A New, General Purpose, Semi-Supervised Machine Learning Algorithm Attempted to be Implemented at Scale in Python

---

A THESIS

Presented to

The Faculty of the Department of Mathematics and Computer Science

The Colorado College

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Arts

By

Robert Malone, Dexuan Zhang, and Jacob Nehama

A New, General Purpose, Semi-Supervised Machine Learning Algorithm Attempted to be Implemented at Scale in Python

Robert Malone, Dexuan Zhang, and Jacob Nehama

April 2019

Computer Science

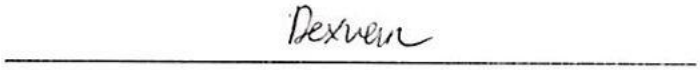
Abstract

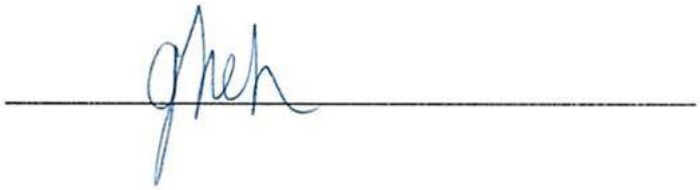
In 2005, Vikas Sindwani and the University of Chicago team proposed a new semi-supervised kernel-based SVM algorithm in the paper *Beyond the Point Cloud: from Transductive to Semi-supervised Learning*. In this algorithm, instead of Euclidean geometry, they utilized the unlabeled data to understand the underlying geometry and construct a Deformed Kernel to encode the information of the deformed geometry. However, the computational time complexity of construction of the Deformed Kernel is cubic. Consequently, the algorithm failed to scale. Our goal is to improve the algorithm and eventually construct a deformed kernel within quadratic computation time complexity.

Acknowledgements

We would like to thank Professor Richard Wellman for all the help and allowing us to work on this project with him. We would also like to thank the entire Mathematics and Computer Science Department for all the work they have put in to help us become better computer scientists.

ON OUR HONOR, WE HAVE NEITHER GIVEN NOR RECEIVED UNAUTHORIZED AID ON THIS THESIS





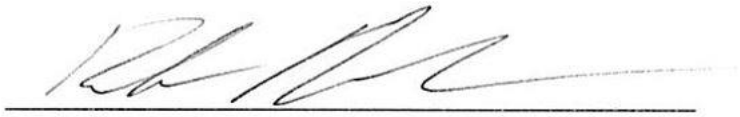


TABLE OF CONTENTS

ABSTRACT i
ACKNOWLEDGEMENTS ii
HONOR CODE iii

1 EXECUTIVE SUMMARY 1
- Goals
- Why this project is important
- Who is this project for

2 REQUIREMENTS 2
- User Stories
- What was completed and what wasn't

3 ARCHITECTURE AND RESEARCH 2
- Belkin Niyogi Sindwani Machine
- Scalability Research
- Semi-Supervised Point Cloud Machine

4 VALIDATION AND DEVELOPMENT PROCESS 6
- Github
- Unit Testing
- Team Communication
- Challenges Faced

5 SUMMARY 7
- Accomplishments
- Failures

6 REFERENCES 10
- Technical Documentation
- Papers Read

7 APPENDICES 10
- Using and testing the code

8 CITATIONS

Executive Summary

This paper will describe our efforts in a computer science research thesis project. We have attempted to implement a new, semi-supervised machine learning algorithm for general purpose at scale in Python. In the world of machine learning, semi-supervised machine learning is becoming extremely popular as the number of unlabeled data is skyrocketing. The kernel-based Support Vector Machine is known as one of the most important models for classifying data; however, it is also known for its fixed cubic time complexity to train on the labeled data. From the SVM model, in order to implement a semi-supervised algorithm that would scale, we want to construct a new kernel with both labeled and unlabeled data to encode the underlying geometry. Since it is impossible to improve the training over the labeled data, our model would take in small amount of labeled data and large amount to unlabeled data. By implementing the training over unlabeled data into quadratic computation time complexity, we will potentially be able to scale up the algorithm and produce significantly better predictions in many fields.

