Algorithmic Infrastructure and Performance of Finding Optimal Conditional Sparse ℓ_p-norm Regression

John Hainline

Fall 2018

1 Introduction

1.1 Overview

The focus of this project was to implement a particular complex algorithm in a performant, robust, and scalable way. The algorithm was specifically designed to solve the *conditional sparse ℓ_p-norm regression problem*. I managed the requirements for this system using libraries contained in the Scala language, and scaled the system using the Google Kubernetes Engine.

The *conditional sparse ℓ_p-norm regression problem* can be motivated by first considering the following related task of trying to find a subset of data on which sparse ℓ_p-norm regression succeeds. This occurs where there is a *linear rule* with a small number of terms that *predict a response variable* accurately in the ℓ_p sense (where l_2, the squared error, is our primary focus).

Unfortunately any linear predictor we find may not be a good predictor of a new point. This is because we cannot be sure if the new point belongs in the subset that the linear predictor applies to. However, if we have the explicit description of the subset where our predictor works, we can tell whether or not this new point belongs to that subset, and so we know whether or not we can use the predictor. Moreover, an explicit subset description makes it possible to get a statistical guarantee that our prediction will, in expectation, be accurate with high probability.

A real world application of this would be discovering patterns of medical conditions that are likely to occur in a sub-population, but that rarely occur in the population as a whole. The vulnerabilities of a particular race, age group, sex, or combination of these, could be more easily ferreted out of a large and complex data set. Not only that, but an explicit description gives us an easily understood rule on which particular portions of the population this pattern of medical issues apply.

1.2 Problems

Unfortunately, searching arbitrary subsets of data is a combinatorial problem and quickly becomes intractable for even moderate data set size. However a method was proposed in an earlier work [Juba, 2017] to solve this problem using a loss function of the ℓ_p-norm and a conditional description using k-DNFs (described later).

The problem was defined as follows.
We are given a matrix $\boldsymbol{Y} \in \mathbb{R}^{n \times m} = (\vec{y_1}, \vec{y_2}, \ldots, \vec{y_n})^T$ containing n vectors of size m and a prediction target $\boldsymbol{Z} \in \mathbb{R}^n$. We construct a boolean matrix $\boldsymbol{X} \in \mathbb{B}^{n \times p} = (\vec{x_1}, \vec{x_1}, \ldots, \vec{x_n})^T$ containing n vectors of size p. We derive $\boldsymbol{X}$ from $\boldsymbol{Y}$ such that our $\vec{x_i}$ partitions $\vec{y_i}$ into a p dimensional space. How this partitioning is done is up to the user, but tends to be by forming many boolean "buckets" for the m variables in $\vec{y_i}$. We define c as a k-DNF conditional [1] that, when applied to $\boldsymbol{X}$, gives us a sub-population. So $c(\boldsymbol{X})$ is a "filter" on $\boldsymbol{Y}$, reducing our full population of n to some smaller number. On this sub-population we can find a linear predictor for $\boldsymbol{Z}$ defined as $\langle \vec{a}, \boldsymbol{Y} \rangle$. So the coefficients $\vec{a}$ define a sparse regression line on $\boldsymbol{Y}$ which ideally predicts $\boldsymbol{Z}$ with small loss.

We restrict ourselves to k-DNFs for our conditional because if we could find conditions with arbitrary conjunctions this would yield PAC-learning algorithms for general DNFs. Juba's earlier work [Juba, 2017] makes clear that it is unlikely any algorithm can hope to find conditions of this kind. So we restrict ourselves to k-DNFs, which are a natural and powerful class of boolean representations that come with useful theoretical guarantees.

1.3 Basic Solution

Our implementation used the Scala language. We dealt with the scale of this problem by designing our code to run on the Google Kubernetes Engine (GKE). We synchronized many machines using the Akka Cluster library, which provides a powerful, managed, peer-to-peer clustering system. To ensure a robust solution we used Akka's Reactive Streams implementation which has very flexible error-handling controls. We also attached this system to a MySQL database to enable saving of all our results, which provided us with safety should the system ever fail during its calculations.

