Abstract—Limited access to the supervised information may
forge scenarios in real-world data mining applications, where
training and test data are interconnected by a covariate shift,
i.e., having equal class conditional distribution with unequal
covariate distribution. Traditional data mining techniques
assume that both training and test data represent an identical
distribution, therefore suffer in presence of a covariate shift.
Kernel Mean Matching (KMM) is a well known approach to
address this important challenge by weighing training instances
appropriately to diminish the effect of covariate shift. However,
it has time complexity cubic in the size of training data,
which is computationally impractical for large or streaming
datasets due to limited scalability. In this paper, we present

Keywords—Sampling Bias Correction, Covariate Shift, Kernel
Mean Matching, Distributed Computing, Spark

I. INTRODUCTION

Traditional data mining techniques for classification make
an important fundamental assumption known as “stationary
distribution assumption”. According to this assumption, the
set of examples on which a model is trained, and the test set
on which the trained model needs to make predictions, both
come from the same underlying distribution [1]. However,
due to limited supervision or lack of control over the data
gathering process, often the above assumption is not practical in real-world applications.

For example, suppose we want to build a model to predict
presence of an animal species based on characteristics of a
geographical location. Since it is easier to collect supervised
data from certain locations than others, it is expected that
the training data will consist of mostly examples drawn from
easily accessible locations. However, test data may come
from any location. Therefore, it introduces sampling bias
between training and test distributions.

For another example, suppose we need to build a model
to predict sentiment of tweets [2]. Typically sentiment is not
provided as ground truth along with the tweet. So, in order to
collect training data, a few users may agree to provide tweets
with sentiment label information. These users represent only
a small portion of the population. However, tweets on which
the model needs to analyze the sentiment may come from
any Twitter user. Clearly, there is a sampling bias between
training and test distributions. This is a scenario where both
training and test data are collected in a stream.

Recent studies have identified the sampling bias problem
in areas such as natural language processing [3], computer
vision [4] and text mining [5]. Traditional classifiers based
on “stationary distribution assumption” greatly suffer in
presence of sample selection bias [1]. Most algorithms
address this first by estimating the distributions, and then by
making appropriate corrections based on the estimation [6].

However, estimating distribution from multidimensional data
itself is known to be a hard problem [7].

There are a few approaches available in the literature
that address sampling bias without estimating the biased
probability densities. Most of these approaches assume that
the training and test data distributions, denoted by \( p_{tr}(\cdot) \) and
\( p_{te}(\cdot) \) respectively, are related by a covariate shift assumption.

More specifically, the relationship between the training and
test data distributions are such that \( p_{tr}(y|x) = p_{te}(y|x) \)
and \( p_{tr}(x) \neq p_{te}(x) \), where \( x \) and \( y \) denote covariates and
label of the data instance respectively. This assumption,
known as covariate shift [6], is necessary since for arbitrary
\( p_{tr}(y|x) \) and \( p_{te}(y|x) \), there is no way to infer a good
estimator for test data based on the training data [6].

Kernel Mean Matching (KMM) [6] is a widely used
method for correcting sampling bias directly without esti-
mating training and test distributions. It calculates density
ratio estimates, denoted by \( \beta(x) = \frac{p_{te}(x)}{p_{tr}(x)} \) for each training
instance \((x, y)\), directly by minimizing mean discrepancy
between the training and test data distributions in a Re-
producing Kernel Hilbert Space (RKHS) [8]. These density
ratios are then used to adapt the given training data for
learning an appropriate model to perform prediction on test
data. These density ratios represent the so-called importance
weight [9] (or instance weight) for training data instances.

KMM solves a quadratic optimization program (more
details in Section II-C) to estimate importance weights for
training data instances. This approach has a time complexity
cubic in the size of training data, and linear with respect
to the test data size. Therefore, despite being very useful

Ahsanul Haque*, Zhuoyi Wang*, Swarup Chandra*, Latifur Khan*, and Charu Aggarwal†

*Department of Computer Science, The University of Texas at Dallas, Richardson TX, USA
Email: \{axh129430, zw151030, src093020, lkhan\}@utdallas.edu
†IBM T. J. Watson Research Center, Yorktown NY, USA, Email: charu@us.ibm.com
in addressing sampling bias, KMM often becomes a bottleneck when employed in data mining operations such as data stream classification [10], where a classifier needs to be updated periodically. Moreover, computations in KMM require the whole training and test dataset to be in the memory. In scenarios where the dataset is distributed across multiple systems, one cannot directly employ KMM to perform density ratio estimations for the complete dataset.

While one can use a smaller subset from the original dataset (especially from the training data due to KMM’s time complexity) to overcome the above challenge, the estimated density ratios depend on the distribution represented by the corresponding subset. Therefore, this subset should preserve the original data distribution. Constructing such a data subset that satisfies both these requirements is challenging. First, this subset may have a completely different data distribution from its original superset. Second, data patterns in the original dataset may not be captured within this subset, which might limit its relevance in applications such as data classification.

