Improving System Configurations using Domain Knowledge Assisted Semi-Supervised Learning

Negar Mohammadi Koushki, Sanjeev Sondur, and Krishna Kant
Computer and Information Sciences
Temple University
Philadelphia, USA
{koushki|sanjeev.sondur|kkant}@temple.edu

***Abstract*—In configuration engineering, the impact of configuration variables on a system's behavior is complex and cannot be easily expressed analytically due to their non-linear and interdependent nature. The recent use of machine learning (ML) models to tackle this problem has been limited by the narrow range of available training data, leading to a *data bottleneck*. To overcome this challenge, this paper proposes a novel methodology that employs semi-supervised learning (SSL) and data augmentation (DA) techniques to generate pseudo-labeled data that closely resembles the real dataset. By incorporating domain knowledge into the data creation process, the proposed methodology allows experienced users to utilize their qualitative behavior in machine learning algorithms. The paper examines available SSL and DA methods and identifies their limitations in IT configuration studies, proposing alternative ways to adapt these techniques to configuration studies. The study's findings demonstrate that the proposed data augmentation methodology's newly created pseudo-labeled data can support data-hungry applications. This study represents the first attempt to extend SSL techniques to address the data bottleneck limitations in configuration studies.**

***Index Terms*—configuration, semi-supervised learning, data argumentation, pseudo-labelled data, confidence measure,**

I. Introduction

In an enterprise IT infrastructure, every major entity (e.g., application, virtual machine/container, infrastructure services, and hardware) have many *Configuration Variables* (CVs), often also known as configuration parameters that can be set at run-time to some value within their specified range. Although many instances of an entity (e.g., servers, VMs, application instances, etc.) have identical configuration, there is still a lot of variety in configuration, often necessitated by different roles, resource amounts, and workloads. Many CVs are considered as "baseboard" and remain at their default values and may not be well documented or understood, but the impact of the specified settings or dependencies between them and other CVs is still there and may lead to problems. Finally, while the CVs typically concern different aspects (e.g., performance vs security, network vs. storage, layer2 vs. layer3 network CVs, etc.), there are often dependencies that may or may not be known. In other words, the mere existence of a large number of CVs in most systems can lead to unexpected behavior due to inadvertent incorrect settings or unexpected interactions between the CVs and the workload. This results in the widespread and well documented problems of misconfigurations whose impact may range from suboptimal performance to large scale service disruptions [12, 22, 28].

This research was supported by NSF grant CNS-2011252

There is often a tendency to define a large number of CVs for each subsystem, particularly the software to provide flexibility, but without clear description or thought behind them, it only complicates the task of properly setting their values for given goals such as a performance and/or cost target. CVs also often take only a specific set of values in a continuous space. For example, depending on the "P"-state, a CPU may run at 3.7 GHz, 3.3GHz, 2.5GHz, and 2.0GHz but not at other frequencies. While these can be represented by integer indices (e.g., 4, 3, 2, 1), such a translation loses information about relative values. A related issue is that of the workload, since the output (e.g., performance) depends on the workload as well, and it is possible that the abnormal workload patterns are not exercised until they actually happen.

The methods available to system administrators to avoid or cope with misconfigurations are rather limited. Given the complexity, neither a detailed simulation model nor a suitable analytic model (e.g., queuing network models) is adequate to set the CVs properly [21, 22, 26, 28]. The sheer number of configurations also rules out a comprehensive experimental characterization even if that were allowed. In reality, production systems are *not* available for experimentation and generally use configurations that have been tested out on the test cluster. The configuration settings from the latter may sometimes be difficult to translate to the production system due to much smaller scale and the fact that they are not driven by the live workload.

ML models can, in theory, capture various types of correlations between CVs and the dependence of the output on the workload and the configuration. However, their accuracy and reliability entirely depends on the availability of data that covers all important portions of the state space reasonably well. Note that the ML model could well be trained on a large amount of available configuration data, but if that data represents only a small portion of the entire configuration and/or workload space, the model is unlikely to be accurate or reliable. In fact, the sparse or no coverage of the vast portions of the configuration space mentioned above makes a blind use of ML models dangerous and misleading as we have shown in [**?**]. Furthermore, collecting a large set of configuration data is often a demanding task [22, 7] leading to a *data bottleneck problem*. In addition, ML and AI are generally domain-agnostic and treat the datasets in the same way [6]. ML models can, in theory, capture various types of correlations between CVs and the dependence of the output on the workload and the configuration. However, their accuracy and reliability entirely depends on the availability of data that covers all important portions of the state space reasonably well. Note that the ML model could well be trained on a large amount of available configuration data, but if that data represents only a small portion of the entire configuration and/or workload space, the model is unlikely to be accurate or reliable. In fact, the sparse or no coverage of the vast portions of the configuration space mentioned above makes a blind use of ML models dangerous and misleading as we have shown in [**?**]. Furthermore, collecting a large set of configuration data is often a demanding task [22, 7] leading to a *data bottleneck problem*. In addition, ML and AI are generally domain-agnostic and treat the datasets in the same way [6].

Semi-supervised learning (SSL), where we create additional *pseudo-labelled* data-points for enhancing the model training can be helpful [23]; however, to capture behaviors beyond what is contained in the raw data, it is essential to make use of the "domain knowledge" that is invariably known to administrators and routinely used informally for setting configurations and diagnosing problems. The purpose of this paper is to propose a methodology to make use of such knowledge in creating additional labelled data for SSL with the goal of coping with the limited coverage of the configuration space. We consider several SSL methods for this [7, 23] and demonstrate their relative performance in the configuration context. To the best of our knowledge, this is the first contribution towards extending SSL techniques to address the limitations of data-bottleneck in configuration studies.

The rest of the paper is organized as follows. In section **??**, we discuss several SSL techniques and propose alternative ways to adapt them to configuration studies. In section **??** we described our method for determining the confidence level of generated data based on the domain knowledge. Section VI then shows the evaluation results. Finally, section VIII concludes the discussion.

