

# An Efficient Sequential Covering Algorithm for Explaining Subsets of Data

Matthew Michelson<sup>1</sup> and Sofus Macskassy<sup>1</sup>

<sup>1</sup>Fetch Technologies, 841 Apollo St., Ste. 400, El Segundo, CA, USA

**Abstract**—Given a subset of data that differs from the rest, a user often wants an explanation as to why this is the case. For instance, in a database of flights, a user may want to understand why certain flights were very late. This paper presents ESCAPE, a sequential covering algorithm designed to generate explanations of subsets that take the form of disjunctive normal rules describing the characteristics (*{attribute, value}* pairs) that differentiates the subsets from the rest of the data. Our experiments demonstrate that ESCAPE discovers explanations that are both compact, in that just a few rules cover the subset, and specific, in that the rules cover the subset but not the rest of the data. Our experiments compare ESCAPE to RIPPER, a popular, traditional rule learning algorithm and show that ESCAPE’s rules yield better covering explanations. Further, ESCAPE was designed to be efficient, and we formally demonstrate that ESCAPE runs in loglinear time.

**Keywords:** Sequential Covering Algorithm; Loglinear

## 1. Introduction

Users often want to understand what differentiates some subset of data from the rest of the records in a data set. More concretely, consider a database of airline flights. A user may want to select all of the flights that were extremely late to try and determine why they were so behind schedule. Intuitively, the user searches for the characteristics that are common to various slices of the subset. For example, consider the flight database in Table 1. The late flights (with lateness  $\geq 1$  hour) are shown in **bold**, and seem to originate from LAX, or are flying into SFO usually resulting in at least one hour’s lateness.

Table 1: Example database of flights

Origin	Destination	Lateness (hrs)
<b>LAX</b>	<b>JFK</b>	<b>2</b>
LBO	JFK	0
<b>SJA</b>	<b>SFO</b>	<b>1</b>
<b>BWI</b>	<b>SFO</b>	<b>1.5</b>
<b>LAX</b>	<b>JFK</b>	<b>1</b>
NWK	JFK	0.1
<b>LAX</b>	<b>JFK</b>	<b>2</b>

We consider the description that the flights are “originating in LAX or flying into SFO” to be an *explanation*

This effort was in part sponsored by the Air Force Office of Scientific Research (AFOSR) under grant number FA9550-09-C-0064. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFOSR or the U.S. Government.

of the subset of late flights because it compactly describes the common characteristics of this subset. That is, the user can tell a story about the records in this subset, without considering every single possible attribute and value pair. Instead, just the most common characteristics are enough to describe the subset. Note that flying into JFK is not a good indicator of being late, since this destination occurs for both the late flights and those on time and is therefore not a common characteristic to just the subset. Therefore, this is not part of the explanation.

Such explanations can be naturally expressed through propositional logic rules in disjunctive normal form (DNF), where conjunctions of the common characteristics are unioned together. In DNF form, some records are covered by some conjunction while other records are covered by others. In our example from Table 1, we might formalize our explanation as the DNF rule:  $[\text{Origin}=\text{LAX}] \cup [\text{Destination}=\text{SFO}]$ . This DNF rule has two conjunctions (singletons in this case) unioned together, since some flights are covered by the  $[\text{Origin}=\text{LAX}]$  conjunction while others are covered by  $[\text{Destination}=\text{SFO}]$ .

In particular, propositional logic is well suited to generating subset explanations because it encodes each term in the conjunction as an *{attribute, value}* pair. This is in contrast to first-order logic where each term is an attribute with a variable for its value. By using propositional logic, we force our explanations to be specific to the records they cover in the subset because they must include the attribute *and* the value. This is important because records in the subset share the attributes with those that are not in the subset (e.g. all flight records have an origin), and it is the difference in those attributes’ values that causes records to be part of the subset or not.

Since generating DNF rules by hand is tedious, prone to errors, and in certain cases impossible (if the data set is large enough), this paper presents ESCAPE<sup>1</sup>, an algorithm for mining descriptive explanations using an approach known as sequential covering. Given a subset of records, and it’s complement (or random sampling of its complement) ESCAPE generates a rule in DNF form to explain the records in the subset, just as was done using the data from Table 1.

This paper presents two main contributions. First, ESCAPE can be formally shown to run in loglinear time. Previous rule covering algorithms that run in loglinear time were either only shown to do so experimentally [6] or shown to run in  $O(n \log^2 n)$  time [7].