It is important to understand that practically, in the real world, every machine learning problem is a semi-supervised one. The method of which the user trains the model may or may not use unlabeled data, but in the end one must make predictions on unlabeled data and expect those predictions meet some level of accuracy. For example, a toy dataset provided by SKLearn titled, “Breast Cancer”, contains 569 data points. This is great for experimentation and study, but unrealistic in the real world where one might expect millions of samples with only thousands of labeled data points. Being able to use unlabeled data is becoming essential in machine learning. The two figures below represent this important distinction. The black points represent the unlabeled data, and the red and blue points are labeled. If we disregard the unlabeled data, a decision function will predict future data points like Fig. A, but if we are able to recognize the pattern in the unlabeled data, then a more accurate decision function will be created like Fig. B

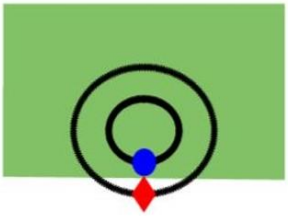


Fig. A

Fig B.

The overall goal of this project is to transcribe Dr. Wellman’s algorithm from Matlab into Python. What we researched was how to do this efficiently. This project has been completed in the short term for Dr. Wellman and in the long term for everyone interested in machine learning. The hope is that one day the popular SKLearn library will accept this algorithm for all computer scientists to use. As stated before, the big issue is keeping the algorithm at scale in Python.

Since the end user of this algorithm will be the users of SKLearn, we needed to follow the guidelines for an SKLearn algorithm. First, there needs to be extensive documentation. There needs to be at least a fit method, predict method, score method, and decision function method. With all this in mind, we will now dig into the research we conducted, the algorithms we implemented, and the success and failures of the project.

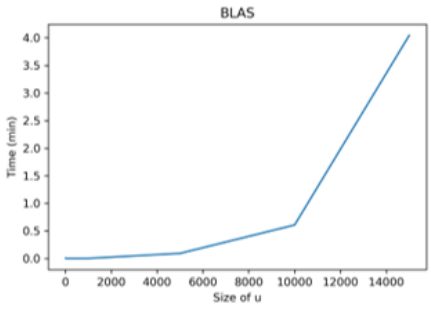
Requirements

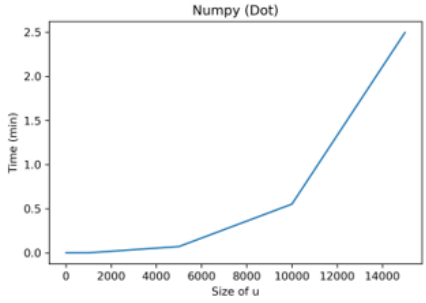
We expected that a user wants to use our model in the field of making predictions. Thus one of our user stories was: John has a data set, and he wants to predict the labels of this data set. This leads to the process of training the model. Hence, John has a data set, and he wants to fit the model to this data set. For the ease of getting a score, John will want to give the model a data set to predict and a set of labels, and evaluate an accuracy score between the predictions and labels. In order to make better judgements on the hyperparameters for the model, John would like to see the decision function on each predicted data point. Finally, John wants to be able to get a list of the parameters and set appropriate parameters after instantiation. This is all the necessities in order for the algorithm to get accepted by SKLearn, and so that is where we started with the implementation.

By the end of the block, we completed two different algorithms that will be discussed later in the paper. We successfully implemented the new, general purpose, machine learning algorithm. But as we investigate the algorithm, it has not been done efficiently enough to work at scale. If we were to continue work on this project, we would implement C modules for specific computations used in the algorithm to step up efficiency. We would investigate new ways to parallelize the algorithm. And finally, we would want to create a retrain function with linear time complexity

Architecture and Research

We began the project with a series of experiments in order to find the optimal way to code this in Python. In our algorithm, there are two computations that occur at a high number. In the Nearest Neighbor model, we calculate a lot of matrix multiplications which is a heavy time complexity computation. Matrix multiplication is a quadratic time complexity function, and we investigated a couple ways of implementing this. First, we analyzed the Basic Linear Algebra System of functions. This is a low level class written in Fortran that can compute matrix multiplication efficiency. We compared the time of this method with Numpy’s dot product. Both are calculating matrix multiplications, but the efficiency for both was different. The two figures below describe our findings.





As one can see above, the Numpy dot product was more efficient than the BLAS implementation. The reason is that BLAS has a series of wrapper functions that hope to reach a Fortran library in the users files. Unfortunately, we cannot assume that the average user has Fortran libraries available on their system, since they are so out of date. So what is happening is that the BLAS method eventually executes the same code that the Numpy dot product does, but with a significant amount of overhead thus causing it to be slightly slower than Numpy. Hence, we decided to implement the Numpy dot product in our code for matrix multiplication.

