rationale is that although the class of unlabeled data is unknown, unlabeled data still contain a lot of information (e.g. parameter distribution) that could benefit the classifier. A well-known procedure of the labeled/unlabeled (LU) training is sketched as Algorithm 2. First, the classifier is trained by a training set, and used to do classification. The result of classification and the training set are then used together to re-train the classifier, and the classifier do the classification again. The procedure repeats until the result of classification is stable enough.

|  |
| --- |
| Algorithm 2: Labeled/Unlabeled (LU) Training |
| LU\_Training (L: Training\_Set, U: Test\_Set)  // A supervised, initial training  Train the classifier by L  Do classification on U  The result of classification  // Iterated semi-supervised training  repeat  Re-train the classifier by  Do classification on U with the re-trained classifier  The result of classification  until the result of classification is stable enough  end |

We change the training strategy to LU training and keep other things unchanged, and then redo previous experiments. The results are also shown in Table VII, under the columns marked as “LU”.

We can observe that the LU training strategy slightly improves the recall of *excludes*, however the precision is still not stable. It even decreases the precision to a great extent in many cases.

In summary, the LU training strategy does not improve the performance of the classifier in our experiments, and considering the additional training time it takes, we prefer the traditional labeled training strategy.

## The Effect of Limited Feedback

In practice, human analysts may use the classifier to get some results, and then provide feedback on the correctness of the results, and in turn, the classifier is re-trained according to the feedback. Figure 7 explains such scenarios. We call it *limited* feedback since the number of results presented to human analysts must be small, to keep their workload as low as possible.



1. The process of limited feedback.

The strategy of choosing results for feedback in our experiment is known as *constraint first and similarity first* strategy. All test feature pairs are sorted by descend order of the similarity between the paired features. Then a number of mined constraints with the highest similarities are selected. If there is no constraint mined or no enough constraints, the non-constrained pairs with the highest similarities are selected to fill the blank.

We experiment limited feedback with supervised training strategy. After each train-optimize-test process, 3 pairs (i.e. about 2% to 5% of the whole test set) are selected for feedback. The process is looped for 10 times, therefore in the end there are 20% to 50% test data are checked. We repeat the whole 10-loop feedback for 5 times, and the changes of average precision and recall over accumulated number of feedback are shown in Figure 8.

1. The effect of limited feedback.

One observation is that the recall will increase eventually, after a certain number of feedback loops. In other words, the recall at the end is not less than the recall at the beginning in every situation. It is best illustrated by the recall of *excludes* of cross-domain strategy in Figure 8(b), where the recall increases from 0 to 1. However, the recall fluctuates within a range in some cases, and sometimes the range is wide, e.g. the recall of hybrid strategy in Figure 8(a). The reason is that since we provide feedback on constraint candidates first, the proportion of real constraints left in the test set continuously decreases after each feedback loop, and in turn, the difficulty of mining the rest constraints rises. Therefore the recall may decrease during a period of time. At some point, when the classifier has learned enough number of feedback (i.e. the training set become large enough), it is again able to find the rest constraints correctly, so that the recall eventually increases.

Another observation is that the precision is still unstable during the feedback loop. It changes a lot (about 40% to 50%) when the test feature model is graph product line, but only fluctuates a little (less than 20%) in another feature model. It is also not guaranteed to increase eventually.

A direct, and maybe most useful benefit brought by feedback is the improvement in mining *excludes* constraints. As shown in Table VII, the cross-domain strategy fails to find any *excludes* when there is no feedback. With the help of feedback, it eventually finds all *excludes* with a precision around 30% in the graph product line feature model (Figure 8(b)), and finds some *excludes* with a precision around 10% in another feature model (Figure 8(d)).

## Summary

Our experiments show that our approach successfully finds binary constraints at a high recall (near 100% in most cases). The precision is unstable and dependent on the test feature models. In most cases the *requires* constraints are better mined than the *excludes* constraints; a possible reason is that the rationale behind *excludes* is often beyond feature descriptions.

