PPDM Preprocessor for Weka Files using Datafly

Nidhin Pattaniyil

Eitan Romanoff

Table of Contents

Introduction

In the field of data mining, preserving sensitive data while maintaining an accurate model is crucial. For this project, given a dataset of donation information, we were tasked to create a preprocessor for .arff files, implementing custom security tags, to perform modifications on the data in an attempt to approach this problem. Questions are not answered in order as they appear on the assignment sheet, as they were re-ordered in an order that flows better in the final report.

Data Cleaning (Q. 1)

As a prerequisite for using the dataset given in our preprocessor, the data was required to be cleaned. Cleaning the data was done with two parallel approaches. Firstly, a java program was created so that particular anomalies, such as outliers, or incorrect number of attributes, could be easily detected through the usage of maps. Secondly, the dataset was loaded into an SQL database, and queries directed towards the validity of the dataset's contents could be carried out. Lastly, examinations of particular individual rows in question were performed by using Microsoft Excel.

As an initial approach, rows with an incorrect number of attributes were identified. Upon close examination, using both methods, rows 475 and 1122, both had "null values" in various columns. When brought up in Excel, it was clear, given the context of the values in other columns, that the values were "pushed" to the right, resulting in one attribute too many being parsed by the programs, and the addition of null values. After

these values were identified, the java program then looked for non-integer values, and logged the row numbers. Two instances were found, one on row 1389, and the other on 916.

Many various solutions to correcting the incorrect and missing values were proposed, but after considering the relatively large dataset, these four rows were simply just discarded. The impact of removing these four values is less pronounced, as one of the data tuples was to be pruned anyway due to a security code of 2, and the other three in light of 6000 instances seemed insignificant.

Preprocessor Design (Q. 2)

The next requirement in the project was to create a preprocessor that works on .arff files, such that it could handle a custom tag, `@secure`. The implementation was fairly simple. All attempts were made on instances that contained a security code of 0 or 1. While instances with security code 2 were removed entirely.

The first true purpose of the pre-processor was to identify instances with a security attribute of “1”, those who wished to have their data used in the overall model, but who felt their data was sensitive, and wanted privacy with regards to identifying their customer identifier with their trends on donations. Sensitive attributes are clearly marked in the .arff file as follows:

```
@secure [attribute]
```

where [attribute] is the name of the attribute needing obfuscation on tuples who identify their security code as “1”. The pre-processor does a one-time pass through the initial

tags, and recognizes all of the attributes that are marked as secure, adding them to a map for lookup. After the file is completely loaded, it runs through the data block, modifying necessary fields.

Our preprocessor, however, implements a version of the `Datafly` algorithm[1], and thus anonymizes what is called the *quasi-identifiers* - a set of attributes which can be used, when combined, to make a near-unique key, allowing anyone to map these quasi-identifier sets to any particular tuple of data. Therefore, rather than making sure the *sensitive* data is immediately obfuscated through manipulation, we obfuscate these quasi-identifier sets in an attempt to achieve k-anonymity. That is, for any particular set of quasi-identifiers, there exists k possible tuples of data that match.

When considering effective methods on anonymizing data for maintaining privacy, it's important that the overall level of noise is kept to a minimum, as to retain the integrity and validity of the model created by the data. That said, methods on actually doing this are still a hot topic of data mining, and coming up with a good one is less than trivial. As an initial approach, we decided to perform very basic mathematical operations on the data that must be secure - adding small offsets, applying linear functions, and so on. In every one of these cases, the results were minimal, as expected. Small mathematical operations introduced a lot of noise, reducing the accuracy determined by Weka's default cross-validator by a moderate amount, and the modification is easily identifiable to anyone intent on reversing the trends of the data.