Further, previous rule learning approaches were difficult to

<sup>1</sup>Efficient Sequential Covering for Propositional-logic Explanations

analyze formally [4], [5], [10]. ESCAPE, however, allows for simple theoretical running time analysis. Since ESCAPE is simple to analyze, it is well understood in terms of efficiency, and therefore future improvements to the approach can be well understood based upon this foundation.

Second, ESCAPE does not sacrifice the quality of the explanations it generates for the sake of efficiency. Our experiments show indeed it generates “expressive” explanations for subsets of data. These explanations are expressive in that the rules describe the records in the subset and not those in its complement. At the same time, the rules are compact in that ESCAPE generates many fewer rules than there are records in the subset (otherwise, it is trivial to return a conjunction for each record in the subset, which generates very specific, but useless rules). Lastly, ESCAPE explains data sets without making an explicit assumption that the data is iid, because the goal is to explain the subset, rather than learn a classifier. Rather, we just require the discovered rules to cover (thus explain) the records. This is an advantage because users are selecting the subsets explicitly, and therefore i.i.d. data cannot be assumed.

The rest of this paper is organized as follows. Section 2 describes the ESCAPE algorithm in detail and formally analyzes its efficiency. Section 3 then empirically demonstrates the effectiveness of ESCAPE’s rules using various subsets from two domains. Section 4 shows the related research and Section 5 presents our conclusions and future research directions.

## 2. The ESCAPE Algorithm

Here we describe ESCAPE in detail, and present a formal running time analysis of the efficiency.

Historically, sequential covering algorithms (SCA) take as input a set of records that act as positive and negative examples, and return a propositional-logic rule, in disjunctive normal form (DNF). Each conjunction of the rule covers some selection of records from the positive training examples, while minimizing the number of negative training examples covered. This DNF rule is then used to classify future records.

We map the classic SCA approach to our problem of explaining subsets, by assuming that the positive examples are the records that form the subset, and the negative examples are the complement of this subset (though a random sampling of the subset would suffice if access to the whole data set is prohibitive). ESCAPE then takes these “positive” and “negative” examples and learns a covering rule that covers as many positive examples as it can, while minimizing the negative examples it covers. In this way, the generated rule acts as a specific explanation of subset focusing on what differentiates it from the complement set.

To make our ESCAPE algorithm clear, we will walk through the running example, starting with the subset chosen in Figure 1. Figure 1 reiterates the example from the Introduction, where we start with a database of flights, each with an origin, destination, and lateness measure, and we want ESCAPE to explain the subset of very late flights. Note that

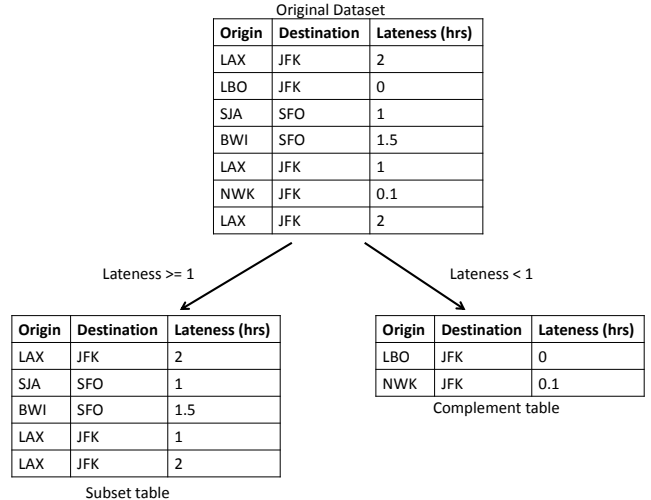


Fig. 1: An example subset to describe

we call the set of records in the subset the “subset table” and the other records the “complement table.”

The intuition behind ESCAPE is to generate conjunctions using only those {attribute, value} pairs that occur almost exclusively in the subset table. This way the explanation is specific to the subset. The conjunctions themselves are generated by considering groups of records in the subset that share these exclusive {attribute, value} pairs and choosing the pairs in common for these records. Therefore, ESCAPE’s goal is to efficiently discover which records share which exclusive {attribute, value} pairs, such that it builds the fewest and most effective conjunctions it can.

As the first step of the algorithm, ESCAPE scans the subset table and records the frequency of each {attribute, value} pair, and then it scans the complement table and records the frequencies of these pairs from that table as well. Then, for each {attribute, value} pair in the subset table, it calculates a performance value using these stored frequencies. The performance metric is chosen to model the notion of “exclusivity” of an {attribute, value} pair in the subset, and is defined as:

$$Performance(c) = P_s(c) - (w * P_c(c))$$

Where  $c$  represents the {attribute, value} pair,  $P_s$  is the probability of the pair occurring in the subset table,  $P_c$  is the probability of the pair occurring in the complement table, and  $w$  is a penalty term used to weight the comparison between occurrences in each table. Since ESCAPE previously calculated and stored the frequencies from both the subset and the complement table, it efficiently computes  $P_s$  and  $P_c$ .