Optimization significantly improves the score of cross-validation of classifiers, where the cross-validation is a standard procedure for estimating the effectiveness of classifiers in data mining research. Optimization helps find most suitable parameters for classifiers, and makes classifiers stable.

The inner-domain and hybrid data set selecting strategies perform better than the cross-domain strategy. It conforms to the intuition that the differences between domains are significant. However, when trained by a small number of known binary constraints in a feature model, our classifier can be effectively find binary constraints left in the feature model, no matter which domain it belongs to.

The more complicated and time-consuming training strategy, i.e. the semi-supervised training, does not show significant improvement in the experiments. The traditional supervised training is enough for us.

Continuous feedback from human analysts benefits the mining process, especially for mining *excludes* constraints. Therefore in practice, our classifier should be used in an interactive way, that is, human analysts check only a few constraint candidates after each turn of mining, and then the classifier repeats the train-optimize-test process again.

# Threats to Validity

## External Validity

A major threat to the external validity of our experiments is that we did not carry out the experiments on real feature models. By “real” we mean that the feature model is created from real applications by experienced domain analysts, such as the Linux [11] and eCos [12] feature models. We did not use these real feature models because they do not contain feature descriptions. However, the two feature models used in our experiments have similar characteristics to real feature models, regarding to our approach:

First, the abstract of a definition in Wikipedia and the description of a feature are similar in the way of wording and expression: they both strive for clear and concise, and they both tend to use domain terminologies. It means that there are few unnecessary words disturbing the computation of the four numeric attributes for feature pairs.

Second, some features do not need descriptions in real feature models. For example, some basic terminologies, such as *File* or *Play Music,* can be understood without further explanation. The feature models we used also contain some non-described features, and our classifier still works in such cases.

Another threat is the size of data set in our experiment is smaller than most data sets used in the data mining field, so the performance of the classifier may be affected. Our solution is to optimize the classifier before it is used for classification. Researchers find that such an optimization even works for small data set (300 to 400 data) [8], and our data set has about 300 data so it is still suitable for a classifier to work. In fact, the error rate before and after optimization reported in [8] follows a similar pattern to ours.

## Internal Validity

We strive for high recall of the binary constraints, because finding a missing constraint in a large feature model is much harder than reviewing constraint candidates. However, the relatively low precision may still impose a lot of workload on human analysts, because they have to review a large number of candidates to get all constraints. To improve the precision while preserving high recall is the target of our future work.

The size of our training set and test set is unfair to semi-supervised training strategy, because the best scenario to applying the strategy is that the training set is far smaller than the test set, which is not satisfied in our experiment. In a large feature model (e.g. thousands of features), if only a couple of constraints are constructed before and we want to find the rest ones (i.e. the training set is very small compared to the test set), then the semi-supervised training strategy may perform better.

# Related Work

Feature models and binary constraints are first proposed in [9]. Since then, several feature modeling methods as well as various forms of feature models have been proposed, e.g. [7] [15]. Feature names, descriptions, and the two types of binary constraints are common elements in all existing form of feature models.

Chen et al. [2] proposed a semi-automated approach to constructing feature models based on requirements clustering. The idea is first identifying features from strongly connected requirements, and then finding the refinement relationships to build a feature tree. In contrast, our work takes a feature tree as input and finds cross-tree binary constraints in a fully automated way. Therefore it is possible that the two methods are used together to build feature models from requirements documents.

She et al. [12] proposed an approach to recover a feature model from a given set of features and constraints. Their main purpose is to recover the refinements (also known as implicit constraints) which are hidden in the constraint set. They also compute similarity between feature descriptions to recommend possible parent features for a given feature. The full constraint set is given as an input, which is the major difference from our work. Similar to [12], Czarnecki [3] proposed a method to build a feature model from a constraint set (given as propositional logic formulas), however they do not utilize feature names and descriptions and it leads to an unsatisfying feature tree.