[1] k-Disjunctive Normal Form (k-DNF) is an OR of ANDs of at most k *literals*, where a literal is either a Boolean attribute or the negation of a Boolean attribute. See `https://en.wikipedia.org/wiki/Disjunctive_normal_form`.

2 Algorithm

2.1 Finding Conditional

At a high level, the conditional $\boldsymbol{c}$ is an accumulation of some number of boolean "terms" that are OR'ed together. We use $\mathbf{2}$-DNFs in our experiments, which means each "term" is an AND of $\mathbf{2}$ boolean variables. A valid $\mathbf{2}$-DNF could look something like

$$\boldsymbol{c} = (\boldsymbol{x_1} \wedge \neg \boldsymbol{x_2}) \vee (\boldsymbol{x_1} \wedge \neg \boldsymbol{x_3}) \vee (\boldsymbol{x_2} \wedge \boldsymbol{x_4}) \tag{1}$$

or in *set* notation

$$\boldsymbol{c} = (\boldsymbol{x_1} \cap \neg \boldsymbol{x_2}) \cup (\boldsymbol{x_1} \cap \neg \boldsymbol{x_3}) \cup (\boldsymbol{x_2} \cap \boldsymbol{x_4}) \tag{2}$$

Recall matrix $\boldsymbol{X} \in \mathbb{B}^{n \times p}$, describes $\boldsymbol{Y} \in \mathbb{R}^{n \times m}$. So the *set* notation of $\boldsymbol{c}$, when applied $\boldsymbol{c}(\boldsymbol{X})$, describes a sub-population of $\boldsymbol{Y}$. Note that in our $\mathbf{2}$-DNF there are a total of $\binom{\boldsymbol{p}}{\mathbf{2}}$ possible $(\boldsymbol{x_i} \cap \boldsymbol{x_j})$ terms. In fact there actually are $\mathbf{4} \cdot \binom{\boldsymbol{p}}{\mathbf{2}}$, since we must account for $(\boldsymbol{x_i} \cap \boldsymbol{x_j})$, $(\neg \boldsymbol{x_i} \cap \boldsymbol{x_j})$, $(\boldsymbol{x_i} \cap \neg \boldsymbol{x_j})$, and $(\neg \boldsymbol{x_i} \cap \neg \boldsymbol{x_j})$.

We go about finding our condition $\boldsymbol{c}$ by searching across all possible regression lines $\vec{\boldsymbol{a}}$ defined by our points $\boldsymbol{Y}$ and for each we construct the best $\boldsymbol{k}$-DNF possible. The $\boldsymbol{k}$-DNF with the lowest error overall is returned.

To find a regression line $\vec{\boldsymbol{a}}$ on $\boldsymbol{Y}$ we need $\mathbf{2}$ points and $\mathbf{2}$ dimensions of $\boldsymbol{Y} \in \mathbb{R}^{\boldsymbol{n} \times \boldsymbol{m}}$. So we need to select $\mathbf{2}$ of $\boldsymbol{n}$ points and $\mathbf{2}$ of $\boldsymbol{m}$ dimensions. Unfortunately there is no easy way of picking which $\mathbf{2}$ dimensions to use, so we will need to check all $\binom{\boldsymbol{m}}{\mathbf{2}}$ combinations. Likewise we have no idea which points may give us an ideal $\boldsymbol{c}$ and so we check all $\binom{\boldsymbol{n}}{\mathbf{2}}$ combinations.

2.2 Red-Blue Set Cover

Now that we have a regression line $\vec{\boldsymbol{a}}$, we need to determine the best $\boldsymbol{k}$-DNF $\boldsymbol{c}(\boldsymbol{X})$ that defines a sub-population of points in $\boldsymbol{Y}$ that are near the regression line. This $\boldsymbol{c}$ can then be used to predict $\boldsymbol{Z}$. Keeping in mind our earlier *set* notation, we can see that each $(\boldsymbol{x_i} \cap \boldsymbol{x_j})$ term in our $\boldsymbol{k}$-DNF becomes a *set* of included points in $\boldsymbol{Y}$. Now we need to find which terms to include in $\boldsymbol{c}$. To accomplish this, we employ a derivation of the Red-Blue Set Cover algorithm [Carr et al., 2000] to find the best terms for $\boldsymbol{c}$ that fit $\vec{\boldsymbol{a}}$.