ESCAPE then uses this Performance value to decide which {attribute, value} pairs to consider when mining the conjunctions. Specifically, if  $Performance(c) < T_c$ , ESCAPE prunes the {attribute, value} pair from the set of all pairs which it may consider to use in conjunctions during rule generation. The behaviors of the parameters  $w$  and  $T_c$  are well understood. However, in the interest of clarity, we

Table 2: Example database of flights

Attribute	Value	Performance
Origin	LAX	0.6
Origin	SJA	0.2
Origin	BWI	0.2
Destination	JFK	-6.9
Destination	SFO	0.4
Lateness	2	0.4
Lateness	1	0.4
Lateness	1.5	0.2

defer explaining their behavior until after describing the whole ESCAPE algorithm.<sup>2</sup>

Table 2 shows each {attribute, value} pair from the subset table of Figure 1. Note that the pair {Destination, JFK} is underlined because ESCAPE prunes it from the set since it is not exclusive enough (i.e. it has a large negative Performance value using a value of 7.5 for  $w$ ).

After pruning away the non-exclusive {attribute, value} pairs, ESCAPE next builds a “binary occurrence” table. A binary occurrence table is a matrix where the rows are the records from the subset table, and the columns each represent an {attribute, value} pair (denoted as attribute=value). Each row has a 1 for all the {attribute, value} pairs that occur for that record, and a 0 otherwise. Essentially, this table takes each record and turns it into a set of bits, where each 1 bit means the record contains that specific {attribute, value} pair. This is the key data structure ESCAPE uses to generate conjunctions efficiently.

The most important aspect of the binary occurrence table is that the columns are ordered from left-to-right in descending order of their Performance (we only care about relative orderings, so ties do not matter, and are broken arbitrarily). As we show below, this ordering property allows ESCAPE to discover fewer conjunctions that cover more records using exclusive attributes, thereby learning a more compact yet explanatory DNF rule. In our running example, the leftmost columns in the table is Origin=LAX because, it has the highest performance value, while the rightmost columns are those with low values, such as Origin=BWI. Note that Destination=JFK is not included as a possible column since it has been pruned out.

Figure 2 shows this translation from records to the binary occurrence table with the columns ordered by performance. We note that for space purposes, each column condenses the attribute in the {attribute, value} pair into its first letter.

Since the bits are ordered in this table according to {attribute, value} performance, each row represents a binary number such that a larger binary number implies that the record contains more of the high-performing pairs in the subset. ESCAPE uses this property of the table to minimize the number of conjunctions it puts into the final DNF rule, so that the final rule represents a compact explanation of the subset. That is, in the next step ESCAPE sorts the binary occurrence table, and then traverses the table row by row, finding the {attribute, value} pairs in common between the

<sup>2</sup>We note that ESCAPE is not tied to the performance metric above. In fact, any comparable metric from other sequential covering algorithms, such as information gain [10], could plug into our performance metric for experimentation.

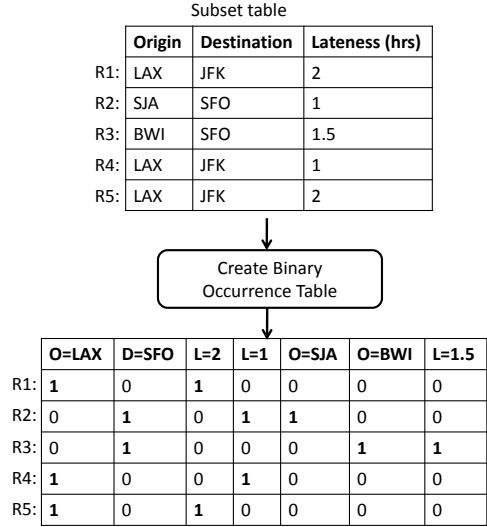


Fig. 2: Creating a binary occurrence table

rows and using these to build conjunctions. The conjunctions mined from the top of the table will cover more records than those mined from lower in the table, because the rules from the top will use the best performing {attribute, value} pairs in their conjunctions. This yields fewer mined conjunctions overall because the conjunctions that cover the most records are mined first. Another property of this sorting is that it places records near each other in the table that share certain pairs, as they will be grouped closely together during the sort since they translate to similar binary numbers.

After sorting, ESCAPE has turned the original subset records into a sorted binary-occurrence table, and is ready to mine the rules using a sequential covering. In this step, ESCAPE takes as input the newly sorted binary occurrence table, and returns a propositional logic rule in DNF form that explains the subset.