II. Data Driven Modeling of Configurations

A. Configuration Management Problem

Consider a system with M CVs denoted by the set $C = \{C_j = j = 1..M\}$ where C_j is the name or ID of the jth CV. Each C_j has a current value between the lower and upper bounds ("lb", "ub") $[lb_{C_j}, ub_{C_j}]$. We assume that these CVs are relevant wrt some measurable system behavior O (e.g., e.g., throughput, latency, etc.) under some workload W. Thus, the specific values of the triplet (W, C, O) provide the data for data driven models, where (W, C) are inputs and O is the output (or "label" in ML terms). For notational simplicity, we denote the data as a pair (x_i, y_i) where y_i is the output and x_i is a vector $x_i = \{x_{ij}, j = 0..M\}$ where x_{i0} denotes the workload measure and $x_{ij},\ j > 0$ the selected value of the CV C_j.[1] Our entire dataset then can be denoted as $D = (x_i, y_i), i = 1..N$. Note that even if N is large, many of the data points are likely to be for the same or similar configuration driven by different workload parameter values. That is, the number of distinct configurations, say N_c, is generally quite limited.

B. Supervised and Semi-Supervised Models

A supervised ML algorithm attempts to implicitly learn the dependence of the output y_i on the input vector $\vec{x_i}$. It could then predict the label for unseen configuration $x^{\vec{(a)}}$, as shown in Fig. **??**.

Semi-supervised learning augments the labelled data with additional data to increase the accuracy of the model [7, 32]. In *transductive learning*, the task is to directly determine the labels for all unlabelled data of interest without learning a model first. Many of the early graph based methods are of this nature [32] and essentially derive a label for the unlabelled point based on the labels of labelled datapoints that are adjacent in the graph model. On the contrary, inductive learners assign *pseudo-labels* to the unlabelled data and use it (in addition to labelled data) to learn a model. In this paper, we focus entirely on inductive methods.

C. Characterizing Pseudo-labels

It is clear that a pure-data driven Semi-supervised learning (SSL) cannot derive accurate pseudo-labels for the unlabelled data unless the underlying marginal data distribution $p(x)$ over the input space x provides some information about the posterior distribution $p(y|x)$ [7]. Generally, this is stated as a "smoothness" assumption, i.e., if inputs x_i and x_j are close, then their labels y_i and y_j should also be close. The closeness relationship is usually not transitive, which makes it difficult to draw conclusions based on it. Yet another way to state the same thing is through the clustering assumption, which states that high density areas of inputs are likely to share the same label. Conversely, the boundaries in the input space that separate the output labels should pass through the low density areas of the input.

In applying such properties to the configuration problem, we first need to transform the input parameters so that one could meaningfully speak of the closeness. For example, the storage space (including caches, memory, and secondary storage) generally (but not necessarily) increases in multiplicative steps (e.g., 32, 64, or 128 GB memory), and the performance increase, if any, from doubling is far less than double. Thus a log compression of the inputs becomes necessary before considering closeness. Real configurations are also likely to be limited to discrete values rather than continuous [22]. For example, based on the available DIMMs, the memory could be 32GB, 48GB, or 64GB, but unlikely to assume other values between 32 and 64. Similarly, the NICs may support 10, 40, or 100 Gb/sec bandwidth. Also, each parameter will have a

[1]The use of a single parameter x_{i0} to denote workload is for notational simplicity – one could surely represent workload by several aspects such as intensity, burstiness, and even composition of different types of transactions.

range that is sensible based on the domain knowledge, even if other values are feasible (e.g., minimum memory of 32GB).

On the output side, although the performance would generally vary smoothly with the resources, this is not true when bottlenecks are involved. For example, if the bottleneck in the memory bandwidth causes long access delays for the CPU, a slight increase in the memory bandwidth could increase the processing rate quite considerably. Similarly, as the resource amount goes below some minimum level, an extreme resource contention (e.g., thrashing) may kick in, thereby killing off the performance. Another crucial issue in configuration studies is that the data points (configurations) are far from representative of the configuration space. Most of the configurations for which we have the data are those that were chosen simply because they perform well; others are likely to be much more sparse. Thus, a pure data-driven pseudo-labelling is likely to fail in configuration problems.

The SSL literature uses the notion of *low-dimensional manifold*, which can be thought of as the surface where points with similar output values (labels) cluster together. In the configuration context, a manifold often arises because of the compensatory effect of different CVs; for example, a configuration of a small amount of DRAM and a fast IO device may provide similar performance as the one with a large amount of memory and slow IO device. The manifold is generally an approximate, low-dimensional representation of such behavior, and in practice like the rules of thumb that the administrators already know.

Consider the 3D configuration space consisting of CPU speed, memory bandwidth, and memory size. One could then identify a 3D surface for each performance level (or output label), which is the true manifold. If the low dimensional approximation to this manifold is approximately correct, for any given target performance, very few points should lie outside the corresponding surface. That is, the density of such points should be low, and that can be used as a means for identifying the boundaries of regions with different performance levels (or labels).

D. Generating Pseudo-labels

The clustering and related assumptions above provide one way to produce the pseudo-labels [20, 7, 32]. Another one, known as a wrapper methodology, starts with some *base learners* that are initially trained only on the labelled data. These learners could then be used to predict the pseudo-labels, and the *pseudo-labelled* data could then be recycled back for further training of the learners. There are many methods to do this and we shall discuss them in the following. The generated pseudo-labelled data is then recycled back to train the base learners further. The process can be repeated multiple times, usually until the performance on the validation data has stabilized, but there is no guarantee that this will happen or the pseudo-labelled data will improve the results.

It is generally a poor idea to admit all generated data from the base learners; instead, we need a reliable measure of confidence in the label prediction, and retain only those data items for which the confidence level is above some threshold (e.g., 90%). Although most supervised learning algorithms produce a confidence level measure in the last step (usually using softmax), it is not necessarily a reliable measure. The pseudo-labelled data could then be used like the labelled data for additional training of the base learners, either starting with the result of original training or from scratch with original and pseudo-labelled data intermixed.

III. Generating and Labelling Data

A. Generating Additional Input Data

For SSL, one often needs to generate suitable input data before trying to put pseudolabels on it. An obvious approach is to randomly choose a value for x_{ij} in the range $[lb_{C_j}, ub_{C_j}]$ for each j. However, doing so would ignore the dependencies between the CV's and may result in unrealistic configurations. To address this, we can make use of the domain knowledge in form of sensible ranges for other CVs relative to some key CV (e.g., memory relative to CPU capacity or number of TCP connections for the target throughput).