Another computation that occurs a lot, specifically in the Riemann matrix file, is a binary singleton expansion. Matlab has an extremely efficient implementation of this, and so we wanted to also find the most efficient version in Python. We investigated Numpy's apply along axis function, add function, and broadcast function. We concluded not to use the broadcast function because we didn't want to deal with broadcasting objects. The time complexity for both Numpy's apply along axis function and add function were very similar. Since the apply along axis function provided more flexibility - meaning that we were able to calculate binary singleton expansions with summation and a product - we decided to implement our algorithm with this Numpy function.

As previously stated, scaling the algorithm properly was a large issue that we dealt with time and time again throughout the block. We investigated multiple technologies to help mitigate this problem, including NumPy, Sci-Kit Learn, BLAS (Basic Linear Algebra Subprograms), and Cython, concluding that Cython would be the best framework to improve our efficiency and solve our problem of scaling upwards for larger data sets. Cython is an "optimising static compiler for both the Python programming language and the extended Cython programming language". In simpler terms, while using Cython one can write code that looks like Python grammatically and syntactically, while reaping the performance benefits of the low level language of C. With the power of Cython, we can call C functions, declare C types, and use raw C pointers, producing code that is much more efficient than that of native Python.

```python
def hamming_sum(s0, s1):
    if len(s0) != len(s1):
        raise ValueError()
    return sum(c0 != c1 for (c0, c1) in zip(s0, s1))

def hamming_loop(s0, s1):
    if len(s0) != len(s1):
        raise ValueError()
    count = 0
    for i in range(len(s0)):
        count += (s0[i] != s1[i])
    return count
```

```
**********
**********
python hamming_sum(): 4.45
cython hamming_sum(): 2.55
optimal hamming_sum(): 2.86
----------

----------
python hamming_loop(): 4.16
cython hamming_loop(): 1.55
optimal_hamming_sum(): 1.15
**********
**********
```

```python
def hamming_sum(s0, s1):
    if len(s0) != len(s1):
        raise ValueError()
    return sum(count(cs) for cs in zip(s0, s1))

cdef int count(tuple cs):
    return (cs[0] != cs[1])

def hamming_loop(char *s0, char *s1):
    if len(s0) != len(s1):
        raise ValueError()
    cdef:
        int N = len(s0)
        int i, count = 0
    for i in range(N):
        count += (s0[i] != s1[i])
    return count
```

Continuing to dive deeper into the performance of Cython, we can see a Hamming Sum native Python implementation on the left and a Hamming Sum optimized Cython implementation on the right. By simply compiling the native Python implementation with Cython, we can already see a significant speed up. If we continue to optimize our code with the use of C types, raw C pointers, and C declared functions we can see the significance of Cython speedups. These Cython speedups turned out to be quite misleading and in actuality evolved into something that we couldn't even use within our algorithm due to three key issues, Python objects, late binding, and Global Interpreter Locking. Our algorithm required a lot of intense NumPy calculations, if you're using NumPy you're using Python objects, and if you're using Python objects, you're not using C objects/not able to reap the benefits of the low level efficiencies that come with C. This leads into the second of three issues which is Python's late binding. At run time, "the interpreter does a lot of work searching namespaces, fetching attributes and parsing argument and keyword tuples." In the world of C, searching namespaces, fetching attributes and parsing argument and keyword tuples is done at compile time, allowing the actual "run" of the program to be executed much more quickly since all of this heavy lifting has been taken care. Finally, in conjunction with late-binding, Python implements what is known as the Global Interpreter Lock (GIL). The GIL only allows one thread to hold control of the Python interpreter at a time, meaning that we can only execute one thread of Python code at a time. If the GIL allowed us to take advantage of more than one thread, we could improve the execution time of our algorithm and better solve our scaling problem through parallel programming technologies like OpenMP or Cilk, both available for use with the Cython superset language.

The overall architecture of the algorithm included a driver file and an algorithm file. The driver file steps through each section of the algorithm file. We organized it in this fashion so that it was easier to debug. This structure allowed us to assess which segment of the code was performing improperly. With this architecture, we implemented both the BNSM algorithm and the S3PCM algorithm. The BNSM algorithm is the algorithm provided by Sindhwani and his team at the University of Chicago. It stands for Belkin Niyogi Sindhwani Machine. The S3PCM algorithm consists of our new model, a emi-supervised point cloud machine, which is supposed to have the exact same results as the BNSM algorithm but with better efficiency. We successfully implemented S3PCM to have the exact same output as the BNSM algorithm, but we

will see later than we did not maintain the scalability that Matlab provided in Python. The segments in the BNSM algorithm include the Nearest Neighbor Model and the Graph Laplacian, while the segments in the S3PCM algorithm also include the Nearest Neighbor Model and Graph Laplacian along with the Riemann Matrix and Spectral Basis functions.