---

[1] Datafly is one of the earlier k-anonymity algorithms. More information on the Datafly algorithm can be found in Dr. Sweeny's paper on PPDM algorithms at the following URL: http://portal.acm.org/citation.cfm?id=774553

Method of Anonymization (Q. 3)

The method used to anonymize these quasi-identifier sets is through the `Datafly` algorithm. Essentially, the `Datafly` algorithm continuously generalizes quasi-identifiers via a Domain Generalization Hierarchy[2] on any particular attribute, noted `DGH`(Attribute). Pseudocode for the algorithm is as follows:

```
FREQ <-- list of quasi-id value frequencies from table
QUASI <-- the set of quasi-identifiers in FREQ with count < k
While QUASI accounts for > k records:
      Choose attribute A with greatest number of distinct values
      Generalize attribute A according to the DGH(A).
      Re-calculate QUASI and FREQ
Remove records with quasi-id set Q, where Q refers to < k records
Return resulting table
```

The `DGH` function, in our case, generalizes the attribute via an attempt to create similar-frequency buckets, and higher levels (more general levels) of the hierarchy account for a smaller bucket count. By doing this, values become increasingly more general over every passing iteration, and therefore, a particular set of quasi-identifiers account for increasingly more instances in the data.

Our implementation of this algorithm therefore used a few maps, and took advantage of Weka's objects for *`Instances`*, *`Instance`*, *`Attribute`*, and *`Filter`*. It was necessary that we kept a constantly-updated map that kept track of the number of unique instances for any particular quasi-identifier, as well as a frequency map for any

[2] Domain Generalization Hierarchy defines a process in which a particular attribute may become more generalized. Ex: For a date (11/2/1989), the levels of generalization are "11/2/1989" →"11/1989" → "1989" →"1980" ..etc

particular set of quasi-identifiers, which we just implement as a concatenated string. Binning is done by copying the particular attribute in question on our general instance of Weka's *Instances* object. After we create our copy, we create a new *Discretize* filter, passing it over the copy. Any time we need to re-bin an attribute on a subsequent iteration, we remove the old copy, create a new one, and re-bin.

These increasing generalizations are the primary tool used to create k-anonymity while preserving the data and adding minimal noise. With `Datafly`, it's possible to set up the `DGH` function such that generalizations are very small. This will increase the number of iterations overall, however, but will also introduce minimal noise per iteration on the new dataset. Furthermore, selecting an initial k that makes sense for the given dataset is crucial, and is dependent on the domain. Is the data secure if a set of public data can lead to 10 possibilities, or 100 possible instances?

We found it important to, taking into consideration these factors, set up specific constraints on data using our algorithm. Our `DGH` function currently only works for attributes that are *numeric*. Furthermore, it's important that K is chosen in a manner entirely dependent on the data. For this application, we speculated that K = 10 to be a sufficient balance between anonymity and data preservation (the larger the K, the more binning occurs). Any further data will specify the K value used. Lastly, it is important that there exists a moderate to large set of data (> 1000 instances), as our initial bin count starts as the square root of the number of instances. Below are a list of final requirements for any `.arff` file that goes through our preprocessor:

- `SecurityCode` must be the LAST attribute
- The classifier must be the second to last attribute, before `SecurityCode`
- All attributes marked with `@secure` are quasi-identifiers
- All attributes marked with `@secure` must be NUMERIC
- For best results, the number of instances should be larger than 1000

While not a direct requirement for our program, it is also important to note that all instances with a `SecurityCode` of 2 are pruned *prior* to going through the `Datafly` algorithm. The reasoning for this is that, should a set of instances on a quasi-identifier key just barely meet k-anonymity, it would be detrimental to preserving the k-anonymity of non-private instances in the same set if we removed these private instances afterwards.

Creating the Training and Validation Sets (Q. 9)

Before running the algorithm on our dataset and testing the integrity of the data, we created a simple program to generate a training dataset and a validation dataset to use in Weka's prediction analysis with Naïve Bayes. Because there are so few instances of classifications 1, 5, and 6, it is important that our training data, and validation data both had occurrences of instances that fall into those classifications, and were still representative of the overall dataset. We decided to take a 70/30 approach in dividing the data (similar to standards of 75/25 splits), where 70% of all instances would be used as training data, and 30% of all instances would be used as validation checking.