It is also possible to use deep neural nets, known as deep generative models (DGM) [9**?**] that learns and preserves the correlations across different dimensions. There are 3 popular DGMs: (a) Normalizing Flows (NFs), (b) Variational Autoencoders (VAEs), and (c) Generative Adversarial Networks (GANs). NF uses a sequence of invertible functions $f_i, i = 1..m$ (for some $m > 1$) applied to n-dimensional input data to reduce it to a multidimensional Gaussian distribution. After training, the mean and/or variance of the distribution could be perturbed to generate additional data. The main problem with NF is invertibility, which precludes dimensional reduction, and makes it largely impractical for configuration studies.

A variational autoencoder is a variant of normal autoencoder (AE) that learns the low dimensional distribution (rather than a direct representation) of the data. Typically, VAE maps all input data to the multidimensional normal distribution, with the loss function minimizing both deviation from the mean/variance of $N(0, 1)$ and reconstruction error. VAE can generate data by using a samples of $N(0, 1)$ distribution. The introduction of $N(0, 1)$ is rather arbitrary and tends to make the generated output "smooth" and concentrated around the mean. Also, VAEs require significant amount of data to train.

GANs can be used to generate any kind of multidimensional data [18, 10], and work by co-training two distinct "adversarial" networks, known as generator and discriminator. The discriminator is rewarded for training itself (using real data) and for flagging the generated data as fake. The generator instead is rewarded for generating data that the discriminator labels as real. The discriminator and generator play this minmax game alternately until a Nash equilibria is reached. GANs generally require enormous amount of data to train and have many other issues [9, 16], and not suitable for configuration studies.

B. Data Labeling and Estimating Label Confidence

With the typical black-box ML models, it is difficult to be sure about its prediction on the previously unseen dataset, particularly if the unseen data is quite different from the data used for training [4]. Although most neural nets provide or can be coaxed to provide confidence level for the output label, the probabilities are generally inaccurate. In particular, the decision trees and related models (e.g., random forest) are known to provide a rather poor estimate of the label probability since they focus on minimizing the tree size and enhancing classification accuracy rather than the probabilities [17]. Augmenting ML with either the explicit domain knowledge can enhance the quality of the results [2, 6, 30]. We have demonstrated this in our earlier work in the configuration context [5, 15]. In particular, we have examined the notion of quality of configuration by defining a *Configuration Health Index* (CHI) that quantifies how well the system is configured based on the expected behavior of the output y with respect to the different components of the configuration vector $\vec{x}$ [22].

Intuitively, the CHI measures the variation of y_i w.r.t x_{ij} for a specific CV C_j while considering other CVs to be set at their nominal values. For example, if index j refers to CPU cores, the CHI may indicate performance vs. #cores. Both the x and y values are normalized to the range 0..1 for this, and thus the y value is expressed as a *weight* rather than intended output. We assume a specific form for this curve based on the domain knowledge. For example, if y connotes to some performance measure and x to some resource amount, the generic behavior is that of an initial rise in y with x but at a diminishing rate, and eventually either saturating or even drooping (due to bottlenecks/overhead). The parameters of this curve are determined from the available data. This method implicitly accounts for the dependencies across the CVs in addition to the dependency of output on each specific CV. We have found it to be extremely successful in accurate prediction even in cases considered to be difficult [22]. Such an independent metric (Eq. 1) will add to the credibility in the pseudo-labeled data $\vec{x^{(a)}}$ provided that the data used for estimating CHI parameters is separate from that used for training the ML model.

The subfigures in Fig. 1 show the CHI measure as a function of different CVs for the BitBrains dataset as reported in [22]. The pseudo-labeled data-points z is as red diamond points ♦ and the *original* data-points x shown as black diamond , thus showing the difference between the pseudo-labeled data and the original system.

Thus CHI can provide a way of measuring the confidence of the dataset $(\{\vec{x^{(a)}}, y^{(a)}\}$ independent of a ML model. Let $y_e^{(a)}$ denote the label estimated from CHI. Assuming that both $y^{(a)}$ and $y_e^{(a)}$ are normalized to the range 0..1, we propose to estimate the confidence ε via the RMS error

$$\varepsilon = 1 - \|y^{(a)} - y_e^{(a)}\|^2 \qquad (1)$$

Perturbation (ρ) **:** One of our key contributions is to extend the metaheuristic that exploits the application domain knowledge in grouping and choosing the parameters for perturbation. The perturbation function ρ in Eq. **??** refers to the varying influence of parameters that can guide which parameters need to be perturbed and by how much to get to the next data-point. This perturbation aspect is an estimate of the amount by which one needs to move the current data-point (i.e. configuration z) to another region that can possibly provide a better neighbourhood point.

(Dr.Kant - need help-should we include Tunnelling?)

Base-predictor: For each artificially generated data z, we consult the base-predictor to get an estimate of the output label, i.e estimated behavior y'. We cautiously avoid using the original ML base predictor (trained on the real world experiment data) to generate the label y', as this would result in a yet-another ML model. Dr.kant- need help... Instead, we use the probabilistic prediction based on the manifold assumption to fairly infer accurate prediction based on the probability of data-distribution(i think they use Lapaclian graphs). Thus, the base learner used as a black box predictor (Fig. **??**) is an independent model....xxxx.

In our prior research [15], we presented an approach for generating data using an improved metaheuristics-based method. Our approach focuses on efficiently solving constrained optimization problems by leveraging domain knowledge and selecting optimal configuration parameters for perturbation.

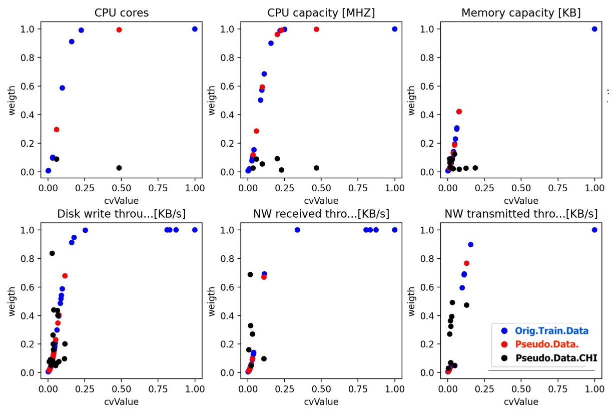


Fig. 1: Confidence procedure.