In this paper, we present a sampling based approach to address the limited scalability problem in KMM. More specifically, instead of using the whole training and test dataset at once, we generate a number of bootstrap samples of size \( m \) from the training data of size \( n \) (\( m < n \)). The number of training samples is determined in such a way that each instance in the original training set is selected at least once with very high probability. Additionally, we split the test dataset into a number of partitions. Then, we form train-test components by considering each possible pair of training sample and test partition. Next, we apply KMM on each of the components separately. Finally, we aggregate instance weights calculated from the components to estimate weight for each training instance.

Importantly, we show that Kernel Mean Matching (KMM) can be applied to the train-test components independently from each other. Therefore, the presented approach can be applied to large datasets in parallel and distributed fashion. It can reduce the overall time of calculating instance weights due to small size of the components. Moreover, increasing number of training samples and test partitions, i.e., increasing number of components help improving quality of density ratio estimation, without increasing the execution time of this approach. Motivated by this, we propose a distributed version of the sampling approach. We implement it using Apache Spark, a distributed cluster computation framework. Experiment results show a significant speed up achieved by the distributed version while maintaining very competitive estimation accuracy.

The primary contributions of our work are as follows:

1) We present a method to address limited scalability of KMM by dividing training and test data into samples and partitions respectively, and applying Kernel Mean Matching (KMM) on each pair of sample and partition referred to as components.

2) We show that KMM on different components can be applied independently in parallel and distributed manner. Therefore, we propose a distributed version of the sampling algorithm.

3) We implement the algorithm using Apache Spark. We discuss about the design challenges of implementing this algorithm in a distributed environment, and propose design choices to address those challenges.

4) We conduct a number of experiments over benchmark datasets. The experiment results show that our proposed approach achieves significant speed up over the centralized algorithm. Moreover, experiment data also indicates that using a larger training data only improves the estimation accuracy, with minimal or no effect on execution time.

The rest of this paper is organized as follows. We review relevant studies in covariate shift, briefly discuss about the Kernel Mean Matching method, and Apache Spark framework in Section II. In Section III, we describe our proposed approach, along with various design challenges. We then present experiment results in Section IV, and conclude the paper in Section V.

II. BACKGROUND

A. Notations

Table I lists frequently used symbols in this paper. In general, we use a bold letter to indicate a set, and a capital-bold letter to indicate a set of sets. Since a set may contain multiple elements, these are indexed by a subscript integer. For example, \( X^{tr} \) denotes training data covariates, and \( X^{tr}_{i} \) denotes the \( i^{th} \) data instance in the training data. A hat or tilde over any symbol indicates estimated value.

B. Covariate Shift

As discussed earlier, sampling bias between training and test data distributions can be addressed under covariate shift assumption, i.e., when \( p_{tr}(x) \neq p_{te}(x) \) with \( p_{te}(y|x) = p_{te}(y|x) \). In general, covariate shift is accounted by computing an importance weight \( \beta(x) = \frac{p_{tr}(x)}{p_{tr}(x)} \) for each training instance \( x \), and using them in the learning process. Recent studies have focused on computing \( \beta(x) \) without explicitly estimating \( p_{tr}(x) \) and \( p_{tr}(x) \). These include Kernel Mean Matching (KMM) [6], Kullback-Leibler Importance Estimation Procedure (KLIIEP) [11], and unconstrained Least Square Importance Fitting (uLSIF) [12]. Although these methods provide elegant solutions for estimating importance weights, they suffer from limited scalability on large datasets.

C. Kernel Mean Matching

The idea in Kernel Mean Matching (KMM) is to minimize the mean distance between weighted training data distribution \( \beta(x)p_{tr}(x) \) and corresponding test data distribution...
\[ p_{\text{te}}(x) \text{ in a Reproducing Kernel Hilbert Space (RKHS) } F \]

with feature map \( \phi : D \rightarrow F \). Mean distance is measured by computing the Maximum Mean Discrepancy (MMD)

\[
\| E_{x \sim p_{\text{te}}(x)}[\beta(x)\phi(x)] - E_{x \sim p_{\text{tr}}(x)}[\phi(x)] \|
\]

(1)

where \( \| \cdot \| \) is the \( L_2 \) norm, and \( x \in X \subseteq D \) is a data instance in a dataset \( X \). Here, it is assumed that \( p_{\text{te}}(\cdot) \) is absolutely continuous with respect to \( p_{\text{tr}}(\cdot) \), i.e., \( p_{\text{tr}}(x) = 0 \) whenever \( p_{\text{te}}(x) = 0 \). Additionally, the RKHS kernel \( h \) is assumed to be universal in \( D \). It has been shown that under these conditions, minimizing MMD in Equation 1 converges to

\[ p_{\text{te}}(x) = \hat{\beta}(x)p_{\text{tr}}(x) \] [13].