The Red-Blue Set Cover problem is defined as follows.

> Given a finite set of "red" elements $\boldsymbol{R}$, and a finite set of "blue" elements $\boldsymbol{B}$ and a family $\boldsymbol{S} \subseteq \mathbf{2}^{\boldsymbol{R} \cup \boldsymbol{B}}$, the red-blue set cover problem is to find a subfamily $\boldsymbol{C} \subseteq \boldsymbol{S}$ which covers all blue elements, but which covers the minimum possible number of red elements [Carr et al., 2000].

In our case we use a greedy algorithm to find an approximate solution to this Red-Blue Set Cover problem. We consider the red-blue spectrum of a point to be its distance from $\vec{a}$. This is represented with a single value of increasing "redness" for every point, and we try to minimize the amount of "redness" introduced while maximizing the number of points included.

To start applying our solution to Red-Blue Set Cover, we first compute a "redness" value for every point $\vec{y_i} \in \boldsymbol{Y}$. We then construct the total "redness" for each $(\boldsymbol{x_j} \cap \boldsymbol{x_k})$ term and sort, successively taking the least "red" terms until reaching a balance between "redness" and point count.

Unfortunately, this will ultimately call our Set Cover algorithm $\binom{n}{2} \times \binom{p}{2}$ times. For even a moderately small data set with $\boldsymbol{n = 1000, p = 6}$ this becomes $\binom{1000}{2}\binom{6}{2} = \mathbf{499{,}500 \times 15 = 7{,}492{,}500}$ Set Cover calls. The scaling issues are obvious.

2.3 Scaling

To deal with this we can simply sub-sample our $\boldsymbol{Y}$. For many real world distributions, random sampling (assuming the sub-sample is not extremely small) will still find our $\boldsymbol{k}$-DNF with high likelihood. The actual statistics is not broken down in this paper, but we very rarely failed when sub-sampling $\mathbf{200}$ out of $\mathbf{10{,}000}$ points. This sub-sampling is one of the **single biggest factors** in speeding up our calculations as it can reduce the number of *SetCover* calls by orders of magnitude, while only marginally reducing our probability of missing a hidden $\boldsymbol{k}$-DNF.

See algorithms 1 and 2 for a pseudo-code implementation.

3 Implementation

We decided to write the system in the Scala programming language. This was due to language familiarity, and for the *Akka framework*, which facilitates computation across clusters of PCs. We specifically used the Akka Cluster[2], Akka Actor[3], and Akka Stream[4] libraries.

[2] `https://doc.akka.io/docs/akka/current/common/cluster.html#intro`

[3] `https://doc.akka.io/docs/akka/current/actors.html#introduction`

[4] `https://doc.akka.io/docs/akka/current/stream/stream-introduction.html`

3.1 Akka Library

3.1.1 Akka Actor

Akka *Actors* provide a high level abstraction for creating concurrent and distributed systems. The basic design is extremely simple. An *Actor* is simply an object that sends and receives messages using *mailboxes* (preventing concurrent access). *Actors* are stateful, but do not alter external state, so locking and other concurrency mechanisms are unnecessary. There are numerous other features of *Actors*, but at a high level this is what is important.

3.1.2 Akka Cluster

The Akka *Cluster* library provides a peer to peer based cluster membership service. It is fault tolerant and decentralized. We use this to establish an initial connection between servers. The Akka *Cluster* library also provides an easy mechanism for passing events or *Actor* messages across any/all members in the cluster, easily bridging any network boundaries.