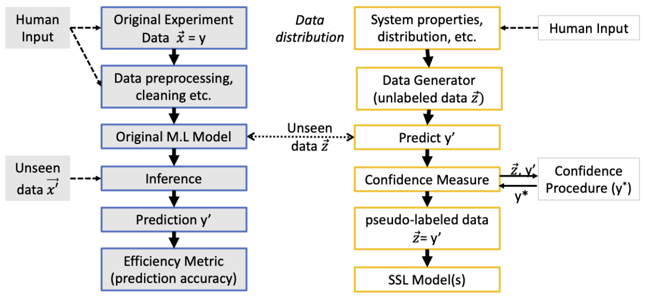

Human Input
Original Experiment Data $\vec{x}$ = y
Data preprocessing, cleaning etc.
Original M.L Model
Unseen data $\vec{x'}$
Inference
Prediction y'
Efficiency Metric (prediction accuracy)
Data distribution
Unseen data $\vec{z}$
System properties, distribution, etc.
Human Input
Data Generator (unlabeled data $\vec{z}$)
Predict y'
Confidence Measure
$\vec{z}$, y'
y*
Confidence Procedure (y*)
pseudo-labeled data $\vec{z}$= y'
SSL Model(s)

Fig. 2: Generation of pseudo-labeled data.

IV. Methods for Using Pseudo-labeled Data

Pseudo-labelled data can be used in many ways [32]. In the following we describe and explore some of the key methods in the configuration context.

A. *Self-training*

Self-training [29] is the most basic method wherein the original supervised learning (SL) classifier is trained further using both original and pseudo-labelled data. Here one could use an expectation-maximization (EM) type of approach wherein we include all of the unlabelled data at once and then iteratively determine the parameters of the labeling process such that we maximize the likelihood that the observed data belongs to the assumed distribution of the labels. For example, Wu, et.al. [27] use Naive Bayes for the classification version of the problem. The implicit assumption here is that each dimension (or configuration variable) k contributes independently to to the label. Furthermore, by assuming that this contribution has a normal distribution, they estimate its mean and variance to maximize the likelihood that the observed data (both labelled and unlabelled) comes from the computed distribution. The Gaussian distribution is reasonable for continuous variables; for discrete values, a binomial model can be used. In self-training pseudo-labelled data may be given a lower weight than the original data; either directly (if the confidence level of each datapoint is known) or collectively through a single hyperparameter.

See loss function from [25]...
Take section 3 from [32] ...

B. *Co-training (Multiview)*

Unlike self-training, co-training involves training $K \geq 2$ distinct classifiers on the labelled dataset and then generating pseudo-labelled data for one another. Again, the generated data with confidence level above some threshold is used. If the classifiers are strongly correlated, they will likely generate very similar labels and thus the advantage of having K classifiers is lost. The required data diversity may be either natural (e.g., multi-modal datasets such as audio and video), or induced by having different classifiers focus on different features [31]. However, the selected features for each classifier should be adequate for a good prediction of the output, else, co-training could actually hurt the accuracy.

For example, for our configuration problem consider the input parameter set $S = (C_f, C_c, M_b, M_s, I_l, I_r, N_b)$, i.e., CPU frequency, #CPU cores, Memory bandwidth, Memory size, IO read latency, IO rate (ops/sec), and network bandwidth. Suppose that we split this into two subsets $S_1 = (C_f, C_c, M_b, M_s)$ and $S_2 = (I_l, I_r, N_b)$, and that we separately train two models M_1 and M_2 using S_1 and S_2 respectively. Now if S' is an instance of S, we can write it $S' = (S'_1, S'_2)$. If S' is labelled with the output (the performance) P', both M_1 and M_2 will use the same label P' in training. However, if S' were unlabelled, the inferences produced by M_1 and M_2, say, P'_1 and P'_2 respectively may be different due to inadequate information given to each model. The related issue is that of independence. If P'_1 and P'_2 are accurate, they are likely to be highly correlated (since in this case both M_1 and M_2 will likely be correctly using the underlying dependencies). A different way of constructing M_1 and M_2 to obtain two different projections of the original data (e.g., via two different AEs), but there is again likely to tension between accuracy (likely requires capturing the essential dimensions) and independence (capturing different features).

Add from section 4.2. We used 2 views (why?), how did you split the views?
Ref...(Wang and Zhou 2010). Zhou and Li (2010)
What is the loss function? How to co-train? what conditions to add more pseudo labeled data? How to add more into original data set? why? stop condition=loss condition.
Take section 4 from [32] ...

C. *Boosting*

Boosting refers to an iterative procedure for updating weights for various data points. The weight of a data-item corresponds to its influence on the loss function; this allows the misclassified data points to be given a higher weight in the next iteration and hopefully improve the classification. The most popular boosting algorithm is *AdaBoost* along with with SAMME (Stagewise Additive Modeling using a Multiclass Exponential loss function). AdaBoost is an ensemble learning method that combines weak learners, such as decision trees, to create a strong classifier. SAMME extends AdaBoost to multiclass classification problems by incorporating the concept of exponential loss, and it seeks to minimize the weighted training error [8, 11]. The algorithm can be summarized as follows [11].

1. Initialize sample weights $w_i^{(1)} = \frac{1}{N}, \forall i$ Where N is the total number of samples.

2. For iteration $t \in 1..T$

a. Train a weak learner h_t using the weights $w_i^{(t)}$.

b. Compute a measure of "error" for learner t as ratio of misclassified and all data points, i.e., $\varepsilon_t = \sum_{i=1, i \in C}^{N} w_i^{(t)} / \sum_{i=1}^{N} w_i^{(t)}$ Where C is the set of datapoints for which $y_i \neq h_t(x_i)$).

c. Compute the learner's weight $\alpha_t = 0.5 \cdot \ln[(1 - \varepsilon_t)/\varepsilon_t]$ and update the sample weights for the next iteration:

$$w_i^{(t+1)} = w_i^{(t)} \cdot \exp\left(-\alpha_t \cdot y_i \cdot h_t(x_i)\right), \forall i$$

3. The final classifier is obtained by combining the individual weak learners:

$$H(x) = \arg\max_{c} \sum_{t=1}^{T} \alpha_t \cdot h_t(x)$$

Where c represents the class labels.

AdaBoost is defined only for supervised learning. To use it in the SSL context, we initialize all weights as 1.0 and follow the weight modification above only for pseudo-labelled data. **KK:*****How to make weights 0..1?

D. Bagging

The Bagging classifier is an ensemble meta-estimator that runs the same or different base classifier on random subsets of the original dataset. The classifications from all classifiers are then aggregated through voting or averaging, resulting in a robust and reliable final prediction [1, 14].

KK:***Needs change To use bagging for SSL we gradually add a fraction of the unlabeled data to the existing labeled data at each iteration and then run the bagging classifier. **KK:*****What's the point? If intended to add 20% new data, do you add few percent at a time and then do what?