In particular, minimizing MMD to obtain optimal importance weights is equivalent to minimizing the corresponding quadratic program that approximates the population expectation with an empirical expectation. The empirical approximation of MMD (Equation 1) to obtain the desired \( \beta(x) \) is given by

\[
\hat{\beta} \approx \arg \min_{\beta} \left\| \frac{1}{n_{\text{tr}}} \sum_{x \in X^{\text{tr}}} \beta(x)\phi(x) - \frac{1}{n_{\text{te}}} \sum_{x \in X^{\text{te}}} \phi(x) \right\|^2
\]

(2)

where \( \hat{\beta}(x) \in \hat{\beta} \), and \( X^{\text{tr}}, X^{\text{te}}, n_{\text{tr}}, n_{\text{te}} \) are training data covariates, test data covariates, size of the training data and size of test datasets respectively. The equivalent quadratic program is as follows.

\[
\hat{\beta} \approx \text{minimize} \frac{1}{2} \beta^T K \beta - \kappa^T \beta
\]

subject to \( \beta(x) \in [0, B], \forall x \in X^{\text{tr}} \)

and

\[
\frac{1}{n_{\text{tr}}} \sum_{x \in X^{\text{tr}}} \beta(x) - 1 \leq \epsilon
\]

where \( K \) and \( \kappa \) are matrices of a RKHS kernel \( h(\cdot) \) with \( K_{ij} = h(X_i^{\text{tr}}, X_j^{\text{tr}}) \in K \), and \( \kappa_i = \frac{n_{\text{te}}}{n_{\text{tr}}} \sum_{j=1}^{n_{\text{te}}} h(X_i^{\text{tr}}, X_j^{\text{te}}) \in \kappa \). \( B > 0 \) is an upper bound on the solution search space, and \( \epsilon \) is the normalization error.

D. Apache Spark

Apache Spark [14] is an in-memory cluster-computing platform for data analytics. It allows the machines to cache data in the memory avoiding disk I/O, and reuses it in multiple MapReduce-like parallel operations. In-memory caching contribute to much faster computation by Spark compared to most MapReduce based platforms, e.g., Apache Hadoop.

The main abstraction of Spark is Resilient Distributed Dataset (RDD), which is a collection of objects partitioned across a set of machines. Spark defines two types of parallel operations on a RDD, i.e., transformations and actions. A transformation such as map, filter, join, etc., is a lazy operation on a RDD, which creates one or more new RDDs. On the contrary, an action such as aggregate, collect, count, etc., is an operation on RDD that materializes a value in a Spark program.

RDDs play a central role in the fault tolerance mechanism of Spark, which maintains transformation operations on each RDD as a lineage. These lineages are recorded as centralized metadata in the master node. Therefore, if a partition is lost, Spark applies the same transformation operations on the original RDD to rebuild just that partition.

III. PROPOSED APPROACH

Kernel Mean Matching (KMM) algorithm (discussed in Section II-C) is sequential in nature. As mentioned in [15], it has time complexity of \( O(n_{\text{tr}}^3 + n_{\text{te}}^2 d + n_{\text{tr}} n_{\text{te}} d) \), where \( d \) is the number of dimensions of dataset. In real-world applications, especially in data streams, high speed data keeps entering into the system in large volume. In such scenarios, the classifier needs to be updated regularly to cope with any change of class boundaries known as a concept drift [16]. Often the size of data using which the classifier needs to be updated, are very large with possible sampling bias between training and test distributions. KMM can be used in such scenarios for updating the classifier with sampling bias correction. However, it adds a bottleneck in the periodic update process due to its limited scalability.

In this paper, we use the principles of KMM to present a sampling based distributed and parallel algorithm for efficient estimation of density ratios. We refer to this algorithm by Sampling based KMM or SKMM.

A. Sampling based KMM (SKMM)

Given an i.i.d. set of training covariates \( X^{\text{tr}} \) and an i.i.d. set of test covariates \( X^{\text{te}} \), such that \( X^{\text{tr}} \) is sufficiently large, the problem is to efficiently estimate density ratio or instance weight \( \beta(x) = \frac{p_{\text{te}}(x)}{p_{\text{tr}}(x)} \) for each \( x \in X^{\text{tr}} \) using the Kernel Mean Matching (KMM) method.
Since the main contributing factor to the high time complexity of KMM comes from \( n_{tr} \), the challenge of limited scalability of KMM can be addressed naively by splitting the training data into smaller subsets, and applying KMM over each subset independently. Union of density ratio estimates from all the samples provides instance weight for each training data instance. However, such a method may not perform well since a small subset of training data instances (chosen uniformly at random) may exhibit a glaringly different distribution compared to the original training data distribution. This can adversely affect the KMM output [13].