To perform the sequential covering, ESCAPE scans the sorted, binary occurrence table once, row-by-row from top to bottom, and compares each row to the next row below it, called its “neighbor.” Intuitively, if a row and its neighbor share some bits, those bits represent a conjunction that covers those two rows. Now, if the neighbor’s neighbor also shares bits with that conjunction, then this new set of shared bits will cover all three rows. At some point during the scan of the binary occurrence table, the current neighbor won’t share any bits with the current conjunction, at which point ESCAPE stores the conjunction, and starts the process over using this new neighbor as the starting conjunction.

This approach to sequential covering leverages the two important aspects of the sorted table. First, as stated above, the conjunctions ESCAPE discovers in the beginning will cover more records, because they will use the highest performing {attribute, value} pairs in them. This means fewer overall rules in the DNF, which leads to a more explicit rule. Second, the sorting procedure grouped the records in the binary

occurrence table such that neighbors share more common bits ({attribute, value} pairs). This is important because ESCAPE only needs to scan the table once, considering just neighboring rows to generate sufficient rules because the sorting clustered similar rows near each other.

To make the covering algorithm clear, Table 3 describes ESCAPE’s traversal down the sorted table, resulting in the final explanation:  $( [ O=LAX ] \cup [ D=SFO ] )$ . ESCAPE starts by taking the intersection between R1 and R5 (ESCAPE initializes the current conjunction to be R1), which are the first two rows in the sorted table. This intersection yields the bits representing the conjunction (Origin=LAX & Lateness=2). Next, ESCAPE compares the current conjunction (Origin=LAX & Lateness=2) with R4. This yields an updated conjunction (Origin=LAX). After that, ESCAPE compares the current conjunction (Origin=LAX) with R2. Since this yields an empty intersection, ESCAPE stores the conjunction (Origin=LAX) and sets the current conjunction to R2 (Destination=SFO & Lateness=1 & Origin=SJA). The covering continues until the whole table has been scanned, at which point the set of mined conjunctions is unioned together to form the final DNF rule that acts as the subset’s explanation.

Step 0:	Initialization	
Conjunction $\leftarrow$ R1: [ O=LAX & L=2 ]		
Step 1:	Conjunction vs. R5	
Intersection: [ O=LAX & L=2 ]		
Conjunction $\leftarrow$ [ O=LAX & L=2 ]		
Step 2:	Conjunction vs. R4	
Intersection: [ O=LAX ]		
Conjunction $\leftarrow$ [ O=LAX ]		
Step 3:	Conjunction vs. R2	
Intersection: {}		
Store Conjunction: [ O=LAX ]		
Conjunction $\leftarrow$ R2: [ D=SFO & L=1 & O=SJA ]		
Step 4:	Conjunction vs. R3	
Intersection: [ D=SFO ]		
Conjunction $\leftarrow$ [ D=SFO ]		
Step 5:	No more rows	
Store Conjunction: [ D=SFO ]		
Return DNF rule: [ O=LAX ] $\cup$ [ D=SFO ]		

## 2.1 Running time analysis

ESCAPE is simple, intuitive, and effective (as shown in our results). Further, as stated above, one of the key contributions of ESCAPE is its efficiency in generating the covering rule. In contrast to other sequential covering methods that are difficult to analyze formally, because ESCAPE is simple, it is easy to analyze. As we next describe, ESCAPE has a worst case performance of  $O((nA)\log(nA))$  time, assuming  $n$  records and  $A$  attributes. In general, for most data sets,  $n \gg A$  so our algorithm effectively runs in  $O(n\log n)$  time.<sup>3</sup>

### Step 1: Prune based on {attribute, value} performance

To calculate the performance values, ESCAPE first performs a single pass over both the subset table and the complement table and records the frequencies of each {attribute, value} pair. So

<sup>3</sup>We note that in our experimental data sets  $|n|$  is about 886 times greater than  $|A|$  on average.

it must examine all  $A$  attributes for all  $n$  records. Therefore, this takes  $O(nA)$  time.

Then, for those attributes in the subset it prunes away the pairs below the threshold  $T_c$ . We note that we want to examine *worst case* performance. Therefore, in the worst case, the entire subset table is the whole data set (i.e. all  $n$  records belong to the subset table), and each record in the table has a unique value for  $A$ . Therefore, the maximal number of distinct {attribute, value} pairs to consider is  $nA$ , and so this pruning takes  $O(nA)$  time. So, all of Step 1 takes  $O(2nA)$  time, which is linear.