3.1.3 Akka Streams

Akka *Streams* follow the "Reactive Streams" initiative. This standard tries to provide asynchronous stream processing with non-blocking back pressure. When trying to solve our k-DNF discovery problem, we utilize this API to provide a cross-machine work pulling strategy for running through all possible sparse regression lines $\vec{a}$ in $\boldsymbol{Y}$.

3.2 Architecture

The application is designed to be run either *locally*, or on the *Google Kubernetes Engine* (GKE). When running locally, events follow this pattern:

1. User sends a workload to the system. This names the data set the system will run, and provides relevant variables (such as μ and the subset size).
2. The ManageActor (an Akka Actor) receives the workload and:
   (a) Loads ($\boldsymbol{X}$, $\boldsymbol{Y}$, $\boldsymbol{Z}$) from the database.
   (b) Prepares database tables for saving this workload's results.
   (c) Creates Akka Stream objects, specifically BalanceHub and MergeHub, to handle work flowing through the system.
   (d) Publishes the workload to the event manager.

3. The ComputeActor receives the workload and:

   (a) Loads $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z})$ from the database.

   (b) Instantiates a number of SetCover objects (Akka Stream objects) that each take in work and give a result. These operate *in parallel*.

4. The BalanceHub, once a SetCover instance has been attached, begins sending work through the system.

For a visual breakdown of this process, see figure 1. By having the Akka Actors create Akka Streams independently, we can dynamically create and destroy SetCover instances, scaling up or down the number of CPUs we use.

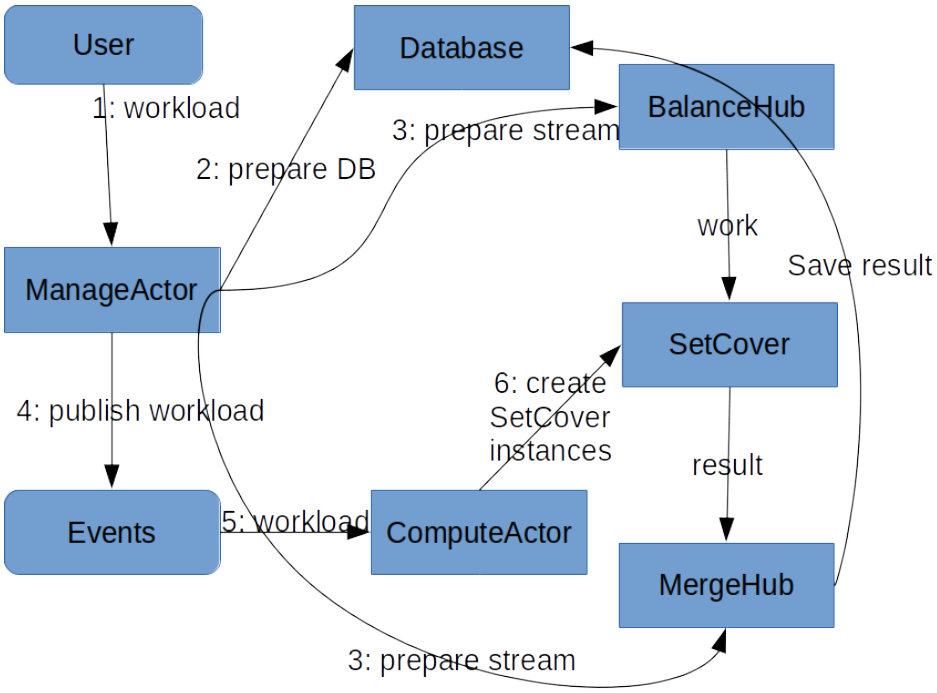

Figure 1: (1) The user sends a workload to the system. The ManageActor receives the workload and: (2) Prepares database tables for the workload. (3) Creates Akka Stream objects, specifically BalanceHub and MergeHub, to handle flowing work through the system. (4) Publishes the workload to the event manager. The event manager (5) sends the workload to the ComputeActor. Finally, the ComputeActor (6) instantiates some number of SetCover objects that each take in work and give a result.