The estimation of \( \beta(x) \in \hat{\beta} \) is sensitive to the training data distribution, i.e., the estimates may vary depending on the size and choice of instances used as training data. Bootstrap methods [17] have been shown to be extremely useful when estimators are unstable. In this scenario, one can employ a bootstrap sampling process by generating samples with replacement from the given training data. However, a naive bootstrap sample from the training data will consist of \( n_{tr} \) instances. This does not aid in improving the computational time efficiency of KMM as desired. Therefore, the \( m \)-out-of-\( n \) bootstrap sampling (or \( m/n \) bootstrap) method is more appropriate since \( m < n_{tr} \) can be fixed. Here, \( m \) is the sample size and \( n = n_{tr} \). We utilize this notion to achieve scalability for sampling bias correction.

The time complexity of KMM is linear with respect to the size of the test data, i.e., \( n_{te} \). If \( n_{te} \) is small, KMM applied on the whole test data along with each sample from the training dataset, estimates weights for corresponding instances in the training sample. If sufficiently large number of samples are considered, union of weights from all the samples provide instance weight for each instance in the training set. On the contrary, if \( n_{te} \) is also large, which is a regular scenario in data streams, complete test data cannot fit into the memory. In such cases, KMM cannot be applied directly using the whole test data. One can address this challenge by sampling from test data also. However, sampling over the test dataset only approximates its data distribution. Moreover, a method that partitions the test data and applies KMM on each partition independently, has been demonstrated to achieve better performance [15]. Therefore, instead of sampling, SKMM divides test data into \( k \) partitions, where \( k \) is specified by the user.

Figure 1 illustrates this many-to-many computation scheme using training samples and test partitions. Components are formed by taking each possible pair of training sample and test partition. Since SKMM takes \( s \) samples from the training data, and creates \( k \) partitions in test data, the number of components is \( s \times k \). Applying KMM to each of the components produces estimated weights for all the instances in the corresponding training sample. Final weight for an instance is calculated by taking the average of all weights calculated for that instance from different components.

**Algorithm 1 SKMM \((X^{tr}, X^{te}, k, \eta, \theta)\)**

**Input:** \( X^{tr} \): Training data covariates; \( X^{te} \): Test data covariates; \( k \): Number of test data partitions; \( \eta \): Sampling error tolerance; \( \theta \): KMM parameters.

**Output:** \( \hat{\beta} \): Estimated weights for training instances.

```
1: m ← size(X^{tr})
2: s ← ⌈ln \frac{n_{tr}}{m ln (1 - \eta)}⌉ // Number of training samples
3: S ← genSample(X^{tr}, m, s)
4: P ← partition(X^{te}, k)
5: C ← cartesian(S, P) // Formation of Components
6: \( \hat{\beta} \leftarrow \text{zeros} \)
7: for \( \forall C_{ij} \in C, i \leftarrow 1 \ldots s \text{ and } j \leftarrow 1 \ldots k \) do
8: \( \hat{\beta}_{ij} \leftarrow \text{KMM}(C_{ij}, \theta) \)
9: \( \beta \leftarrow \text{aggregate}(\hat{\beta}_{ij}) \)
10: end for
11: Return \( \hat{\beta} = \left\{ \frac{\hat{\beta}(x)}{\text{count}(x)} : \forall x \in \bigcup_{i=1}^{s} S_{i} \right\} \)
```

Algorithm 1 sketches the sampling based approach. First, SKMM takes samples from the training data with replacement. Size of each sample is calculated by \( m = \frac{\text{size}(X^{tr})}{k} \), where \( k \) is the number of partitions in test data, which is a user input. Ideally, union of all the samples should contain each instance from the training data at least once. Since sampling is done randomly with replacement, inclusion of each training instance in the sampling process cannot be guaranteed. However, if a sufficiently large number of samples are taken, one can be highly confident that each instance from the training data will be selected at least once in the sampling process. We denote this confidence as \((1 - \eta)\), where \( \eta \) is the sampling error tolerance. The minimum number of samples \( s \) to be generated can be calculated using the following Lemma.

