COMP523

One-Class Learning & Concept Summarization for Vaguely Labeled Data Streams

Xavier Simmons & Tyler Hale

Paper Outline

Section 1. Introduction

Section 2. Define Problem and Discuss Simple Solutions

Section 3. Discuss Proposed One-Class Learning & Concept Summarization Framework

Section 4. Discuss VOCL Module

Section 5. Proposed Solution for Concept Summarization (OCCS Module)

Section 6. Analyze System Time Complexity

Section 7. Results

Section 8. Review Related Work

The Problem

There are two problems the paper is trying to address.

1. Vague One-Class Learning Problem
2. One-Class Concept Summarization Problem

What is the Vague One-Class Learning Problem?

One-class classification tries to determine whether a new test datum is a member of a specific class or not, instead of traditionally trying to determine which out of n-classes it best fits. In Vague one-class learning, a labeled sample set has mixes of instances which may or may not be of interest to users. Effective mechanisms must be developed to differentiate positive samples in order to identify users' genuine interests

What is the One-Class Concept Summarization Problem?
When testing data over a stream of the user's interests, the samples may contain multiple concepts due to the user's changing interestings such as rainfall, pressure, wind velocity etc. Therefore we need a method to *best* summarize the user's interests such that we can predict what else might interest the user.

The Test Data

They tested the performance of the VLCS system on four datasets but we will expand on two of the scenarios to clearer define the problem we are trying to solve:

1. Sensor Test Data Scenario
2. Power Test Data Scenario

The first used the sensor data stream provided by Intel Berkeley Research Lab. The data stream contains information about **temperature**, **humidity**, **light**, and sensor **voltage** readings gathered from 54 sensors inside the lab, as shown in Figure 1. The goal is to correctly identify which region a particular reading is coming from.

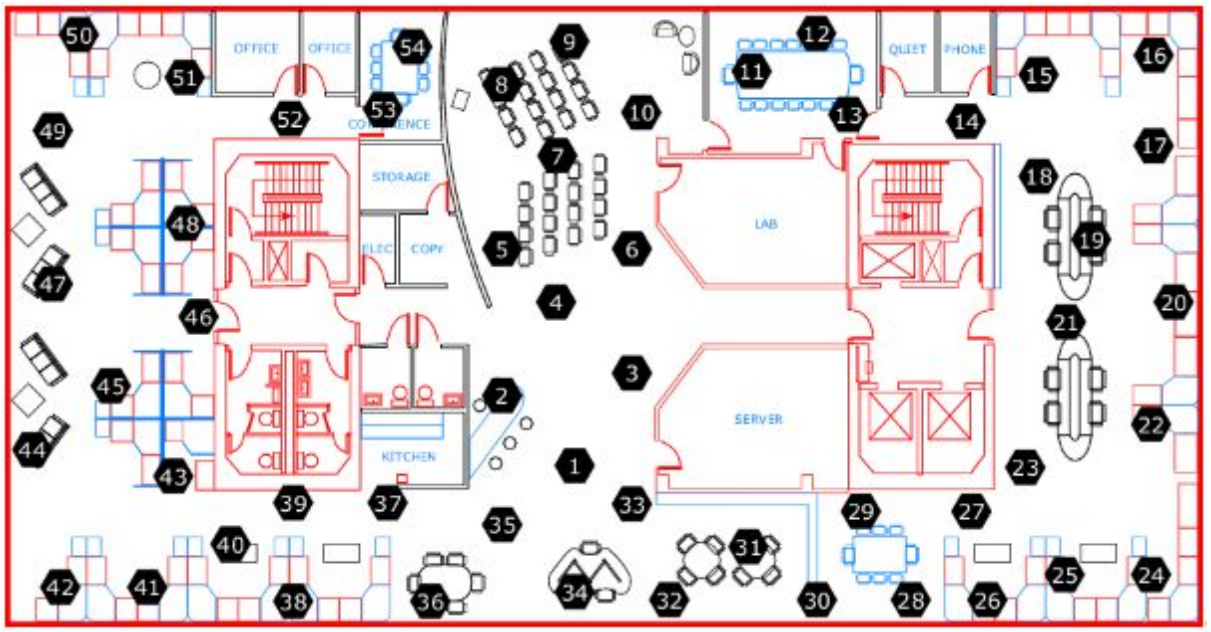


Figure 1: the sensor distribution map deployed in Intel Berkeley Research Lab

The second experiment uses the power supply data stream from an Italy electricity company, which contains power supply records each hour from 1995 to 1998 from two sources: **power supply from the main grid**, and the **power transforms from other grids**. The goal is to predict which hour the current power supply entry belongs to. However in the paper they grouped the records into 2-hour groups ([0,1], [2,3], ..., [23,24]) so the task was actually to predict which out of the 12 time periods the power supply reading belongs to.