For running in GKE, it should be noted that both Akka Actors as well as Akka Streams are capable of passing messages to separate machines on the network. Simply by instantiating ComputeActors on other machines, we get message passing over the network (rather than locally) which allows an entire cluster of machines to participate.

4 Experimental Evaluation

The code itself can be found at `https://github.com/johnhainline/sclr`. The latest design is able to run a data set with $\boldsymbol{X} \in \mathbb{B}^{\mathbf{10000 \times 50}}, \boldsymbol{Y} \in \mathbb{R}^{\mathbf{10000 \times 20}}$ (so $\boldsymbol{n = 10000, m = 20, p = 50}$) and sub-sampling of $\mathbf{500}$ points. This meant $\boldsymbol{terms.size = 4 \cdot \binom{50}{2} = 4900}$ and *SetCover* ran $\binom{\mathbf{20}}{\mathbf{2}} \cdot \binom{\mathbf{500}}{\mathbf{2}} = \mathbf{27{,}702{,}500}$ times.

We successfully ran $\mathbf{26}$ machines on Google Kubernetes Engine (GKE), each with $\mathbf{8}$ CPU cores. The entire run took roughly five days. We had a single instance of the *ManageActor* and Database running on $\mathbf{1}$ machine, while the other $\mathbf{25}$ all ran a single instance of the *ComputeActor*. When a workload reached the *ComputeActor* on each machine, they instantiated $\mathbf{8}$ *SetCover* instances apiece. The *ManageActor* used the *BalanceHub* to send work to the *SetCover* instances as fast as they could process data. The *ManageActor* used the *MergeHub* to collect all results from the *SetCover* instances and saved them in the database in batches. This gave us $\mathbf{200}$ CPUs each running *SetCover* in parallel at near 100% utilization.

4.1 Optimizations

4.1.1 Sub-sampling

Sub-sampling was our single greatest optimization. By reducing $\boldsymbol{n}$ to some small number $\boldsymbol{s}$ using random sampling we often run multiple orders of magnitude fewer *SetCover* calls than we would otherwise. This reduction, for many real world point distributions, only marginally reduces our probability of recovering an embedded $\boldsymbol{k}$-DNF (assuming one exists in the data set).

4.1.2 Set Implementations

Originally we used a standard *Set* implementation for the $\boldsymbol{terms}$ variable in *SetCover*. As we had sets that were thousands of items long, it quickly became apparent that a BitSet implementation was necessary. This simple change increased performance roughly by a factor of $\mathbf{10}$.

4.1.3 Embedded Native Code

At one point we experimented with using Java Native Interface (JNI) to embed native C++ code, going so far as to implement *SetCover* in C++ and create bindings from Scala. While we did make this work, the inherent overhead in calling a C++ function from the JVM was so high that it was not performant. If we had used a different batching strategy (such that the C++ code was called once for a large group of SetCover calls) this may have become a viable strategy. Unfortunately this change would have required a rather drastic refactor of Scala code and so we pursued other methods instead.

4.1.4 Batching

The Akka Streams implementation allows easy batching of messages as they travel. We began batching messages that had to travel across the network (i.e. to/from the *SetCover* instances). We also batched our database save calls. These changes gave us a roughly $\mathbf{10}\boldsymbol{x}$ speed up.

4.2 Recovery

4.2.1 Database

During a workload, every run of *SetCover* saves any resulting discovered $\boldsymbol{k}$-DNFs to the database. Each record also contains all the relevant variables passed to *SetCover* (specifically $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}, \boldsymbol{l}$). With this information it is possible to know how far along we were in our computation and potentially resume after a catastrophic system shutdown (assuming the database still exists).

4.2.2 Sub-sampling

Because we sub-sample $\boldsymbol{Y}$, attempts at resuming a failed workload could generate a completely different sub-sample and be unreproducible. We avoid this by allowing a random number generator seed to be passed in with the workload, which is used when sub-sampling $\boldsymbol{Y}$.