**Lemma 1.** Let \( s \) be the number of training samples generated from \( X^{tr} \) in SKMM, where each sample \( S_{i}, i \leftarrow 1 \ldots s \), consists of \( m \) number of instances selected randomly with replacement from training data. The minimum number of
samples required to be generated such that an instance \( x \in \mathbf{X}^{tr} \) belongs to the set \( \bigcup_{i=1}^{s} \mathbf{S}_i \) with probability at least \((1 - \eta)\) is given by

\[
\ln \eta \over mn \left(1 - \frac{1}{n_{tr}}\right)^m.
\]

Proof: Probability that a data instance \( x \in \mathbf{X}^{tr} \) is not selected in any of the \( s \) independent samples, each having \( m \) independent trials, is \( \left(1 - \frac{1}{n_{tr}}\right)^{ms} \). Using the definition, \( \eta \leq \left(1 - \frac{1}{n_{tr}}\right)^m \). Therefore, \( s \geq \frac{\ln \eta }{m \ln \left(1 - \frac{1}{n_{tr}}\right)} \). ■

SKMM also splits the test data into \( k \) partitions. Let \( \mathbf{P} \) be the set of test data partitions, where \( \mathbf{P}_j, j \leftarrow 1 \ldots k \), denotes the \( j^{th} \) test partition consisting of \( n_{te}^{ms} \) test instances. Next, SKMM takes a cartesian product of set \( \mathbf{S} \) and \( \mathbf{P} \), to pair each sample from the training set with each partition of the test data. We refer to each pair as a train-test component, or simply as a component. Let \( \mathbf{C} \) be the set of such components, where \( \mathbf{C}_{ij} \) is one of the components in set \( \mathbf{C} \) that consists of the training sample \( \mathbf{S}_i \) and test partition \( \mathbf{P}_j \). SKMM then applies kernel mean matching algorithm on each of the components \( \mathbf{C}_{ij} \in \mathbf{C} \) to calculate weights for each instance \( x \in \mathbf{S}_i \). In this process, multiple weights may be calculated for training instances that are selected in multiple samples. Therefore, SKMM aggregates instance-wise weights denoted by \( \hat{\beta}(x) \) for all \( x \) selected in the sampling process, i.e., \( \bigcup_{i=1}^{s} \mathbf{S}_i \). Finally, SKMM outputs \( \hat{\beta}(x) \in \hat{\beta} \), density ratio or importance weight for instance \( x \), by dividing aggregated \( \hat{\beta}(x) \) with the number of times \( x \) is selected in \( \bigcup_{i=1}^{s} \mathbf{S}_i \).

B. Sampling based Distributed KMM (SDKMM)

Next propose the distributed version of SKMM, referred to as Sampling based Distributed KMM (SDKMM).

\begin{algorithm}
\caption{SDKMM (\( \mathbf{X}^{tr}, \mathbf{X}^{te}, k, \eta, \theta \))}
\begin{algorithmic}[1]
\State \textbf{Input:} \( \mathbf{X}^{tr} \): Training data covariates; \( \mathbf{X}^{te} \): Test data covariates; \( k \): Number of test data partitions; \( \eta \): Sampling error tolerance; \( \theta \): KMM parameters.
\State \textbf{Output:} \( \hat{\beta} \): Estimated weights for training instances.
\State \hspace{1cm} 1: \( m \leftarrow \frac{\text{size}(\mathbf{X}^{tr})}{k} \)
\State \hspace{1cm} 2: \( s \leftarrow \left\lceil \frac{\ln \eta}{m \ln \left(1 - \frac{1}{n_{tr}}\right)} \right\rceil \) \hspace{0.5cm} // Number of training samples
\State \hspace{1cm} 3: \( \mathbf{S} \leftarrow \text{genSample}(\mathbf{X}^{tr}, m, s) \)
\State \hspace{1cm} 4: \( \mathbf{P} \leftarrow \text{partition}(\mathbf{X}^{te}, k) \)
\State \hspace{1cm} 5: \( \mathbf{C} \leftarrow \text{cartesian}(\mathbf{S}, \mathbf{P}) \) \hspace{0.5cm} // Formation of Components
\State \hspace{1cm} 6: Master node distributes components over worker nodes.
\State \hspace{1cm} 7: \textbf{MAP:}
\State \hspace{1cm} \hspace{1cm} 8: Each worker node calculates instance weights by applying KMM on each component \( \mathbf{C}_{ij} \) (consists of \( \mathbf{S}_i \) and \( \mathbf{P}_j \)) received.
\State \hspace{1cm} \hspace{1cm} 9: Emit \( \langle \text{ind}(x), \hat{\beta}_i(x) \rangle, \forall x \in \mathbf{S}_i \).
\State \hspace{1cm} \hspace{1cm} 10: \textbf{REDUCE:}
\State \hspace{1cm} \hspace{1cm} 11: Calculate \( \langle \langle \text{ind}(x) \rangle, \sum \hat{\beta}(x), \text{count}(x) \rangle, \forall x \in \bigcup_{i=1}^{s} \mathbf{S}_i \).
\State \hspace{1cm} \hspace{1cm} 12: Emit \( \langle \text{ind}(x), \hat{\beta}(x) = \frac{\sum \hat{\beta}_i(x)}{\text{count}(x)} \rangle, \forall x \in \bigcup_{i=1}^{s} \mathbf{S}_i \).
\State \hspace{1cm} \hspace{1cm} 13: \textbf{Return} \( \hat{\beta} = \left\{ \hat{\beta}(x) : \forall x \in \bigcup_{i=1}^{s} \mathbf{S}_i \right\} \)
\end{algorithmic}
\end{algorithm}