The program loads all of the data as Weka instances, randomizes the ordering, and then chooses 70% of the instances to be go into a training dataset, and the

remainder into a validation set. This was performed separately on instances for each particular classification. By doing this, not only was the selection randomized, but also proportionally representative of the actual dataset, while assuring our training data to have instances of all classifications.

Beyond using this 70/30 split of the data, we also made frequent usage of Weka's default self-validation analysis on each model. Specifically, training datasets had their validity checked using Weka's default self-validation routine, and validation set accuracy is based on prediction analysis using the training set. Lastly, to note, we only anonymized 2 quasi-identifiers, which will be explained in the analysis. All models are generated by using Naïve Bayes. See the tables below for the compiled results.

Compiled Data

Model	$Class_{correct}$	$Class_{incorrect}$	Accuracy %	Suppressed*
Training (No Preprocesing)	3595	461	88.6341	0
Validation (No Preprocessing)	1519	214	87.6515	0
Training (k=5)	3660	392	90.3258	4
Validation (k=5)	1534	199	88.517	4
Training (k=20)	3617	439	89.1765	0
Validation (k=20)	1555	178	89.7288	0
Training (k=50)	3591	431	89.2839	48
Validation (k=50)	1527	192	88.8307	48

Figure 1. Overall accuracy of models

*Records were suppressed during the preprocessing stage

Training Data (No Preprocessing)

a	b	c	d	e	f	Correctly Guessed
16	0	0	0	0	0	1
0	272	3	1	0	0	0.986
0	86	2265	223	0	30	0.87
0	0	57	1018	46	9	0.901
0	0	0	5	22	0	0.815
0	0	1	0	0	2	0.667

Figure 2-a. Confusion Matrix of Training Data (No Preprocessing)

Validation Data (No Preprocessing)

a	b	c	d	e	f	Correctly Guessed
6	0	0	0	0	0	1
0	117	0	0	0	0	1
0	46	958	98	0	13	0.859
0	0	22	430	26	5	0.89
0	0	0	3	8	0	0.727
0	0	1	0	0	0	0

Figure 2-b. Confusion Matrix of Validation Data (No Preprocessing)

Training (k=5)

a	b	c	d	e	f	Correctly Guessed
16	0	0	0	0	0	1
0	273	2	1	0	0	0.989
0	84	2302	208	0	8	0.885
0	0	56	1044	25	3	0.926
0	0	0	5	22	0	0.815
0	0	0	0	0	3	1

Figure 3-a. Confusion Matrix of Training Data (k=5)

Validation (k=5)

a	b	c	d	e	f	Correctly Guessed
6	0	0	0	0	0	1
0	115	1	1	0	0	0.983
0	39	972	101	0	3	0.872
0	0	27	432	21	3	0.894
0	0	0	2	9	0	0.818
0	0	1	0	0	0	0

Figure 3-b. Confusion Matrix of Validation Data (k=5)

Training (k=20)

a	b	c	d	e	f	Correctly Guessed
16	0	0	0	0	0	1
0	274	2	0	0	0	0.993
0	83	2236	274	0	11	0.859
0	0	37	1067	24	2	0.944
0	0	0	6	21	0	0.779
0	0	0	0	0	3	1

Figure 4-a. Confusion Matrix of Training Data (k=20)

Validation (k=20)

a	b	c	d	e	f	Correctly Guessed
6	0	0	0	0	0	1
0	114	3	0	0	0	0.974
0	36	967	105	0	7	0.867
0	0	15	460	8	0	0.952
0	0	0	3	8	0	0.727
0	0	0	1	0	0	0

Figure 4-b. Confusion Matrix of Training Data (k=20)