NN-model is the first step to construct the Graph Laplacian. To make it as efficient as possible, instead of loops, we used only matrix operations to find the nearest neighbors for each data point. The NN-model module outputs a matrix in the size of approximately number of nearest neighbors by n, size of the data set. The size cannot be accurately stated because we reconstruct the output NN matrix into a symmetric matrix. As mentioned previously, construction of the Graph Laplacian is in $O(n^2)$.

In the original paper, Vikas and the team constructed the Deformed Kernel using the following equation.

$$G(x,y) = K(x)^T(I + LH)^{-1}K(y)$$

Where G stands for the kernel function. K denotes the basis function where the output matrix would be n, the size of the whole data set, by the size of input vector. I is an identity matrix in the size of n by n. L stands for the Graph Laplacian which is an n by n matrix encoding deformed geometry. H is an n by n matrix from basis function with the whole data set as input. As a result, $(I + LH)$ would output an n by n. This is the place leading the scalability issue. Noted that to construct the Deformed Kernel, we would need to invert the matrix $(I + LH)$. Given that the computational time complexity for a matrix inversion is cubic, this construction would take $O(n^3)$. As a result, the algorithm cannot work with large-sized data.

However, it is theoretically possible to improve the construction of the kernel to be in quadratic computational time complexity. The deformed geometry is encoded in the Graph Laplacian and the construction time of the Graph Laplacian is quadratic. We had two routes to improve the algorithm: we could potentially alter the math to either remove the matrix inverting step or replace the matrix inside the matrix inverting step with a smaller matrix to achieve $O(n^2)$

Our solution is to replace the matrix inside the matrix inverting step so we will still be able to achieve $O(n^2)$ given the cubic time complexity for matrix inversion. We construct the kernel using the following equation:

$$G(x,y) = S(x)^T(I + \gamma M)^{-1}S(y)$$

Where S is a variation of Basis Function and size of outputs are in the size of c by the size of input vector. I is an identity matrix in the size of c by c. c denotes the number of control points, the labeled data points and the data points that we're predicting. γ is one of the hyper-parameters. M is a c by c Gelarkin Matrix constructed from the Graph Laplacian using the following steps:

$$R = K^T L K$$
$$M = S(c)^T R S(c)$$

Where R stands for Riemann Matrix and K is an n by c matrix produced from Gaussian function.

For our revised algorithm, we are still inverting the matrix. However, we are inverting a matrix in the size of c by c. As mentioned previously, to scale up a semi-supervised SVM, we intend to reduce the size of labeled data input and to use large amount of unlabeled data. As a result, the cubic of size c is still under the quadratic of size n. Given that the construction of Riemann Matrix is in $O(nc^2)$ and the construction of Spectral Basis, variation of Basis Function, is in $O(c^4)$. To keep the algorithm in $O(n^2)$, our algorithm has a limitation:

$$C \leq n^{1\backslash 2}$$

The semi-supervised point cloud machine has been implemented with quadratic time complexity. Hence, we have completed our goal. But as we will see later, we were not able to maintain scalability. This is not due to the underlying math behind the SP3CM algorithm, but it is due to the limitations of Python, and the fact that Matlab has been built in a way to maximize efficiency for large matrix computations.

Validation and Development Process

Throughout the implementation of the two algorithms, we used Github. This allowed us to share our individual progress. We divided the work up in a way that allowed us to avoid merge conflicts. Stan had the role of implemented the computationally heavy segments of the code such as the nearest neighbor model, graph laplacian, riemann matrix, and spectral basis. Robert was in charge of the overall structure of the algorithm and putting the pieces together so that it could all run. Jacob was responsible for attempting to get the python code into Cython. Github worked great for us because we were able to switch to each others branches and use what we needed.