Figure 2 shows the workflow of SDKMM. Following the creation of components from training samples and test partitions, SDKMM distributes the components to different Mappers. Ideally, each Map function should be invoked only on one component \( \mathbf{C}_{ij} \) for calculating weights of instances in \( \mathbf{S}_i \). All the weights from Mappers are then aggregated instance-wise by a Reducer. Finally, the reducer divides the aggregated weight for each instance by the number of times that instance is selected in the sampling process to calculate the final estimated weight for that instance.

Algorithm 2 details the proposed approach SDKMM. First, it calculates size of each training samples and number of such samples at Lines 1-2 according to Lemma 1. Then, it takes \( s \) number of samples, each consisting of \( m \) instances selected randomly with replacement from the training data at Line 3. It also splits the test data into \( k \) partitions, and create \( s \times k \) components by considering each possible pair of training sample and test partition at Lines 4-5. Next, the components are distributed over the Mappers (worker nodes) for applying KMM on these in parallel. For each component \( \mathbf{C}_{ij} \), Map function applies KMM on training sample \( \mathbf{S}_i \) and test partition \( \mathbf{P}_j \). The Map function provides output in \( \langle \text{key}, \text{value} \rangle \) pairs for all \( x \in \mathbf{S}_i \), where key is the index of \( x \) and value is the estimated weight for \( x \) from \( \mathbf{C}_{ij} \), denoted by \( \hat{\beta}_{ij}(x) \).

All these estimated weights for different \( x \in \mathbf{S}_i \) are
received by the Reducer (worker node), where $\bigcup_{i=1}^{s} S_i$ denotes the set of all training instances which are selected at least once in the sampling process. Since the instances are selected randomly with replacement, an instance can be selected multiple times in the sampling process. Therefore, the Reduce function first aggregates all these estimated weights instance-wise denoted by $\sum \hat{\beta}(x)$. To get the final estimated weight of a training instance $x \in S_i$, which is denoted by $\hat{\beta}(x)$, $\sum \hat{\beta}(x)$ is divided by the number of times $x$ is selected in the sampling process (denoted by $\text{count}(x)$).

C. Challenges and Design Choices

In this section, we discuss about some of the design challenges that exist in implementing Algorithm 2, and choices available to address these challenges. First, the main objective of SDKMM is to reduce execution time for addressing sampling bias. As discussed in Section II-D, Apache Spark is much faster than other existing cluster computing framework due to in-memory caching ability. Therefore, we use Spark to implement SDKMM.

Second, as shown in Algorithm 2, SDKMM creates components in the master node by taking every possible combination of training samples and test partitions, and then distributes these components among the worker nodes for parallel computations. SDKMM is more efficient when large number of small-size samples are taken from the training dataset, because the samples can be processed quickly in parallel due to small size of each sample. At the same time, more samples help to estimate the distribution of the training data more accurately as discussed in Section III-A. However, taking large number of samples from training data also increases number of components to a great extent. Often we observe that total size of components exceeds size of the main memory in the master machine. Therefore, instead of full data instances, we store only the indices of training and test data instances in a component. This greatly reduces the total size of the components.

Third, since only indices of instances are stored in the components, worker machines need access to the original training and test data to form the actual component before applying KMM. We share the original training and test data among the worker nodes using Spark Broadcast API before actual Map begins.

Finally, it is desired that only one component be processed per invocation of Map function. Spark automatically sets the number of Map tasks based on the number of slices in RDD (Resilient Distributed Dataset) depending on its size. However, SDKMM greatly reduces the size of RDD containing components by replacing actual data instances by indices. Therefore, using default setting does not satisfy the objective of invoking one Map per component. To fully utilize the cluster, we set the minimum between the number of cores and the number of components as number of slices, i.e., number of Map tasks.

D. Complexity Analysis

As mentioned in [15], the original sequential KMM approach has a time complexity of $O(n_{tr}^2 + n_{te}^2 + n_{tr} n_{te} d)$. Our proposed approach SKMM creates components by pairing a training sample of size $m$, and a test partition of size $\frac{m}{k}$. Therefore, time complexity of estimating weights for all training instances in a component is $O(m^3 + m^2 d + m \frac{m}{k} d)$. In case of SDKMM, time complexity to process all the components remains the same as to process a single component due to parallel processing. Finally, the aggregation of weight estimations requires $O(m)$. Together, the time complexity of SDKMM is $O(m^3 + m^2 d + m \frac{m}{k} d) + m$. Clearly, SDKMM achieves quicker execution time as $k$ increases, i.e., $m$ decreases. Similarly, the space complexity of sequential KMM is $O((n_{tr})^2 + n_{tr} n_{te})$, whereas that of SDKMM is $O(m^2 + m \frac{m}{k} + n_{te})$.