E. Scalability issues in SSL Models

Add [32], section 6.3

V. Experimental Evaluation

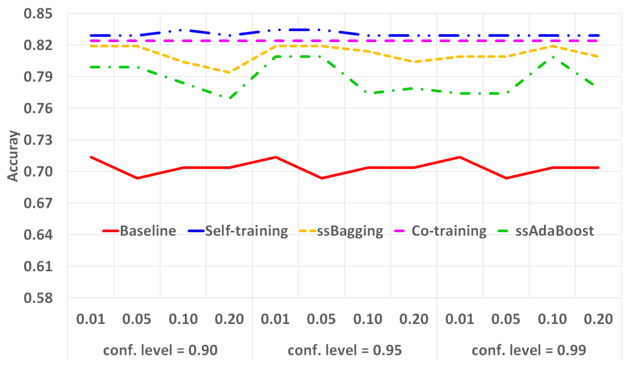
Negar to do: Restate the problem and assoc. to configuration studies in 2 sentences to lay ground. what is the criteria for validating the results? (how to validate the results? i.e artificial data generation?) What data sets we use to validate the solution. how did we address each of the sub-sections in earlier discussions? assumptions, data generation, confidence measure, validation? etc.

A. Data-set Overview: A Closer Look at the Data

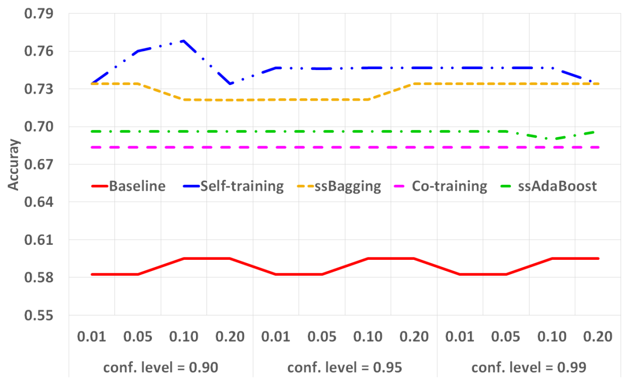
We utilized the SSL models described above on multiple publicly available data sets, which are listed in Table I. Since these data sets originate from different studies, we have no influence over factors such as data collection, experimental procedures, CVs, and variability. We will now provide a brief explanation of these data sets.

1) Cloud/Edge Storage Data-set: Edge computing provides localized computation and storage capabilities in close proximity to the data source, resulting in reduced latency and less strain on higher layers of the infrastructure hierarchy. To address the gap between low-latency local access and Cloud connectivity, Edge Storage (ES)[2] acts as a cache for remote Cloud storage, leveraging limited local compute and storage resources to enhance performance [24, 5].

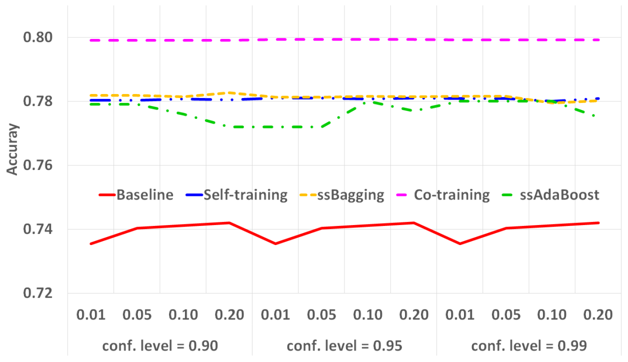
However, effectively allocating resources for ES presents challenges. It must adapt to the specific requirements of end-users' applications and devices, considering factors such as processing latency, capacity, security, location, and cost. Sanjeev et al. [5] provide a comprehensive description of the ES system, including various CVs that influence its behavior, as well as workloads and other relevant details. In this study, the ES configuration is represented by a combination of required compute and storage resources: number of cores (nc), core speed (cs), memory capacity (mc), memory bandwidth (bw), and disk IO rate (di). The workload is defined by the request arrival rate (ar), request size (rs), and metadata size (ms). We express the ES configuration as a combination of required compute and storage resources: nc, cs, mc, bw, di, ar, rs, and ms. In the context of the ES system being studied, we denote $\vec{x_i} = \{nc, cs, mc, bw, di, ar, rs, ms\}$, and $y_i = \{p\}$, representing the performance as the label.



(a) ES Data-set



(b) BB Data-set



(c) MIT Data-set

Fig. 3: Results of semi-supervised learning Models.

2) BitBrains Data-set: The research utilizes another publicly available dataset called BitBrains (BB)[3], which consists of performance logs obtained from 1,750 virtual machines (VMs) in a distributed data center operated by BitBrains. The dataset covers a span of four months and encompasses 5 million CPU hours across 5,000 cores. It primarily focuses on

[2][ES] https://www.kkant.net/config_traces/CHIproject

[3][BB] http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains (RND500) :

TABLE I: CVs and output of the data-sets.

Data-Set	**Domain**	**Size** (N^O, N^A)	**CVs** $\vec{x}$	**Output (Label** y**)**
ES [5]	Cloud Storage	991, 14717	No. of Cores, Core Speed, Memory Capacity, Memory Bandwidth, Disk IO Rate, Request Arrival Rate, Request Size, Metadata Size	Performance
BB [3]	Virtual Machines	391, 3504	No. of Cores, Core Speed, Memory Capacity, Network Data Rcvd., Network Data Transmit, Disk Read Throughput, Disk Write Throughput	CPU Usage (%)
MIT [19]	HPC Env.	97376, 20457	CPU Frequency, Resident Memory Size, Virtual Memory Size, Amount of Data ReadWrite (MB)	CPU Util. (%)

TABLE II: Comparing data accuracy through varying confidence thresholds and fractional data.