### Step 2: Sort the {attribute, value} pairs

As we stated in Step 1, in the worst case there are  $nA$  distinct {attribute, value} pairs, and if none of them are pruned based on their performance values, ESCAPE has to sort all  $nA$  of these pairs. Using traditional sorting based on comparisons takes  $O((nA)\log(nA))$  time.

### Step 3: Build up binary matrix

For this step, assuming no pruning was done, our table has a maximum number of  $A$  columns, and  $n$  rows. So, for each row in our set  $n$ , ESCAPE records a 1 for all columns contained in that record. Therefore, for each row, ESCAPE makes  $A$  checks, and so building this table takes  $O(nA)$  time.

### Step 4: Sort the binary table

As noted above, each row in this table represents a binary number and can therefore be sorted using some comparison sorting method. Since there are  $n$  rows to sort, this sorting takes  $O(n\log n)$  time.

### Step 5: Sequential covering

ESCAPE traverses the sorted, binary table row by row, taking the intersection each time. So, for each row comparison, it checks  $A$  bits (columns) for the intersection. Since there are  $n$  rows, it essentially scans each attribute of each row, and so this step takes  $O(nA)$  time.

Therefore, the dominating aspects of our algorithm in terms of running time are the sorts, which take loglinear time. Since  $n \gg A$ , our algorithm effectively runs in  $O(n\log n)$  time in the worst case.

## 2.2 Parameter behavior

As stated previously in this section, our two parameters,  $w$  and  $T_c$  have well understood behavior. Now that our whole approach is clear, we can describe their behavior and effect on the mined DNF rule.

To start, we consider the penalty value  $w$  and its effect on our approach. To understand its impact, consider the extreme cases for this parameter. On one end,  $w$  can be set so high that only a few {attribute, value} pairs produce positive performance values and hence can be used in the conjunctions. In this case, since ESCAPE can only choose from these select few {attribute, value} pairs, it will generate very specific conjunctions in the final rule. However, these rules may not cover all of the records in the subset because they are so specific, since ESCAPE had a limited choice in which pairs to consider.

On the other hand, if  $w$  is so small that each pair is dominated by its  $P_c$  value (imagine  $w$  as a large negative number), then ESCAPE will use very generic {attribute, value} pairs in its rules. These rules will cover all of the records in the subset, but they will also cover many records in the complement set, and therefore this rule does not explain the subset well, since it is not specific to the records in the subset.

We demonstrate this behavior for changing  $w$  in our experiments, and show that our empirically chosen value for  $w$  (held at 7.5 across our experimental domains) produces rules that both use highly exclusive attributes (so the rules are specific to only the subset) while also covering the records in the subset (i.e. they are not overly specific).

The threshold  $T_c$  also has well understood behavior. Since  $T_c$  acts as a pruning threshold, the more {attribute, value} pairs it prunes, the fewer conjunctions in the final rule. Fewer conjunctions yield more “compact” explanations, which is a desired behavior. Again, it is useful to consider the extreme cases to understand the behavior of the threshold. If  $T_c$  is so high as to prune all {attribute, value} pairs, ESCAPE generates a maximally compact explanation (since it has no conjunctions) but at the cost of covering records in the subset (since it covers none). If it prunes no {attribute, value} pairs, it allows ESCAPE to consider any {attribute, value} pair, and therefore the rule will cover all of the records in the subset, but at the cost of using many more conjunctions. Therefore, this rule would not be compact enough to provide a useful explanation. Again, we demonstrate the behavior of varying  $T_c$  in our experiments, and show that our empirically chosen value of 0.0125 (i.e. 1.25%) produces sufficient rules.

### 3. Experiments

The above sections describe ESCAPE for generating explanation rules, and formally analyze the efficiency of our approach in terms of running time. Next, we empirically test whether ESCAPE generates “expressive” rules. There are two components to well expressed rules. First, the rules should be “compact.” That is, the fewer conjunctions in the rule to explain the subset the better, since fewer rules are easier to interpret, and because trivially ESCAPE could return each tuple in the subset as its own conjunction, making a maximally specific, but useless explanation. To make this definition concrete, we quantify “compactness” as:

$$Compactness = 1 - \frac{\|r\|}{\|n\|}$$

Where  $r$  is the number of conjunctions in our DNF rule, and  $n$  is the number of records in our subset. The bigger the compactness score, the better the rules are at generalizing over the subset in that fewer rules cover more of the subset. As an example, the rule generated in Table 3 has a compactness of 0.6 (60%) because it has two conjunctions (O=LAX) and (D=SFO) for the five tuples in the subset.