IV. Evaluation

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Features</th>
<th>Total Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForestCover</td>
<td>54</td>
<td>50,000</td>
</tr>
<tr>
<td>KDD</td>
<td>34</td>
<td>50,000</td>
</tr>
<tr>
<td>PAMAP</td>
<td>53</td>
<td>50,000</td>
</tr>
<tr>
<td>PowerSupply</td>
<td>2</td>
<td>29,928</td>
</tr>
<tr>
<td>SEA</td>
<td>3</td>
<td>50,000</td>
</tr>
<tr>
<td>Syn002</td>
<td>70</td>
<td>50,000</td>
</tr>
<tr>
<td>Syn003</td>
<td>70</td>
<td>50,000</td>
</tr>
<tr>
<td>MNIST</td>
<td>780</td>
<td>50,000</td>
</tr>
</tbody>
</table>

Table II: List of datasets

A. Dataset

Table II lists the datasets used in the experiments. We use two synthetic datasets, Syn002 and Syn003, which are generated using MOA [18]. Others are real-world datasets, and publicly available [19], [20]. Since the execution time of centralized KMM increases greatly with increasing size of the dataset, we consider the first 50,000 instances from each dataset in our experiments. In order to simulate sampling bias between the training and test data, we follow a procedure similar to a previous study [6]. For each dataset, we first compute the covariate mean $\bar{X}$ of all data instances, and select $n_{te}$ data instances with probability of $p(\xi = 1|X_i) = \exp\left(-\frac{\|X_i - \bar{X}\|^2}{2\sigma^2}\right)$, where $\xi$ is an indicator variable with 1 indicating selection of $X_i$ as a training instance, and $\sigma$ is the standard deviation of $\|X_i - \bar{X}\|$, $\forall X_i \in X$. Remaining part of the dataset is considered for testing.

B. Baseline Methods

We use two baseline methods to compare performance with our proposed approach SDKMM. The first baseline approach is the original centralized Kernel Mean Matching (KMM), which uses the whole training and test data for density ratio estimation. We denote this approach as CenKMM.
The second baseline method is denoted as EnsKMM, which is proposed by Miao et al. [15]. In this approach, first the test data instances are divided into $k$ partitions. Since $\beta(x) \propto p_{te}(x)$, an ensemble of estimators is then obtained in EnsKMM, where each estimator estimates weights for all the training instances based on one of the test partitions and the whole training data. Finally, the estimates from individual estimators are combined to form $\hat{\beta} = \frac{1}{k} \sum_{i=1}^{k} \hat{\beta}_i$. 

Figure 3: Logarithm of NMSE with increasing size of training set ($n_{tr}$)

Figure 4: Logarithm of NMSE with increasing $k$
where \( \hat{\beta}_i \) is the set of estimated weights from \( i^{th} \) estimator. While the study demonstrates improvements in accuracy and execution time, computational efficiency is still limited by requiring the complete training dataset in the memory. Since computations on individual estimators can be done in parallel, we implement EnsKMM also using Apache Spark for a fair comparison with the proposed approach.

C. Setup

We implemented all the approaches considered in this paper using Python version 2.7.5. We used the well-known QP solver in CVXOPT python library [21] to execute the KMM quadratic program, with \( B = 1000 \) and \( \epsilon = \frac{\sqrt{n_{tr}} - 1}{\sqrt{n_{tr}}} \). Following [6], we use a Gaussian kernel with width \( \gamma \) equal to the median of pairwise distances. All the experiments related to SDKMM and EnsKMM were performed on a cluster running Spark version 1.5.1. The cluster has 12 nodes, each with eight 2.40 GHz cores and 16 GB of main memory. In the experiments, we have used \( n_{tr} = 1000, k = 10, \) and \( \eta = 0.01 \) as default settings if not mentioned otherwise.