4.3 Limitations

This algorithm's limitations are primarily derived from the non-parallelizable portions of the code. Specifically, sending information across the network as well as saving to the database. The latest version of code has reduced these issues using batching.

While we are sure there is a limit to the number of parallel *ComputeActor* machines (and therefore *SetCover* instances) we can work with, we have not reached it yet. As our largest run of the program took advantage of $\mathbf{200}$ CPUs simultaneously, we were afraid to bring a great deal more machines online due to the large cost in GKE. Earlier versions of our algorithm could only handle parallelization factors of $\mathbf{30}$ (before an improved *SetCover* implementation) and then $\mathbf{100}$ (before batching). Based on our experience at this point, it seems likely we could scale to somewhere around $\mathbf{300 - 400}$ CPUs before needing to further optimize.

4.4 Future Work

Running on a $\mathbf{10{,}000}$ point data set took about five days with roughly $\mathbf{200}$ CPUs. We still wish to increase the allowable data set size even further. There is still likely another order of magnitude speed up attainable in *SetCover*. However moving away from GKE and instead writing the system to easily run on a local computer cluster would probably yield greater dividends. Using thousands of CPUs on Kubernetes is of course possible, but local resources will certainly be far cheaper.

Algorithm 1: This algorithm breaks down the relevant search space of $\boldsymbol{X} \in \mathbb{B}^{n \times p}$ and $\boldsymbol{Y} \in \mathbb{R}^{n \times m}$, giving SetCover the parameters it needs. Note that $\boldsymbol{terms}$ must include the $\neg$ versions of our $\binom{p}{2}$ possible combinations of $\boldsymbol{X}$, which is why we have $4 \cdot \binom{p}{2}$ combinations.

```
def ComputeBestKDNF(X, Y, Z, r):
    X, Y = SubsamplePoints(X, Y, r)
    terms = (x_i ∩ x_j) for all 4 · (p 2) combinations of X
    best_terms = ∅
    best_error = ∞
    for (i, j) ∈ (r 2) points do
        for (k, l) ∈ (m 2) dimensions do
            t, e = SetCover(X, Y, Z, terms, i, j, k, l)
            if e < best_error then
                best_terms = t
                best_error = e
            end
        end
    end
    return best_terms, best_error
```

Algorithm 2: The SetCover algorithm searches for the best k-DNF (lowest error in terms of "redness") for a given pair of data points i, j and dimensions k, l in $Y \in \mathbb{R}^{n \times m}$. The variable $terms$ holds every *set* generated by our $\binom{p}{2}$ terms $(x_i \cap x_j)$ in $X \in \mathbb{B}^{n \times p}$. μ dictates how large of a sub-population we are looking for.

def *SetCover(*X*,* Y*,* Z*,* $terms$*,* i*,* j*,* k*,* l*)*:
    $\vec{a} =$ CalculateRegressionLine$(\vec{y}[k]_i, \vec{y}[l]_j)$
    $r\_points =$ CalculateRednessForPoints$(Y, \vec{a})$
    $r\_terms =$ CalculateRednessForTerms$(terms, r_points)$
    $sort =$ SortByAverageRedness(r_terms)
    $best\_terms = \emptyset$
    $best\_error = \infty$
    $cur\_terms = \emptyset$
    while $(cur\_terms.size < \mu \cdot n)$ **do**
        $next =$ NextUnusedTermInSort$(sort, cur_terms)$
        $cur\_terms.add(next)$
    end
    $cur\_error =$ AverageRednessPerPoint(cur_terms)
    if $cur\_error < best\_error$ **then**
        $best\_error = cur\_error$
        $best\_terms = cur\_terms$
    end
    return $best\_terms, best\_error$

References

[Carr et al., 2000] Carr, R. D., Doddi, S., Konjevod, G., and Marathe, M. (2000). On the red-blue set cover problem. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, pages 345–353, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

[Juba, 2017] Juba, B. (2017). Conditional sparse linear regression. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 67. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. Proc. 8th ITCS.