More importantly than compactness, however, is “coverage.” Coverage is the number of records in the subset covered by the learned DNF rule and is represented as the fraction of records covered by rule over the number of records in the subset:

$$Cov = \frac{|\#recordscoveredbyruleindataset|}{|\#totalrecordsinthedataset|}$$

There are actual two flavors of coverage to consider. First, is  $Cov_s$ , which is the coverage of the rule in the subset. Higher values of  $Cov_s$  are better because that means the

discovered rule explains the subset well, since it covers more of the records in the subset. The second flavor is  $Cov_c$ , which is the coverage in the complement to the subset. In this case, lower values of  $Cov_c$  are desired because the explanation rule should be as specific to the subset as possible. That is, a high value for  $Cov_c$  means that the rules explains not just the subset, but also records in the table in general (which would not be very useful). Therefore, an “expressive” rule is one that has high compactness, high values for  $Cov_s$  and low values for  $Cov_c$ .

To test ESCAPE we use data sets from two domains. The first domain is the “Flights” domain. This domain consists of 48,629 records about flights culled from the flightaware website and joined with information from the FAA.gov website. This combined data set contains 31 attributes such as origins, destinations, routes, operator information (e.g. company and plane information), and flight information such as arrival times, delay times, altitude, speed, etc.

From this data set, we generated 10 subsets of data. These subsets are defined for given attributes of interest as either “Above” or “Below” where “Above” means the z-score for that attribute value was greater than 2.57 and “Below” means the z-score was less than -2.57. For instance, one subset is “Arrival Delay Above,” which means all members were selected because the delay in their arrival time attributes was large. The z-score approach is a simple metric for defining “outliers” because the probability of a value having a z-score above 2.57 is roughly 0.5% and similarly for a z-score below 2.57 (assuming normal distributions). Therefore, these are interesting subsets to try and explain. Table 4 describes these subsets in detail.

Table 4: Subsets for our Flight data

Subset Name	Subset size	Complement size
Actual Duration Above	1,156	47,473
Actual Duration Below	7	48,622
Arrival Delay Above	1,324	47,305
Arrival Delay Below	317	48,312
Departure Delay Above	618	48,011
Departure Delay Below	370	48,259
Planned Altitude Above	607	48,022
Planned Duration Above	414	48,215
Planned Speed (kts) Above	6	48,623
Planned Speed (mach) Above	30	48,599

Our second domain is the “Cars” domain. This domain consists of 4,703 records about cars culled from fueleconomy.gov website and joined with information from MSN’s auto site. This combined data set contains 23 attributes such as the car’s make, model, engine size, car type, fuel economy, price, user rating, and crash ratings. From this data set, we created 10 subsets by selecting records that share an interesting property, such as having a 5-star frontal crash rating. The subsets for this domain, along with their criterion are given in Table 5

We note that we created the data for each domain by joining together interesting sources, to yield richer sets of attributes. Beyond making explanations more interesting, it also makes it possible that the data is no longer iid, which makes our data conform to more realistic settings. Our experiments, then, use the subsets described in Tables 4

Table 5: Subsets for our Cars data

Subset Name/Condition	Subset size	Complement size
5-star frontal crash rating	1,155	3,548
1 or 2-star frontside crash rating	106	4,597
5-star frontside crash rating	698	4,005
MPG rating $\leq 15$	731	3,972
MPG rating $\geq 30$	1,086	3,617
5-star frontal-passenger rating	975	3,728
Price $\leq$ \$2,000	314	4,389
Price $\geq$ \$75,000	53	4,650
1 or 2-star rear-side crash rating	72	4,631
5-star rear-side crash rating	546	4,157

and 5, and generate rules to explain them using our approach. Note that as stated we set  $w$  to 7.5 and  $T_c$  to 1.25%.

As a baseline, we compare ESCAPE to RIPPER [5].<sup>4</sup> RIPPER is one of the most popular rule learners that builds a classifier out of the rules it discovers. Although traditional rule-learning classifiers must assume iid data, we still believe RIPPER is an interesting baseline to compare against as one of the state-of-the-art covering algorithms for learning a classifier.

Since RIPPER does learn a classifier, it requires positive and negative training data. So we train it on the subsets defined above, treating members of the subset as positive examples, and members of the complement as negative examples. Note, however, that given the large number of {attribute, value} pairs for our datasets, we could not feed the whole complement set to our implementation of the algorithm for the flights domain. Therefore, for that domain, we had to limit the size of the complement set to 5,000 randomly chosen records. However, since we use probabilities in the performance metric, sampling is sufficient for this study. We note that this issue did not affect the cars domain.