The Algorithm (VLCS)

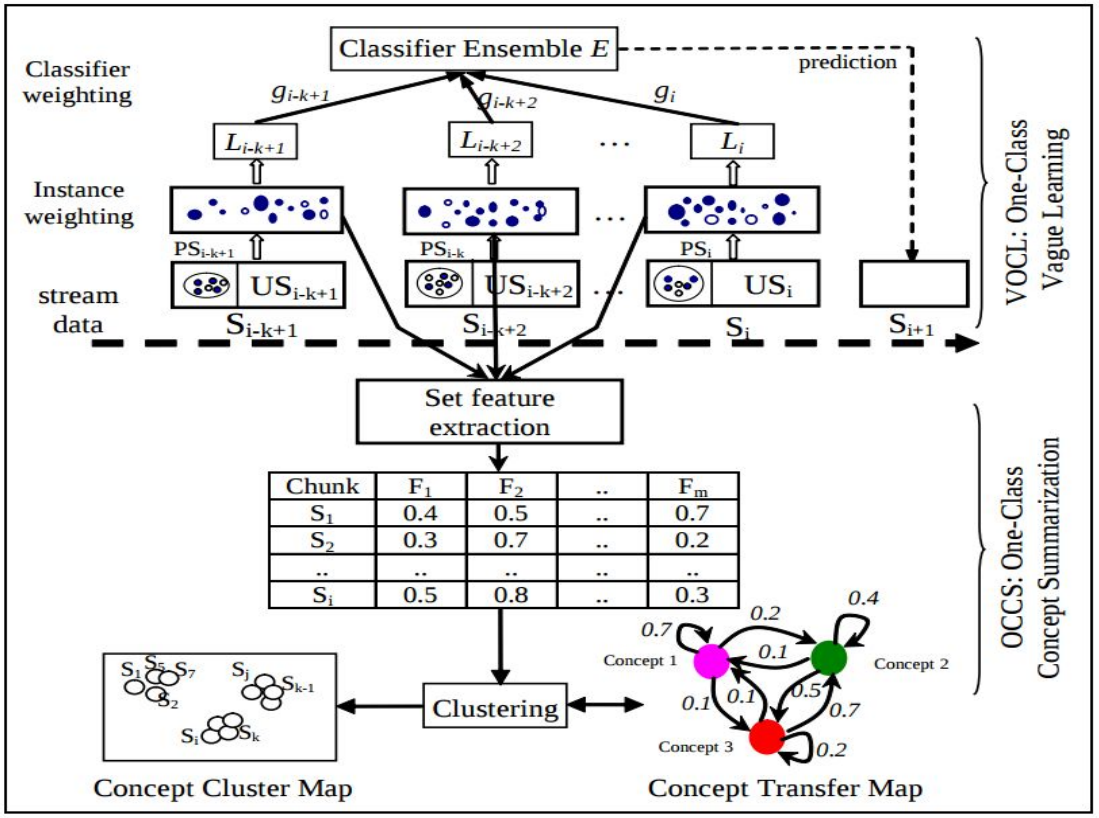


Figure 2: VLCS: Vague one-class learning and concept summarization framework

This diagram can be quite daunting at first however let's break it down into modular components that we can understand.

The VLCS (Vague one-class Learning and Concept Summarization) framework can be decomposed into two major modules each to solve the two challenges be covered above.

1. VOCL: One-Class Vague Learning module
2. OCCS: One-Class Concept Summarization module

The **VOCL** module can be further broken down into 3 main layers:

1. **Partition the data**
    - This part of the algorithm takes in a batch of instances called a chunk (Si) and partitions it into two subsets (positively labeled and unlabeled subsets). This is can be defined by clustering the data via its attributes and effectively sorting by smallest size and labeling the first N instances from the set as positive. This is user emulation of them selecting

clusters/regions of interest. This is known as *vague-labeling as* the positively labeled samples may not be of the user's genuine interest and therefore is just a prediction.