Since a feature represents a cohesive set of requirements, mining constraints between features can also be regarded as mining constraints between requirements. Dag et al. [4] proposed a method to find similar or duplicated requirements. The main technique is to use information retrieval method to compute similarity between requirements. Duan et al. [5] utilize a clustering algorithm to find prioritization relations between requirements. Weston et al. [14] use a natural language parser to annotate aspect-oriented requirements which enables them to partially discover conflicts between the requirements. The information retrieval, natural language processing and machine learning techniques utilized in these methods inspired our work. In our work, we use natural language processing techniques to find objects in feature descriptions, information retrieval techniques to compute similarity, and machine learning (classification) techniques to do actual mining work.

The support vector machine classifier, LIBSVM, used in our experiments is implemented by [1]. The need of optimizing a classifier is explained in [8], and the error rate before and after optimization shown in [8] follows a similar pattern to ours. The basic idea of genetic algorithm is learned from [6], and we implemented the algorithm by ourselves.

# Conclusions

In this paper, we propose an automated approach to mining binary constraints from a partially constructed feature models. The partially constructed feature model contains a set of features and their descriptions, and a few binary constraints. It also may contain a feature tree. We first construct feature pairs from the input, and then train a classifier with previous known binary constraints. A support vector machine is incorporated to implement the classifier, and a genetic algorithm is utilized to optimize the classifier. The trained and optimized classifier is then check feature pairs with unknown class to find possible binary constraint candidates.

We conduct a series of experiments on two feature models in the SPLOT repository. The descriptions of the features are extracted from Wikipedia. We evaluate different data set selecting strategies and training strategies. We also examine the effect of accumulated feedback on predicted constraint candidates. Results show that we can mine binary constraints at a high recall (near 100% in most cases), which is important because finding a missing constraint is very costly in real, often large, feature models.

##### References

1. C. C. Chang and C. J. Lin, LIBSVM : a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1--27:27, 2011.
2. K. Chen, W. Zhang, H. Zhao, H. Mei. "An approach to constructing feature models based on requirements clustering," Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference , pp. 31- 40, 29 Aug.-2 Sept. 2005.
3. K. Czarnecki, A. Wasowski. Feature Diagrams and Logics: There and Back Again. Software Product Line Conference, 2007. SPLC 2007. 11th International , vol., no., pp.23-34, 10-14 Sept. 2007.
4. J. N. Dag, B. Regnell, P. Carlshamre, M. Andersson and J. Karlsson. A Feasibility Study of Automated Natural Language Requirements Analysis in Market-Driven Development. Requirements Engineering, 2002 (7): 20-33.
5. C. Duan, P. Laurent, J. Cleland-Huang, C. Kwiatkowski. Towards automated requirements prioritization and triage. Requirements Engineering. 2009, 14: 73-89.
6. D.E Goldberg. Genetic algorithms in search, optimization and machine learning. Addison-Wesley, 1989.
7. M. L. Griss, J.Favaro, M. d’Alessandro, “Integrating Feature Modeling with the RSEB,” Fifth Intl. Conf. on Software Reuse (ICSR 98), IEEE Computer Society, Jun. 1998, pp. 76–85.
8. C. W. Hsu, C. C. Chang, C. J. Lin. A practical guide to support vector classification. http://www.csie.ntu.edu.tw/~cjlin/papers/guide/
9. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
10. K. Nigam, A. K. Mccallum, S. Thrun and T. Mitchell. Text Classification from Labeled and Unlabeled Documents using EM. Machine Learning, vol. 39, no. 2-3, 103-134, 2000.
11. S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Variability Model of the Linux Kernel", Fourth International Workshop on Variability Modeling of Software-intensive Systems (VaMoS 2010), Linz, Austria, 2010.
12. S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 461-470.
13. J.A.K. Suykens and J. Vandewalle. Least Squares Support Vector Machine Classifiers. Neural Processing Letters, vol. 9, 293-300, 1999.
14. N. Weston, R. Chitchyan, A. Rashid. Formal semantic conflict detection in aspect-oriented requirements. Requirements Engineering. 2009, 14: 247-268.
15. W. Zhang, H. Mei, H. Zhao, “A feature-oriented approach to modeling requirements dependencies,” in Proc. of the 13th IEEE Intl. Conf. on Requirments Engineering (RE 05), 2005, pp. 273–282.

1. http://www.splot-research.org [↑](#footnote-ref-1)