Dataset	Model	Confidence Threshold = 0.90				Confidence Threshold = 0.95				Confidence Threshold = 0.99			
		Fractional Data				Fractional Data				Fractional Data			
		0.70	0.80	0.90	0.95	0.70	0.80	0.90	0.95	0.70	0.80	0.90	0.95
ES	Baseline	0.7136	0.6935	0.7035	0.7035	0.7136	0.6935	0.7035	0.7035	0.7136	0.6935	0.7035	0.7035
	ssSelfTraining	0.8141	0.8141	0.8191	0.7990	0.8141	0.7990	0.8141	0.8191	0.8141	0.8141	0.8141	0.8141
	ssBagging	0.7638	0.7688	0.7638	0.7638	0.7839	0.7839	0.7839	0.7789	0.7789	0.7889	0.7789	0.7739
	ssAdaBoost	0.7638	0.7638	0.7688	0.7638	0.7638	0.7739	0.7688	0.7538	0.7688	0.7638	0.7688	0.7538
	ssCoTraining	0.8191	0.8191	0.8191	0.8191	0.7839	0.7839	0.7839	0.7839	0.7839	0.7839	0.7839	0.7839
BB	Baseline	0.5823	0.5823	0.5949	0.5949	0.5823	0.5823	0.5949	0.5949	0.5823	0.5823	0.5949	0.5949
	ssSelfTraining	0.7468	0.7468	0.7468	0.7342	0.7468	0.7342	0.7468	0.7468	0.7468	0.7468	0.7468	0.7468
	ssBagging	0.6835	0.6962	0.6962	0.6962	0.6709	0.6709	0.6835	0.6709	0.6835	0.6835	0.6835	0.6962
	ssAdaBoost	0.6582	0.6582	0.6582	0.6582	0.6582	0.6582	0.6582	0.6582	0.6582	0.6582	0.6582	0.6582
	ssCoTraining	0.6835	0.6835	0.6835	0.6835	0.6835	0.6835	0.6835	0.6835	0.6835	0.6835	0.6835	0.6835
MIT	Baseline	0.7354	0.7403	0.7411	0.7419	0.7354	0.7403	0.7411	0.7419	0.7354	0.7403	0.7411	0.7419
	ssSelfTraining	0.7822	0.7823	0.7818	0.7816	0.7818	0.7822	0.7817	0.7818	0.7823	0.7815	0.7823	0.7823
	ssBagging	0.7852	0.7862	0.7857	0.7868	0.7855	0.7854	0.7851	0.7847	0.7850	0.7847	0.7857	0.7844
	ssAdaBoost	0.7862	0.7837	0.7842	0.7840	0.7853	0.7859	0.7865	0.7854	0.7859	0.7864	0.7864	0.7861
	ssCoTraining	0.7990	0.7990	0.7990	0.7990	0.7993	0.7993	0.7993	0.7993	0.7992	0.7992	0.7992	0.7992

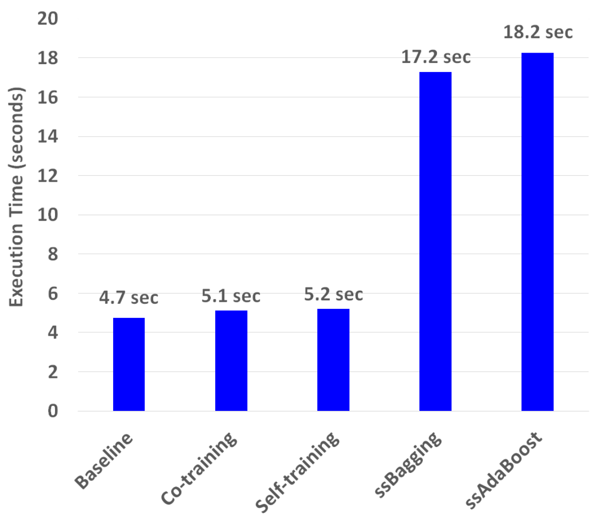


Fig. 4: Execution time comparison for the BB data-set.

providing specialized interactive services and batch processing workloads within a cloud environment, catering to managed hosting and business computation needs. Noteworthy clients include prominent banks, insurance companies, credit card operators, and more.

Iosup et al. [3] conducted a thorough analysis of requested and actual resource usage, focusing on CPU, memory, disk, and network resources. The traces contain data for each virtual machine (VM), with the initial configuration denoted as CV. Table I outlines the attributes associated with each VM configuration.

Considering the limited knowledge of the dataset specifics, we define the BitBrains VM configuration as a combination of essential resources: compute (number of cores, nc), storage (memory capacity, mc), network (receive bandwidth, $nwrd$, and transmit bandwidth, $nwwr$), and disk load (read request rate, $dskrd$, and write request rate, $dskwr$). These resources can be represented by the vector $\vec{x_i} = \{nc, mc, nwrd, nwwr, dskrd, dskwr\}$. The corresponding label ($y_i = \{cu\}$) in this context represents the CPU usage.

3) MIT Cloud Data-set: MIT recently published a comprehensive dataset[4] from their Supercloud petascale cluster [19], which involved running a diverse range of high-performance computing (HPC) workloads. This extensive dataset, totaling a massive 2 terabytes (2TB), includes time-series data from various aspects such as the scheduler, file system, compute nodes, CPU, GPU, and even sensor data from physical monitoring of the cluster's housing facility. For our analysis, we specifically focused on the second partition of the dataset, which consisted of 480 CPU nodes. Each node was equipped with a powerful configuration, featuring dual 24-core Intel Xeon Platinum 8260 processors, 192GB of RAM, and a high-performance parallel file system called Lustre, which operated on a 3-petabyte Cray L300 parallel storage array.

The key data attributes considered in this study were CPU frequency (cf), residual memory (rms), virtual memory size (vms), amount of data read/write (mw), and CPU utilization percentage (cu). To address our specific problem, we formulated the input vector as $\vec{x_i} = \{cf, rms, vms, mw\}$, and the corresponding output values as $y_i = \{cu\}$.