2. **Instance Weighting**
    - There are 3 weighting stages; Local Weighting, Global Weighting, and Unified Weighting.
    - With **Local weighting**, each instance in the chunk is individually classified using cross-validation on the chunk and assign weights of:
        - 1.0 if classified positive and was vaguely labeled positive
        - 0.5 not classified positive but was vaguely labeled positive
        - 0.5 - 1.0 if unlabeled and classified as positive.
    - **Global Weighting** is used to generate predictions for both positive and unlabeled instances. This is done by using the ensemble of the previous k classifiers and assigns a weight/percentage to the number of classifiers predicting that the given instance is positive (part of the class)
    - Finally **Unified Weighting** is interesting because it combines the local and global weights to generate a weight value that favours high local and global weights. That is to say it also favours local and global weights that are near exact weights. The highest weight is given when local weight = 1.0, and global weight = 1.0 where the unified weight will be 5. The unified weight will be 1.0.
3. **Classifier Weighting**
    - Finally we assign weights to each classifier in the ensemble to adjust to the users' current interests. Note that each classifier is trained from samples that are independently sampled from the unified weight distribution using Rejection Sampling which leads to more accurate results.

The second half of the algorithm is the **OCCS** module. This can also be decomposed into 3 main layers however our implementation focused on replicating the VOCL module:

4. **Concept Cluster Map Creation**

The concept cluster map directly employs the cluster structure to visually demonstrate concept-chunk relationships in the stream, starts with extracting features from each chunk into a virtual set and then generates clusters from it. The goal is to get to an output state of that displayed in the paper:

Chunk S_i

Instance ID	v_1	v_2	Instance weight
1	2.5	T	3.0
2	5	T	1.0
3	3	F	4.0
4	7	F	2.0
5	6.2	T	0.0

Virtual Instance	Γ_1^1 [0,2]	Γ_1^2 (2,4]	Γ_1^3 (4,6]	Γ_1^4 (6,8]	Γ_1^5 (8,10]	Γ_2^1 T	Γ_2^2 F
Γ_{S_i}	0	0.7	0.1	0.2	0	0.4	0.6

Figure 8: A toy data chunk S_x with five instances, each of which has a numerical feature v_1, a categorical feature v_2, and a weight value. The virtual instance and the set features extracted from the toy set are shown at bottom of the picture.

5. Concept Transfer Map Creation

Initial reasoning for using this method over the Concept Cluster Map was that it the CCM had issues in relation to temporal correlation being discarded in the labeling process. The usage of a Markov Model allows for a conditional probability to be transferred.

A Markov model to capture concept transfer patterns during the user's labeling process with the probability for users to choose a particular concept given that that the prior occurred, the conditional probability transferring between two consecutive chunks Si-1 and Si:

P(Ci=cxi|Ci-1=cxi-1)

Shown visually:

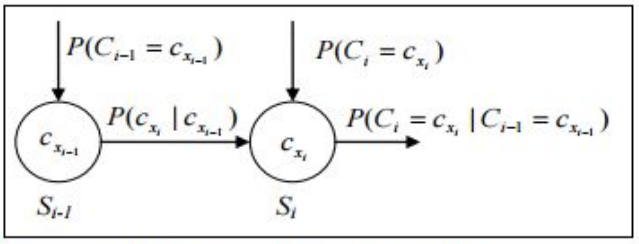


Figure 9: The graphical model of the conditional probability transferring between two consecutive chunks S_{i-1} and S_i.

6. Feature Extraction

To extract feature for each chunk, we propose to transform original feature values in each chunk into some histogram format, such that each chunk Si can be represented by using histogram based features

Maven and Packaging

When we initially started the project, we setup our implementation with library references to the lib/ folder filled with required jars for Moa, Weka, LibSVM and a deprecated OneClassClassifier jar from an old Weka version that we needed. We found that there was a maven repository for them so we updated the project by adding a pom.xml that defined our dependency configuration and restructured to link to them.

Our Repository Dependencies:

- 1.1.3 https://mvnrepository.com/artifact/nz.ac.waikato.cms.weka/phmm4weka
- 1.0.4 https://mvnrepository.com/artifact/nz.ac.waikato.cms.weka/oneClassClassifier
- 1.0.6 https://mvnrepository.com/artifact/nz.ac.waikato.cms.weka/LibSVM
- 2014.04 https://mvnrepository.com/artifact/nz.ac.waikato.cms.moa/weka-package
- 2014.04 https://mvnrepository.com/artifact/nz.ac.waikato.cms.moa/moa

After we setup the Maven packages it was easier to add libraries and work out any issues with dependencies. One of the issues we encountered was the instances not being compatible between Moa and Weka for certain classifiers, specifically for those that used `Samoa.instances` instead of `weka.core.Instance`. This was using the Weka 2016.04 and Moa 2016.04 versions from the Maven repository found here. To resolve this we rolled back to the 2014 version for both Weka and Moa which leader to an interoperability improvement between Moa and Weka.