For each of the rules learned by each approach for each domain, we calculate the Compactness,  $Cov_s$ , and  $Cov_c$  to demonstrate how the discovered rules explain the data. We then average these metrics for all experimental subsets, and present the results comparing the methods Table 6.

Table 6: Results of ESCAPE and RIPPER

Flights Domain	Compactness	$Cov_s$	$Cov_c$
ESCAPE	88.78	75.44	2.35
RIPPER	85.25	56.90	12.41
RIPPER (exclude missing)	94.73	63.22	2.68
Cars Domain			
ESCAPE	97.87	100.00	1.42
RIPPER	98.47	98.30	0.06

First, we point out that for the “Actual Duration Below” subset, RIPPER did not actually learn a rule. Therefore, when we report the averages for RIPPER in the Flights domain, we report both the actual averages from each column and also the averages when “excluding missing,” which excludes this subset from the average calculations since it affects them so adversely. Regardless, in the flights domain ESCAPE discovers rules with a much higher  $Cov_s$  (over 12% difference), while maintaining a smaller  $Cov_c$  value. This means that ESCAPE is in fact generating more expressive rules that are more specific to the subsets due

<sup>4</sup>We use the JRip implementation in Weka [11] with default parameters

to the smaller  $Cov_c$ , but the rules also explain many more records in the subset, since the  $Cov_s$  score is so much higher. We note that both algorithms also do a good job of generating compact explanations (fewer conjunctions) because they both generate high scores for Compactness, although RIPPER’s Compactness score is higher. However, while RIPPER does have a higher Compactness score, this is largely due to the fact that it is learning fewer, restrictive rules that generate a much lower  $Cov_s$  value.

In the Cars domain, we again see that the rules mined by ESCAPE cover the subset records extremely well, with a perfect  $Cov_s$  value, that is slightly higher than RIPPER’s. Also, it has a very low value for  $Cov_c$  and so does not over generalize. Again, both methods do an excellent job in terms of compactness. Given that both methods perform so similarly on the Cars data, and they both do so well, we conclude that this is a much easier data set than the Flights data. Therefore, based on ESCAPE’s results, especially in the Flights domain, we show that ESCAPE can mine very specific and compact explanations for subsets of data.

As stated in our algorithmic description above, ESCAPE includes two parameters,  $w$  and  $T_c$ . Previously we described how ESCAPE’s behavior differs based on changes to these values, and here we show this to be the case empirically. Table 7 shows the average values of Compactness,  $Cov_s$ , and  $Cov_c$  using ESCAPE with varying values for  $w$  and  $T_c$ . The first row of the table shows  $w = 7.5$  and  $T_c = 1.25\%$ . These are the settings we use to test ESCAPE against RIPPER above.

The second row maintains  $w$  at 7.5 but drops  $T_c$  down to 0%, to test the effect of making  $T_c$  much less restrictive in terms of which {attribute, value} pairs ESCAPE can use for mining. While the Cars domain (the easier one) shows no difference, in the Flights domain we see the expected behavior. Since more {attribute, value} pairs are available to ESCAPE, it learns more conjunctions, and therefore covers more records in the subset. This yields the high value for  $Cov_s$ . (Note that these are specific conjunctions too because  $Cov_c$  does not change). However, generating more rules comes at the expense of Compactness which drops dramatically. Therefore, as we described previously, by lowering  $T_c$  more pairs become available which yields a better covering, though it does so by generating many more rules and losing compactness. Indeed, this is the expected behavior for this parameter.

The final row restores the  $T_c$  value to 1.25%, but drops the value of  $w$  to 0. In this case, each {attribute, value} pair’s performance is essentially the raw difference in probabilities of occurring in the subset table versus the complement table. This makes the pairs that occur in both tables have higher performance values, and this is reflected in the result. Specifically, in both domains the  $Cov_c$  value increases dramatically because the generated conjunctions use {attribute, value} pairs that are not exclusive to the subset but that occur frequently in the complement set as well. Of course, in the Flights domain we see an increase in  $Cov_s$  too, since these are popularly occurring attributes in the rules, but this is at the expense of  $Cov_c$  and so these explanations are not

specific and thus not very useful. However, this does show the expected behavior for changing  $w$ .

We note that these results also justify our empirical choices for  $w$  and  $T_c$  as these values lead to a good mix of compact rules that still yield great subset table coverage and low complement table coverage, across both domains. Also, by understanding the behavior of these parameters, users can tune them to their own specific needs. For instance, when looking at records where  $\text{Cov}_s$  is chiefly important, even at the expense of Compactness, a user may want to tune down  $T_c$ . Such a case might occur when the records in the subset are anomalous financial transactions, for instance, and the goal is to explain as many of the records as possible.