[4][MIT] https://dcc.mit.edu

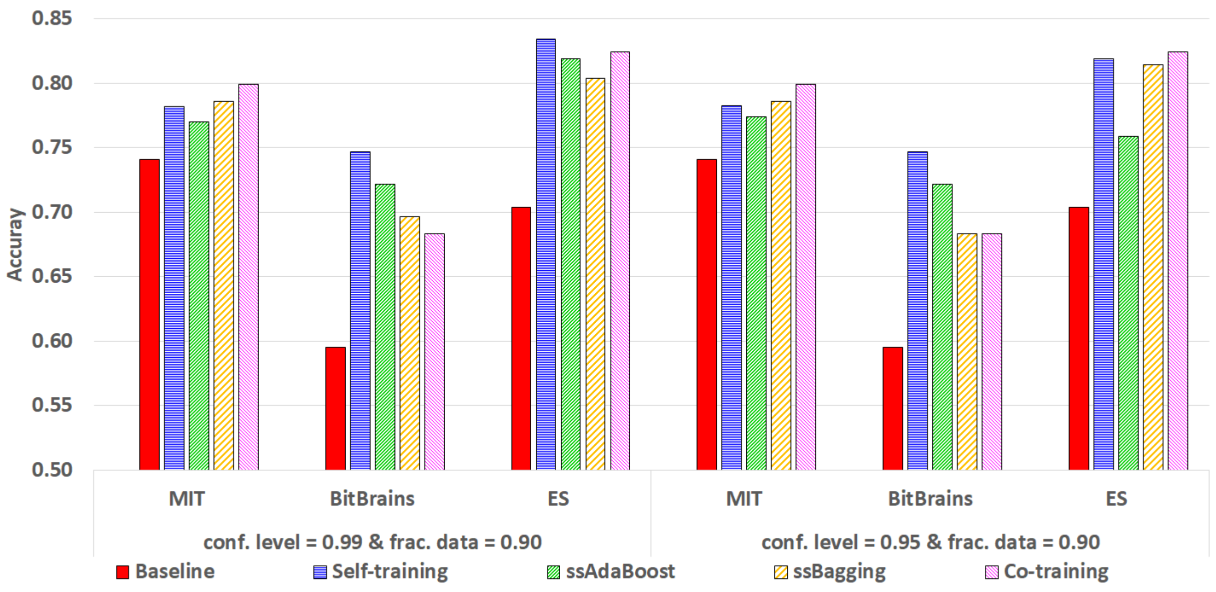


Fig. 5: Results summary.

B. *Experimental setup*

In this paper, we conducted experiments to evaluate the performance of SSL models using different datasets introduced in the above section. For each dataset, we followed a standard split where approximately 20% of the labeled data (original data) was set aside as test examples, while the remaining labeled data, along with artificially generated unlabeled data, was used as the training set. Our experiments aimed to investigate the impact of gradually adding unlabeled data to the labeled data in the training set while considering different confidence measures for the artificial data.

In order to accomplish this, we employed a confidence measure to assess the quality of the artificial data generated during the training process. Initially, we trained the model using the original data. Subsequently, we gradually incorporated unlabeled data into the labeled training set, starting with 10% and increasing it in 10% increments (20%, 30%, and so on). This iterative process allowed the algorithms to adapt to unfamiliar labels and prevented convergence to incorrect points. Additionally, we explored the impact of different confidence measures on the synthetic data. Specifically, we experimented with confidence levels of 90%, 95%, and 99%. By incorporating these measures, our objective was to examine the relationship between confidence levels and the performance of SSL methods. Through systematic variations in the quantity of unlabeled data and the confidence measures applied to the synthetic data, we were able to evaluate the accuracy of the models across diverse datasets. This comprehensive analysis enabled us to assess the influence of these factors on model performance.

In addition to evaluating SSL models, we employed a performance prediction oracle, represented as a black box function, to establish a baseline for comparison. This oracle helped us identify the best prediction model for each dataset, serving as a reference point. For instance, in the case of Dataset ES, we found that the Logistic Regression model outperformed over 20 other machine learning algorithms contained within the black box. The baseline evaluation involved training and testing the model exclusively on the original dataset, without considering any confidence measures.

As depicted in Figure 3, the results clearly demonstrate that SSL models achieved higher accuracy compared to the baseline. The SSL models consistently outperformed the baseline approach across multiple datasets. These findings provide compelling evidence for the efficacy of SSL techniques in enhancing prediction performance by incorporating both labeled and unlabeled data, as well as utilizing confidence measures to guide the learning process.

Figure 4 illustrates the execution times in seconds for different models using the BB data-set. Despite the baseline model having the shortest execution time of 4.74 seconds, it is important to consider its lower accuracy compared to other models. Consequently, focusing solely on the baseline model's execution time would undermine its significance. Conversely, the ssCoTraining and ssSelfTraining models exhibited execution times of 5.11 and 5.2 seconds, respectively, while maintaining reasonable accuracy levels. In contrast, the ssBagging model required 17.29 seconds for execution, and the ssAdaBooster model had the longest execution time of 18.57 seconds. These findings suggest that although the ensemble-based models require more time, they may provide superior accuracy compared to the baseline model.

Table I provides information on three data sets: ES, BB, and MIT. These data sets belong to the domains of Cloud Storage, Virtual Machines, and HPC Environment, respectively. The size of each data set is indicated by the numbers N^O and N^A, representing the number of original data points and artificial data points. Each data set has specific CVs listed in the table, along with their corresponding outputs. The CVs capture different aspects relevant to each data set's domain, while the outputs represent the specific target or label being predicted.

Table II compares data accuracy across different confidence thresholds and fractional data values. The fractional data values represent the amount of unlabeled data being added to train the model. The table is divided into three sections based on the confidence threshold: 0.90, 0.95, and 0.99. Within each section, the fractional data values are listed as 0.70, 0.80, 0.90, and 0.95. The table presents results for three different datasets: ES, BB, and MIT. Each dataset is evaluated using various models, including Baseline, ssSelfTraining, ssBagging, ssAdaBoost, and ssCoTraining. The table lists the accuracy values for each combination of dataset, model, confidence threshold, and fractional data value.

Upon reviewing the results, it can be observed that the Baseline model consistently achieves relatively lower accuracy compared to the other models across all datasets. The ssSelf-Training model demonstrates higher accuracy values, particularly at confidence thresholds of 0.90 and 0.95. Conversely, the ssBagging and ssAdaBoost models generally produce similar accuracy scores. The ssCoTraining model performs well, exhibiting high accuracy across different confidence thresholds and fractional data values.

VI. Results and Discussions

Important: What is your baseline for comparison or for this study???
Take ideas from [7] 2.4 Empirical evaluation of semi-supervised learning methods - why it is difficult to evaluate? but in our study we are not evaluating the Ssl models, but we are interested in config studies and generating addition pseudo-labelled data...

A. Results Quality

result 1: quality of artificial data, e.g: 90% of my artificial data has a conf level $\geq 95\%$. Quantity of data: original dataset N took 4 months to collect, artificial dataset 5xN took 4 hours to generate !!!
result 2: confidence in 'artificial dataset' i.e. confidence in 'confidence metric'!!! We used blind (unseen, keep-aside) data to eval/test the confidence model. In CHI we called it HI, here we use the loss function above as conf. metric.
result 3: quality of new pred. 'model' (e.g regression, decision tree) is close to 95% of orig model, even if we can not exceed orig.pred.accuracy....we are still close to it using 'confident artificial data'.
challenges in result 3: orig. pred model used algorithm XYZ (e.g decisiontree, logistic regression etc) and memorized some weights w1, w2, etc. New pred model "MOST LIKElY" will use the same algorithm XYZ and predict w1', w2', etc. So new pred. model maybe poor in weights/because of new dataset.