Training (k=50)

a	b	c	d	e	f	Correctly Guessed
16	0	0	0	0	0	1
0	268	2	1	0	0	0.989
0	88	2236	251	0	9	0.865
0	0	47	1048	24	2	0.935
0	0	0	6	21	0	0.778
0	0	1	0	0	2	0.668

Figure 5-a. Confusion Matrix of Training Data (k=50)

Validation (k=50)

a	b	c	d	e	f	Correctly Guessed
6	0	0	0	0	0	1
0	113	2	0	0	0	0.983
0	45	944	115	0	2	0.854
0	0	11	458	8	3	0.954
0	0	0	5	6	0	0.545
0	0	1	0	0	0	0

Figure 5-b. Confusion Matrix of Training Data (k=50)

Integrity of Data (Q. 4, Q. 5, Q. 6)

Note that in the above data, classifications a through f reflect Donor Codes 1 through 6, respectively.

As you can see from *Figure 1*, increasing the value of k does not drastically affect the overall accuracy of the model. In fact, the total fluctuation of accuracy between all models is never greater than 3%. We speculate, however, that as more attributes are chosen to be anonymized with bigger k values, the overall accuracy of the model will decrease. Further testing should be done on datasets with fewer attributes, and more specific `DGH` functions on quasi-identifiers.

Looking closely at the confusion matrices, specifically those on the training data, Weka's self-validation shows that the overall accuracy on classifications remains relatively the same, with the biggest decrease being in classification errors on Donor Code 5, where the error rate increased by just over 3%. Once again, we expect that for large values of k, and a greater number of quasi-identifiers being obfuscated, the error percentage will continue to increase.

As for the validation datasets, error rates were more pronounced. Once again, there are minor increases or decreases of classification accuracy exist as k increases, but comparing *Figure 2-b* with *Figure 5-b*, the increase of error rates are expected to be the highest. We were surprised, however, to find that error rates increase only on the more "sparse" classifications, particularly with Donor Code 5, where there was nearly a 20% decrease in prediction accuracy based off the training set. Overall, however, these predictions do not vary greatly with the clean, unmodified data predictions.

Looking into specific instances of data, both before and after our Datafly modifications where k=50 (expected to have the highest error rate), we did notice that some instances that were previously correctly classified, were misclassified after the data changed. For example, instance number 26, classified as Donor Code 2 in the original data, was misclassified as 3 in the modified dataset. This occurred numerous times, especially for instances that were originally correctly classified as Donor Code 5, and then misclassified in the modified dataset. It should also be noted that all data in the `Datafly` algorithm have been perturbed, as every instance's quasi-identifiers get binned. However, the majority of our misclassifications occurred on different instances between the two datasets. Because of this, examination on the confusion matrices gives the best overall picture on the error rate for any particular classification.

Other Trends (Q6)

There are still a few trends in the data that have not been identified previously. Firstly, it is important that we look for indications of success with our `Datafly` algorithm with regards to the overall classification of instances. When looking at *Figure 5-b,* the confusion matrix for the k=50 validation dataset, it's easy to notice that a few instances have been pushed into classification d (Donor Code 4), a sign of instances on these two classifications joining to meet the k=50 anonymity.

Weaknesses of Algorithm (Q. 6)

After taking into consideration this data, and these trends, we found many weaknesses and limitations to the `Datafly` algorithm. The most noticeable problem with `Datafly` is also its strength – the `DGH` function. `DGH` functions are best created by implementing customized hierarchies dependent on the data. For example, creating hierarchies to generalize address information is simple – Remove specific residence information, then remove town information, then state information, and so forth. However, not all generalizations are best done, or are possible by the usage of generic binning. In the case of donation data used in this project, this is especially seen through the inclusion of high-end donors whose data may be sensitive. Our generic binning will eventually pool them into the same category as other, lower end donators, but not in a way that preserves all high-end donors together.