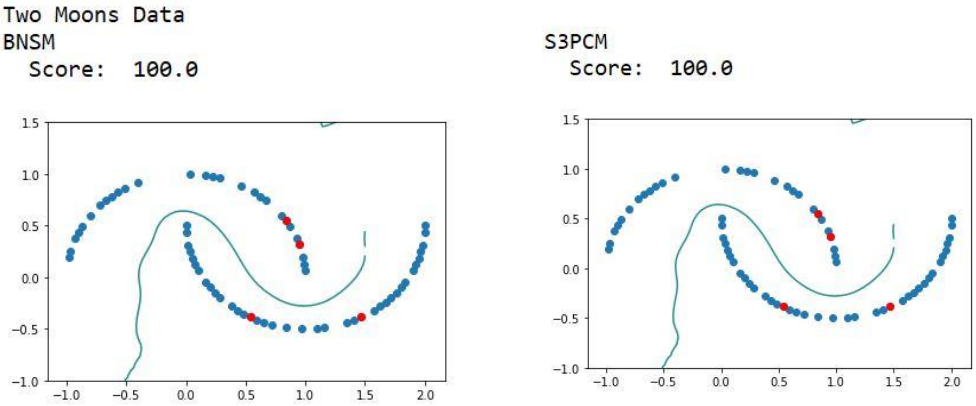
We practiced unit testing in the Agile development process of implementing our code. A python segment test script was kept to check that individual segments produced the correct output. Once a segment of code produced the exact same output on a small set of test data in Python as the Matlab code produced, that segment was moved into the full algorithm. This process helped us avoid digging through the entire algorithm to find bugs, but catch the bugs at an earlier state.

Our team worked well together. We all knew that we played different roles in the completion of the project, but we were open to helping each other in our individual struggles. We would meet with Richard every other day as a team so that we were all on the same page with questions and progress. On top of that, we would also meet typically at the end of the work day and again on the day we didn't meet with Richard to discuss issues and questions. We were always on the same page about what needed to get done, helping each other meet our expectations, and complete the project.

We faced a lot of challenges and interesting technical details throughout the completion of this project. First of all, we discovered some poor programming in the Scipy sparse matrix implementation. Essentially, while using this data structure, we were unable to update locations in the sparse matrix once the matrix reached a large enough size. This is extremely inefficient because our Nearest Neighbor model requires the sparse matrix to be updated segment by segment. Another issue that we faced dealt with Numpy arrays and Numpy matrices. What we found out was that when we used specific methods on Numpy matrices, it would return a Numpy ndarray. This prevented us from future computations, and thus we had to add another line of code converting the ndarry back into a Numpy matrix. We all had quite the learning curve in Matlab. It was a program and language none of us had ever used. On top of that, Dr. Wellman's code didn't have a single line of comments in it. The task of transcribing a new language where the code is uncommented is difficult and tedious, as we need to check what every symbol in each line of code is doing exactly. Finally, due to the high level of math used in the algorithm, understand Vikas Sindhwani's paper took a lot of time. We all read the paper within the first week approximately thirty times. We were still understanding the logic behind the math that drives this algorithm, but we know a lot more than what we did on the first day of this project.

Summary

We successfully implemented both the BNSM algorithm and the S3PCM algorithm. In both models, we are able to fit the model to a set of data, predict, score, decision function, and show a contour plot of the decision function. We were hoping to see the exact same results in both algorithms, and this was a success. Below you see two plots, where plot 1 represents the decision function of the BNSM algorithm and plot 2 is for the S3PCM algorithm. For this experiment, we used a classic machine learning data set, the two moons data set. What is special about this experiment, is that not all of the data is labeled. In the two charts below, the red data points represent labeled data, while the blue points are unlabeled. We trained the model with 67 data points, and then predicted on the remaining 33. The score is also displayed, thus showing that the algorithm works in the exact same way for both implementations. Notice that the decision line is nearly the exact same in both algorithms although the implementation of each is extremely different. This is a success.