VII. Team Discussions

dataset(s) – ?? ES??
Collect N artificial points. How much is N? Let's say real data is M, N = 3*M... Where do you set the confidence threshold, 0.9? 0.8? 0.7? Which model works better?

SSL Models (to use artificial + original data combined): what percentage, what eval/test dataset, what confidence level and what fraction of unlabelled data? RESULTS? How do you eval. "final accuracy" ?

To do: Negar - (1) Existing dataset [13, 3, 5], get new data set from recent papers (sanjeev will verify if data set is usable for our config studies), Results - what do you want the paper to say- strong points- eval results

KK:***To do: Dr Kant (i) review overleaf (iii) manifold clustering probability discussion, w.r.t. predicting new labels for unseen data.

VIII. Conclusion & Further Discussions

Closing the gap between theory and practice is critical to improve the reliability of DGM training and reduce the immense computational costs. This paper has demonstrated the sampling problem in VAEs and enforcing the Lipschitz constraint in WGAN training. While most existing DMG approaches use black-box neural networks as generators, there is a lack of models for incorporating domain specific knowledge. This is a significant limitation in scientific studies.

References

[1] Leo Breiman. Bagging predictors. *Machine learning*, 24:123–140, 1996.

[2] Changyu Deng, Xunbi Ji, Colton Rainey, Jianyu Zhang, and Wei Lu. Integrating machine learning with human knowledge. *Iscience*, 23(11):101656, 2020.

[3] Iosup Alexandru et al. The grid workloads archive. *FGCS*, 24(7):672–686, 2008.

[4] Maksims Ivanovs et al. Perturbation-based methods for explaining deep neural networks: A survey. *Pattern Recognition Letters*, 150:228–234, 2021.

[5] Sanjeev Sondur et al. Towards automated configuration of cloud storage gateways: A data driven approach. In *Cloud Computing*, pages 192–207. Springer, 2019.

[6] Tirtharaj Dash et al. A review of some techniques for inclusion of domain-knowledge into deep neural networks. *Scientific Reports*, 12(1):1040, 2022.

[7] Van Engelen et al. A survey on semi-supervised learning. *Machine learning*, 109(2):373–440, 2020.

[8] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT'95, Spain*. Springer, 1995.

[9] Ian Goodfellow and et al. Generative adversarial networks. *Communications of the ACM*, 63(11), 2020.

[10] Google. Overview of GAN Structure. https://developers.google.com/machine-learning/gan/gan_structure, 2020.

[11] Trevor et al. Hastie. Multi-class adaboost. *Statistics and its Interface*, 2(3):349–360, 2009.

[12] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. Unicorn: Reasoning about Configurable System Performance through the lens of Causality. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22. ACM, 2022.

[13] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. Distance-based sampling of software configuration spaces. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1084–1094, 2019.

[14] Gilles Louppe and Pierre Geurts. Ensembles on random patches. In *Machine Learning and Knowledge Discovery in Databases: ECML PKDD 2012, Bristol, UK*, pages 346–361. Springer, 2012.

[15] Negar Mohammadi Koushki, Sanjeev Sondur, and Krishna Kant. Automated Configuration for Agile Software Environments. In *2022 IEEE CLOUD*, 2022.

[16] Jaime Perez, Patricia Arroba, and Jose M Moya. Data augmentation through multivariate scenario forecasting in data centers using generative adversarial networks. *Applied Intelligence*, 53(2):1469–1486, 2023.

[17] Foster Provost and Pedro Domingos. Tree induction for probability-based ranking. *ML*, 52:199–215, 2003.

[18] Lars Ruthotto and Eldad Haber. An introduction to deep generative modeling. *GAMM-Mitteilungen*, 44(2):e202100008, 2021.

[19] Siddharth Samsi et al. The MIT Supercloud Dataset. In *2021 IEEE HPEC*. IEEE, 2021.

[20] Chen Shani, Jonathan Zarecki, and Dafna Shahaf. The Lean Data Scientist: Recent Advances Toward Overcoming the Data Bottleneck. *Communications of the ACM*, 66(2):92–102, 2023.

[21] Sanjeev Sondur, Anis Alazzawe, and Krishna Kant. Optimal configuration of high performance systems. In *HPCS*, 2020.

[22] Sanjeev Sondur and Krishna Kant. Performance Health Index for Complex Cyber Infrastructures. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 2022.

[23] Enmin Song, Dongshan Huang, Guangzhi Ma, and Chih-Cheng Hung. Semi-supervised multi-class adaboost by exploiting unlabeled data. *Expert Systems with Applications*, 38(6):6720–6726, 2011.

[24] Polyzois Soumplis and et al. Resource Allocation Challenges in the Cloud and Edge Continuum. In *Advances in Computing, Informatics, Networking and Cybersecurity*, pages 443–464. Springer, 2022.

[25] Jafar Tanha, Maarten Van Someren, and Hamideh Afsarmanesh. Semi-supervised self-training for decision tree classifiers. *International Journal of Machine Learning and Cybernetics*, 8:355–370, 2017.

[26] Qingyang Wang and et. al. Optimizing n-tier application scalability in the cloud: A study of soft resource allocation. *ACM ToMPECS*, 4(2), June 2019.

[27] Zhiang Wu, Junjie Wu, and et al. Hysad: A semi-supervised hybrid shilling attack detector for trustworthy product recommendation. In *18th ACM SIGKDD*, 2012.

[28] Tianyin Xu and Yuanyuan Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys (CSUR)*, 47(4):70, 2015.

[29] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *33rd ACL*, 1995.

[30] Shanshan Zhang, Lihong He, Eduard Dragut, and Slobodan Vucetic. How to invest my time: Lessons from human-in-the-loop entity extraction. In *The 25th ACM SIGKDD International Conference*, 07 2019.

[31] Zhi-Hua Zhou and Ming Li. Semi-supervised learning by disagreement. *KAIS*, 24:415–439, 2010.

[32] Xiaojin Jerry Zhu. Semi-supervised learning literature survey. *CS Technical Reports*, 2005.