A second weakness in the algorithm is seen when a dataset has many quasi-identifiers, and each quasi-identifier has many bins. The number of quasi-identifier sets is equal to $\text{numBins}(A_1) * \text{numBins}(A_2) * \ldots * \text{numBins}(A_n)$. In the case of the donation dataset, first iteration of `Datafly` produced approximately 80^6, or over 2.6×10^{11} quasi-identifier sets. The ultimate result is that when attempting to generalize too many quasi-identifiers with too many buckets (a consequence of a generic `DGH` function), the algorithm will ultimate put all instances in an extremely low quantity of buckets in an attempt to reach k-anonymity, thus rendering the modified data completely unrepresentative of the original dataset. In our case, marking all six public donation attributes as secure resulted in the final data being categorized in 1-2 bins.

Another weakness lies within the selection of k. The `Datafly` algorithm will continue to run until there are < k instances that do not meet k anonymity, and then will suppress those records entirely. However, in situations where k is relatively small, and the number of distant outliers is large, the algorithm will continue to generalize quasi-identifiers, even if the non-outlier data is sufficiently anonymized. To prevent this, we discussed multiple measures.

One measure we propose is the addition of a ”flexibility constant”. This constant would thus allow a greater number of outlier instances to be ignored, and later suppressed. This constant can be added to K in the algorithm’s loop condition, which will not affect the number of instances that must be accounted for by any particular quasi-identifier set, thus preserving k-anonymity for the number of instances minus some value less than K + Constant. Another, more preemptive measure, is ensuring that the data is clean of all outliers, the definition of which being dependent on the data at hand.

Overall, we believe that these weaknesses should be taken into account when selecting an appropriate k value, and when marking attributes as quasi identifiers. It is extremely important to note, then, that for full k-anonymity on the entirety of the data, `Datafly` relies on anonymizing *all* attributes that make up strong quasi-identifier sets. Therefore, we can only conclude that we’ve **obfuscated relationships between any attributes marked as quasi-identifiers to any particular instance of data.**

Overhead of Preprocessor (Q. 8)

The current implementation of the preprocessor reads in an .arff file, and then loads all the relevant data into Weka instances. Filters are then created and performed over these instances in a linear fashion The consequences of this is that there are many linear passes over the data, as well as many expensive Weka objects for each instance, leaving a relatively high processing and memory overhead. The `Datafly` algorithm is also an $O(n^2)$ function with a high constant (some number of iterations, each going through N instances). Furthermore, Weka's internal classes don't seem to be multithreaded. Overall, this implementation would be slow with extremely large datasets. For our particular dataset, however, the runtime of the algorithm only lasted a few seconds, dependent on the value of K and the number of attributes tagged as secure.

Extension to Other Datasets (Q. 7)

The `Datafly` algorithm can be generalized to other datasets, so long as there is a defined `DGH` function for all of the quasi-identifiers. Our implementation can also be applied, so long as all quasi-identifiers are nominal. The difficulty in creating an implementation of `Datafly` is that it's near impossible to create a generic `DGH` function that works well for any given dataset. Much like how Weka's attempts at binning are not always perfect for the data is receives, the `DGH` function is best created with the specific data in mind. It may be possible, however, to create an implementation of `Datafly` that requires an overridden `DGH` function for each quasi-attribute, so that every actual implementation can quickly define a `DGH` function and use the code.

README for Anonymize.java and TrainingGenerator.java

AUTHORS:
Nidhin Pattamiyil
Eitan Romanoff

Anonymize.java

Anonymize.java is our preprocessor.

Anonymize <infile> <outfile>

Infile – the original .arff file

Outfile – the output .arff file

K (used in K-anonymity) is a constant inside the code.

TrainingGenerator.java

TrainingGenerator.java is our simple program that made our training data and validation data.

TrainingGenerator <infile> <training> <validation> <split>

Infile – the original .arff file

Training – the destination for the training .arff file

Validation – the destination for the validation .arff file

Split – percentage of data to put used in training