Our VOCL Implementation

Since there were a lot of layers to the algorithm we decided to focus primarily on the Vague One-Class Learning module. The structure of our implementation is as follows:

VOCL **Package**:

- VageLabeling **Interface**
  - ClusterVagueLabeling **class**
  - RandomVagueLabeling **class** (Optional TODO)
- LocalWeighting **class**
- GlobalWeighting **class**
- MOAOneClassClassifier **class**
- OneClassClassifierEnsemble **class**

- RejectionSampling **class** (Needed TODO)
- VOCL **class**

VOCLRunner **class**

The VOCL class is the main pseudo code implementation lies (Figure 5.) It also contains the unified weighting and classifier weighting functions.

We used the OneClassClassifier class provided by a deprecated version of Weka as it contained exactly what we needed. However since the assignment was to use Moa we wrote a MOAOneClassClassifier wrapper that ports enables it to interop with other Moa components (more information about this can be found in the **Issues** section).

Since all the ensembles in Moa update their own weights and use their own default classifier types we implemented our own OneClassClassifierEnsemble class that contains extracted and ported functionality from the WeightedMajorityAlgorithm class along with other implementation details that can be read about in the **Issues** section.

Issues

The main issues we kept running into was Weka to Moa interoperability conflicts and result ambiguity. This paper used Weka for the implementation so the goal was to cross-match the concepts in the paper with the “best fitting” data structures in Weka, then then cross-match those data structures with somewhat “equivalent” data structures in Moa.

One of the scenarios we had was where we wanted to use the `OneClassClassifier` in the deprecated Weka package inside the Moa `WeightedMajorityAlgorithm` classifier. The final step of the VOCL module required a weighted voting mechanism so we had the idea of manipulating the class to use my Moa Wrapper/Ported one class classifier as shown below:

```
public class MOAOneClassClassifier extends WEKAClassifier implements Classifier {

    public MOAOneClassClassifier() { this.classifier = new OneClassClassifier(); }
```

When looking at the Moa implementation of those ensembles we realized the contained a `learnerListOption` that instantiated the classifiers inside the option list. By default this is set to the Hoeffding Trees. Therefore we tried writing my own ensemble class that inherited from the `WeightedMajorityAlgorithm` and overwrote the option list to use my `MoaOneClassClassifier’s.` This built and

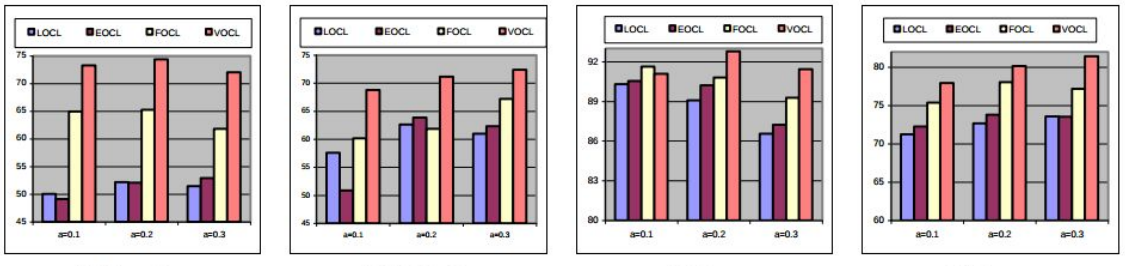
we were able to override all the weights as we wanted, however there was a lot of ambiguity on what was going on since I was overriding a lot of internal state and we weren't obtaining the results we expected.

Therefore we moved to an alternative solution which was implementing my own `OneClassClassifierEnsemble` class that implemented the Moa Classifier interface so It would interop with Moa. We then extracted the weighted voting logic from the `WeightedMajorityAlgorithm` and overloaded the getVotesOnInstance function to extract the classification distribution from the MoaOneClassClassifier. This successfully generated the weights we expected! Along with the ported functionality, we also added the sliding window classifier ensemble functionality the paper describes and a simple classifier weight update function. This ended up being an elegant solution and worked nicely in Moa.

In summary, one thing we realized is that Moa seems to have much better ensemble support compared to Weka, specifically with weighting ensembles. However there are some useful Classifiers such as the OneClassClassifier that would be really good to have in Moa. We believe there needs to be some proper development into merging these 2 APIs into one cohesive Machine Learning & Data Stream Learning API.