D. Normalized Mean Square Error (NMSE)

In the first set of experiments, we compare goodness of estimated importance weights by different approaches mentioned in Section IV-B. We measure goodness of estimated weights (denoted as \( \hat{\beta}(\mathbf{x}) \in \hat{\beta} \)) by Normalized Mean Square Error (NMSE) defined as
\[
\frac{1}{n} \sum_{i=1}^{n} \left( \frac{\hat{\beta}(\mathbf{x}_i) - \beta(\mathbf{x}_i)}{\sum_{j=1}^{p(\{1:|\mathbf{x}_i|\})} \hat{\beta}(\mathbf{x}_j)} \right),
\]
where \( \hat{\beta}(\mathbf{x}_i) = \frac{1}{p(\{1:|\mathbf{x}_i|\})} \), following [15]. Figure 3 shows NMSE of all the approaches with increasing size of the training set \( (n_{tr}) \). In the experiments, we report natural logarithm of NMSE value for ease of interpretation. It can be observed that all the approaches show lower MSE score with increasing size of the training set as expected. The proposed approach SDKMM shows very competitive performance if not better in terms of weight estimation accuracy compared with CenKMM and EnsKMM.

In the next experiment, we vary the number of partitions in the test data \( (k) \), and observe the effect on the quality of weight estimation. As discussed in Section III-A, the size of each training sample \( m \) is inversely proportional to the value of \( k \). Moreover, the number of training samples \( s \) is inversely proportional to the value of \( m \). In other words, increasing value of \( k \) results into decreasing \( m \) and increasing \( s \), and vice versa. Therefore, it is expected that NMSE should decrease with increasing \( k \) in case of SDKMM, due to a better estimation of the training data distribution by more bootstrap samples from training data. This is evident from Figure 4, which show that SDKMM outperforms the other approaches in most of the cases with increasing value of \( k \).

E. Execution Time

Total execution time of SDKMM along with baseline approaches with increasing size of training data set is shown in Figure 5. As discussed in Section III-D, both CenKMM and EnsKMM have cubic time complexity with respect to the size of the training data \( (n_{tr}) \). However, in SDKMM, components are formed by taking samples from training data, and by partitioning test data. Moreover, each partition is processed in parallel. Therefore, SDKMM should have the best performance in terms of execution time among all the approaches considered, which is also evident from Figure 5. We observe that with increasing \( n_{tr} \), time required for estimating instance weights remain almost same in case of SDKMM due to distributed and parallel execution of components. On the contrary, time increases rapidly in case of CenKMM and EnsKMM. This is significant since in data streaming scenario, \( n_{tr} \) can be extremely large due to high speed continuous data entering into the system. SDKMM can be employed in these scenarios.

Figure 6 shows total time consumed by different approaches with increasing value of \( k \) with \( n_{tr} = 500 \). As discussed before, size of each training sample \( (m) \) and the number of training samples \( (s) \) are inversely proportional and proportional respectively with respect to \( k \). Consequently, execution time decreases in general with increasing \( k \) due to distributed processing of smaller sized samples in case of both SDKMM and EnsKMM. However, SDKMM requires much lower time than EnsKMM due to sampling from the training data besides partitioning the test data. Both EnsKMM and SDKMM require lower time than CenKMM.

F. Speed up

We compare speed up achieved by SDKMM and EnsKMM on different datasets in Figure 7. We define speed up by \( \frac{T_{sq}}{T_d} \), where \( T_{sq} \) and \( T_d \) are the execution time of sequential and distributed approach respectively on a given set of training and test data. It is clear from the plots that SDKMM achieves much more speed up compared to EnsKMM. More importantly, with increasing \( n_{tr} \), speed up of SDKMM increases rapidly, whereas EnsKMM shows only a limited speed up. As evident from Figure 3, larger \( n_{tr} \) also results in better estimation accuracy. Therefore, both Figure 5 and Figure 7 suggest that if more training data are provided, SDKMM provides much better estimates while having similar execution time. We skip identical results on some of the datasets in Figure 6 and 7 due to space limitation.

G. Sensitivity

Figure 8 shows sensitivity of SDKMM to the parameter \( \eta \). We observe from Figure 8a that NMSE increases in general with increasing \( \eta \). On the contrary, Figure 8b indicates that time in general decreases with increasing \( \eta \). Increasing \( \eta \) results in lower number of larger size samples. Therefore,
increasing $\eta$ affects inherent approximation of the training data distribution adversely, which results in increasing NMSE. Moreover, less number of samples results in slightly less execution time. In both cases, we observe that NMSE changes slowly with increasing $\eta$, indicating that SDKMM is not significantly sensitive to $\eta$. 

Figure 5: Total execution time in seconds with increasing size of training set ($n_{tr}$)

Figure 6: Total execution time in Seconds with increasing $k$

Figure 7: Speed up with increasing size of training set ($n_{tr}$)
We have proposed a sampling based distributed and parallel approach in this paper that computes density ratios between training and test data distributions efficiently using Kernel Mean Matching (KMM) algorithm. These density ratios can be used for sampling bias correction in data. Experiment results indicate that the proposed approach is effective in addressing the limited scalability problem in the original KMM algorithm. Therefore, it can be very useful to address sampling bias problem in real-world data mining applications with large datasets.

**REFERENCES**