Table 7: Results varying  $w$  and  $T_c$

Domain: <b>Flights</b>				
$w$	$T_c$	Avg. Compactness	Avg. $\text{Cov}_s$	Avg. $\text{Cov}_c$
7.5	1.25%	88.78	75.44	2.35
7.5	0.0%	59.18	99.78	2.38
0.0	1.25%	89.97	98.08	46.34

Domain: <b>Cars</b>				
$w$	$T_c$	Avg. Compactness	Avg. $\text{Cov}_s$	Avg. $\text{Cov}_c$
7.5	1.25%	97.87	100.00	1.42
7.5	0.0%	97.82	100.00	1.42
0.0	1.25%	98.99	100.00	33.97

## 4. Related Work

At its core, our approach to generating explanations for subsets of data uses rule learning of which there are many flavors [4], [5], [10]. However, it has been noted in the past that, “the rule learning systems that perform best experimentally have the disadvantage of being complex, hard to implement, and not well-understood formally. [6]” Our approach, on the other hand is simple and well understood formally. In fact, we can show that our algorithm runs on loglinear time, which is on par with other fastest rule learner [6], but unlike that method we can formally show our method to do so, without relying on an experimental analysis of running time.

Given our emphasis on using the most commonly occurring {attribute, value} pairs, it may seem that our approach is similar to the “apriori” algorithm for mining association rules using frequent item sets [1]. However, there are a few major differences between our work and that. First, ESCAPE is built to explain a subset of data. That is, the rules it generates are those that differentiate the subset from the rest of the data set. This is in contrast to association rules where the goal is to find associations between one item set and another. We make no claims about the associations between our rules, other than that they describe members of a subset. Further, that work is about finding associations between items from a single set of transactions in an unsupervised manner. In some sense, our approach is supervised because we leverage set members and their complements, and the goal is to discover what separates the subset from its complement. So in this sense, the problems solved are fundamentally different.

Lastly, work that is similar to ours in spirit is the research on discovering rules to describe more opaque machine

learning models such as SVMs or Neural Networks [8], [2], [9], [3] In this context, the research aims to have the machine produce rules that describe how other, opaque machine learning models behave. In some sense, the rules are “explaining” the machine learning models, which is similar our explaining of data subsets. However, although the approaches are similar in spirit, our problem is fundamentally different. First, our explanations focus on describing why a subset of records is fundamentally different than the complement set, while this other work aims to discover rules that describe the behavior of another system, but these rules do not show a contrast between this system and another. Also, our approach is built for efficiency while generating descriptive rules, while these approaches focus on generating the descriptive rules only, since this is an extremely difficult problem in this space. Lastly, the approaches are very different in that our method takes as input a set of records and produces rules to explain the records, while this other field of research takes both models and records as input to produce rules to explain the models rather than the data.

## 5. Conclusions

In this paper we presented ESCAPE, the first sequential covering algorithm that can be formally shown to run in loglinear time in general. Our approach emphasizes both learning descriptive rules, and doing so efficiently.

## References

- [1] R. Agrawal, T. Imielinski, and A. Swami, “Mining association rules between sets of items in large databases,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 1993, pp. 207–216.
- [2] R. Andrews, J. Diederich, and A. B. Tickle, “A survey and critique of techniques for extracting rules from trained artificial neural networks,” *Knowledge Based Systems*, vol. 8, pp. 373–389, 1995.
- [3] N. Barakat and J. Diederich, “Learning-based rule-extraction from support vector machines,” in *Proceedings of the International Conference on Computer Theory and Applications*, 2004.
- [4] P. Clark and T. Niblett, “The cn2 induction algorithm,” *Machine Learning*, vol. 3, no. 4, pp. 261–283, 1989.
- [5] W. W. Cohen, “Fast effective rule induction,” in *Proceedings of the International Conference on Machine Learning*, 1995.
- [6] W. W. Cohen and Y. Singer, “A simple, fast, and effective rule learner,” in *Proceedings of AAAI*, 1999.
- [7] J. Fürnkranz and G. Widmer, “Incremental reduced error pruning,” in *Proceedings of the International Conference on Machine Learning*, 1994.
- [8] S. A. Macskassy, H. Hirsh, F. Provost, R. Sankaranarayanan, and V. Dhar, “Intelligent information triage,” in *Proceedings of the International Conference on Research and Development in Information Retrieval*, 2001.
- [9] H. Nunez, C. Angulo, and A. Catala, “Rule-extraction from support vector machines,” in *Proceedings of the European Symposium on Artificial Neural Networks*, 2002.
- [10] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1994.
- [11] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.