Results

Expected



(a) HyperP (b) KDD-99 (c) Power (d) Sensor

Figure 12: Average prediction accuracies with respect to different percentages of labeled samples (α) in each chunk (the x-axis denotes α values, and β is fixed to 0.5 for all streams, using probability shifting user interest model). For each α value, the bars from left to right correspond to LOCL, EOCL, FOCL, and VOCL.

Ours

For testing our implementation we ran our algorithm on the powersupply dataset you can find here. We used their recommended chunk size of 1000, KMeans size of 20, number of folds set as 10, and the percentage of positive labeled instances as 0.3. We supplied our algorithm the required class index (hour) and the attribute (power supply from the main grid, and the power transforms from other grids). We experimented with all permutations however we could not obtain high accuracies (< 20%).

Due to our issues with Moa and Weka we unfortunately could not replicate the papers results; i.e replicate the predicted accuracy on each new chunk. Because a lot of our efforts were focussed on resolving our issues we have not implemented the Rejection Sampling pseudo code so instead we apply the unified weight to each instance and built the most recent classifier from that. We assume this has a strong negative impact on our results as they are still still far from ~90% predicted accuracy so there is some further investigation that is required. On the other hand we can discuss our three weighting results we obtained from Local, Global and Unified Weighting.

Local Weighting Results

```
Is pos: 0 Was_pos_pred: 0 WLx[p]: 0.0
Is pos: 0 Was_pos_pred: 0 WLx[p]: 0.0
Is pos: 0 Was_pos_pred: 0 WLx[p]: 0.0
Is pos: 1 Was_pos_pred: 0 WLx[p]: 0.5
Is pos: 1 Was_pos_pred: 0 WLx[p]: 0.5
Is pos: 0 Was_pos_pred: 1 WLx[p]: 0.75
Is pos: 0 Was_pos_pred: 1 WLx[p]: 0.75
Is pos: 1 Was_pos_pred: 1 WLx[p]: 1.0
Is pos: 0 Was_pos_pred: 0 WLx[p]: 0.0
Is pos: 1 Was_pos_pred: 0 WLx[p]: 0.5
Is pos: 0 Was_pos_pred: 1 WLx[p]: 0.75
Is pos: 0 Was_pos_pred: 1 WLx[p]: 0.875
Is pos: 0 Was_pos_pred: 1 WLx[p]: 0.75
```

As you can see high weights were assigned to positively labeled samples that were also predicted as being positive, followed by unlabeled samples that were predicted as positively labeled and finally the lowest weights were assigned to vaguely labeled samples that were not classified as positive.

Global Weighting Results

```
% Predict_positive: 0.0
% Predict_positive: 0.0
% Predict_positive: 0.11111111
% Predict_positive: 0.44444445
% Predict_positive: 0.44444445
% Predict_positive: 0.5555556
% Predict_positive: 0.5555556
% Predict_positive: 0.5555556
```

This is quite simple, as it is calculating the number of K-1 classifiers in the ensemble that classified the unlabeled or positively labeled samples as being positively labeled (part on the class)

Unified Weighting Results

```
LocalW: 0.5 GlobalW: 0.0 UnifiedW: 1.0
LocalW: 0.0 GlobalW: 0.33333334 UnifiedW: 1.0
LocalW: 0.75 GlobalW: 0.5555556 UnifiedW: 2.6000001
LocalW: 0.75 GlobalW: 0.7777778 UnifiedW: 3.842105
LocalW: 0.75 GlobalW: 0.7777778 UnifiedW: 3.842105
LocalW: 1.0 GlobalW: 0.7777778 UnifiedW: 3.153846
```

These are the most interesting results. If we have a look at the highlighted example and the bottom example weightings, we can successfully observe the behaviour we expect. That is to say that the unified weighting scheme favours both high local and global weightings, but also, similar local and global weightings evaluate to a higher weight value than one of the weighting schemes being higher but the difference is greater. I.e

> Highlighted example: LW + GW < Bottom example: LW + GW but
>
> Highlighted example: UW > Bottom example: UW

Summary

To reflect, there were a lot of layers to the algorithm making it a bigger bite than we could chew. However we adapted fast by narrowing the scope down to just the VOCL

module. There was vagueness on how the pieces fit together exactly on the original paper and some of the mathematical notation did not quite match up with the written explanation. However our main problems resided interoperability between Weka and Moa data structures. Some of the problems arose when trying to cross-match ideas with data structures from the paper to Weka to Moa. We had some thoughts for further development and improvements to the algorithm for the future. Specifically we wanted to look at alternative clustering methods such as spectral clustering which they mentioned (but didn't evaluation) along with DBSCAN or HDBSCAN. Finally we would like to have a cleaner implementation using our own or with purely just moa.