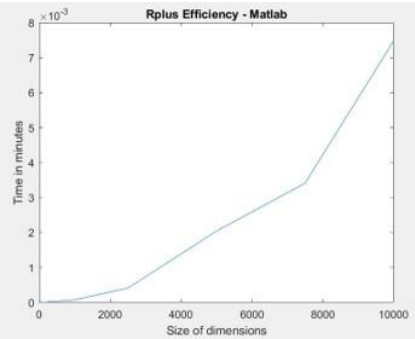


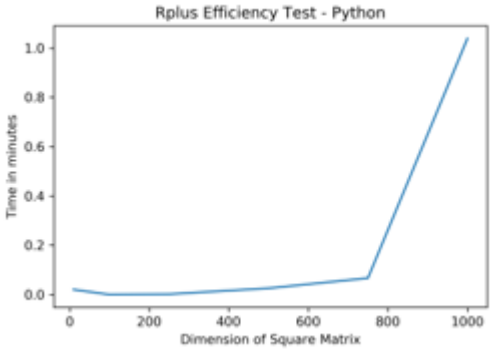
We did not succeed with everything. For all the research and work we put into finding a way to implement this at scale, we failed. Below are two charts representing the profiler runs of each algorithm. This is depicting the time used in each algorithm, and we can see that when we increase the number of samples, we see a huge difference in time. The red table consisted on fitting the model with the full, toy, breast cancer data set. This data set consisted of 569 samples with 30 features. The blue table fit the model with a cut of the MNIST data set consisting of 3000 samples with 784 features. Notice that the NN model jumps from tenths of a second to minutes in Python, while the Matlab code jumps from hundredths of a second to tens of a second. This means that we did not scale the Python implementation equivantely to the Matlab implementation. Although we see a failure, there is a success to notice in the charts. Notice that the times from the red table are not significantly different in Python and Matlab. Thus we feel like our implementation was done at best with the tools we used. And the future work will be to find and use other tools to step up the efficiency of our algorithm. In fact, our spectral basis function was faster than the matlab function in both cases which is a great success. The main issue with the Nearest Neighbor model and Riemann Matrix script is that they contained a series amount of binary singleton expansions, which is where Numpy is not efficient enough.

Code	Python - BNSM	Python – S3PCM	Matlab - BNSM	Matlab – S3PCM
NN Model	0.164 sec	0.164 sec	0.075 sec	0.051 sec
Graph Laplacian	0.004 sec	0.004 sec	0.002 sec	0.004 sec
Riemann Matrix	n/a	0.067 sec	n/a	0.055 sec
Spectral Basis	n/a	1.49 sec	n/a	1.93 sec

Code	Python - BNSM	Python – S3PCM	Matlab - BNSM	Matlab – S3PCM
NN Model	3.38 min	3.38 min	0.653 sec	0.69 sec
Graph Laplacian	0.023 sec	0.023 sec	0.006 sec	0.018 sec
Riemann Matrix	n/a	2.82 min	n/a	0.21 sec
Spectral Basis	n/a	3.63 min	n/a	4.28 min

It is important to talk about why our algorithm isn't scaling the same way that the Matlab implementation does. The main reason is binary singleton expansion. Like we mentioned early, this computation is completed a lot in both the Nearest Neighbor model as well as the Riemann Matrix file. Our implementation of binary singleton expansion in Python was not up to par with the ability that Matlab provides. Our binary singleton expansion functions are titled Rprod and Rplus. Rprod computes the expansion with production, and Rplus with summation. Below is the efficiency comparison between Rplus in both Python and Matlab. Notice that in Python, the function is already taking a minute with a matrix of size 1000x1000. However, in Matlab, with the same size matrix, we are taking less than 0.0001 seconds to compute this value. This is our problem, and this is why our algorithm will not scale.





At this point, we can see what the future work will include for this project. We need to find an optimal way to calculate binary singleton expansions in Python. After extensive research, we did not find a Python library that can do this for us. Hence, the solution will either be to code a C module computing this function, and then call it from Python, or even step up the efficiency and code this computation in Assembly language. Although we are disappointed we didn't maintain scalability at this point, we feel like it was a success as we have discovered the main function in the algorithm that needs to be efficiency implemented such that it matches up with the efficiency that Matlab provides.

We do see this project as a success. We have implemented the semi-supervised point cloud machine correctly in Python. Now the continued work will be to step up the efficiency of the code that we have written.

References

Our code is highly documented, however because this project was done for Professor Richard Wellman, the Github repository is private. If someone is interested in continuing work on this project, please reach out to Professor Wellman in order to gain access to the algorithm.

A series of literature was used in the completion of this project. Mainly, we invested the paper by Vikas Sindhwani and his team at the University of Chicago. The citation is in the appendix.

Appendices:

In order to use the code, there is a series of test files. The TSVM_tester is a series of three experiments that use the fit, predict, score, and show_contour methods. The TSVM_segement_tester is our unit testing file where we test little segments of the code. The profiler tester was used in order to showcase how our algorithms in Python do not scale at the same rate at the code in Matlab.

Citation

Sindhwani, Vikas, et al. “Beyond the Point Cloud: from Transductive to Semi-Supervised Learning.” *Proceedings of the 22nd International Conference on Machine Learning - ICML '05*, 2005.

“Welcome to Cython's Documentation.” *Welcome to Cython's Documentation - Cython 3.0a0 Documentation*, docs.cython.org/en/latest/.

Kwmsmith. “Kwmsmith/Scipy-2015-Cython-Tutorial.” *GitHub*, 6 July 2015, github.com/kwmsmith/scipy-2015-cython-tutorial.