Supervised Learning by Training on Aggregate Outputs

David R. Musicant, Janara M. Christensen, Jamie F. Olson
Department of Computer Science
Carleton College
Northfield, MN
dmusican@carleton.edu, christej@carleton.edu, jamie.f.olson@alumni.carleton.edu

Abstract

Supervised learning is a classic data mining problem where one wishes to be be able to predict an output value associated with a particular input vector. We present a new twist on this classic problem where, instead of having the training set contain an individual output value for each input vector, the output values in the training set are only given in aggregate over a number of input vectors. This new problem arose from a particular need in learning on mass spectrometry data, but could easily apply to situations when data has been aggregated in order to maintain privacy. We provide a formal description of this new problem for both classification and regression. We then examine how k-nearest neighbor, neural networks, and support vector machines can be adapted for this problem.

1. Introduction

Supervised learning is a classic data mining problem where one wishes to be able to predict an *output value* associated with a particular *input vector*. A predictor is constructed via the use of a *training set*, which consists of a set of input vectors with a corresponding output value for each. This predictor can then be used to predict output values for future input vectors where the output values are unknown. If the output data is continuous, this task is referred to as *regression*. If the output data is nominal, this task is referred to as *classification* [14].

An important application that we are currently engaged in is the analysis of single particle mass spectrometry (SPMS) data. A typical SPMS dataset consists of a series of mass spectra, each one of which is associated with an aerosol particle. One important goal to the SPMS community is to be able to use the mass spectrum associated with an individual aerosol particle to be able to predict some other quantity associated with that particle. For example, we are interested in being able to identify the quantity of *black carbon* (BC) associated with an individual aerosol particle. While SPMS data provides detailed mass spectra for individual aerosol particles, these spectra are not quantitative enough to allow one to precisely identify the amount of BC present in a particle. On the other hand, filter-based machines can collect quantitative BC data, but in aggregate form. These devices allow one to determine how much BC has been observed in aggregate over a particular time span, typically on the order of an hour. SPMS data, on the other hand, may be collected for individual particles (non-aggregate data) many times per minute. Our goal is to be able to predict the quantity of BC present in an individual aerosol particle. Therefore, the mass spectra associated with individual particles become our input vectors, and the quantities of BC become our output values. This would appear to be a traditional regression problem, but there is a key difference. The challenge that we face is that we do not have a single output value for each input vector. Instead, each output value in our training set represents the sum of the true output values (which are unknown to us) for a series of input vectors in our training set. We wish to learn a predictor that will still produce individual output values for each new input vector, even though the training set only contains these output values in aggregate.

We therefore present a new supervised learning problem, which we refer to as the *aggregate output learning problem*. We develop both a classification and a regression version of it, and show how a number of popular supervised learning algorithms can be adapted to build predictors on data of this sort. While we arrived at the need for this algorithm from a particular scientific application, this framework would easily apply to training data which has been aggregated for purposes of privacy preservation. We should point out that traditional algorithms could be used on this data if the input vectors were aggregated to the same granularity as the output values, but this ignores useful individual data that could be used to improve the quality of the predictor.

There are three key contributions of this paper. First, we present a formal framework for this new machine learning problem. Second, we adapt three classic algorithms, namely k-nearest neighbor, neural networks, and support vector machines, for use under this scenario and show that they perform the task well. Finally, as this new machine learning problem does not seem to have been considered before, this paper opens up research opportunities for the adaptation of other popular algorithms or invention of new ones to help solve this problem.

In the next section, we examine previously related work on this subject. We then follow it by precisely defining the aggregate output learning problem, and then present the details for how a number of standard supervised learning algorithms can be adapted to work for this problem. Finally, we present experiments on a number of datasets to show the efficacy of our approach.

Age	Income	Weight	Savings?
50	75000	220	
30	56000	180	1,900,000
50	60000	170	
48	40000	150	
22	45000	160	
25	50000	180	400,000
23	38000	165	
24	61000	190	

Age	Income	Weight	Savings?
40	48000	170	?
29	60000	180	?
57	18000	195	?

Table 1. Sample regression training and test sets. In the training set, output values are known only in aggregate.

2. Previous Work

The aggregate output learning problem certainly bears some similarities with other well-known learning problems, but has significant differences as well. The unsupervised learning problem has no output values at all, whereas the supervised learning problem has a distinct output value for each individual input vector in the training set [9, 18, 20]. Semi-supervised learning [2, 3] describes a scenario somewhat between the two, where some of the input vectors have associated output values (as in supervised learning), but others do not (as in unsupervised learning). This is rather distinct from our aggregate output learning problem where *all* input vectors are associated with an output value, but multiple input vectors map to the same output value which represents the sum of the actual (unknown) output values. Moreover, our problem is different from the others described above in that the granularity of the output values is different in the training set than in the test set. In the training set, output values are aggregated over multiple input vectors. In the test set, each input vector has its own output value.

Other forms of aggregate learning seem to have been examined, though they differ from the form we discuss here. Arminger et. al. propose using log-linear models to disaggregate data with a similar model to ours, but their framework is based on the idea of filling in missing data in the training set. Their approach does not seem to generalize easily to a test set [1]. McGrogan et. al. consider an approach where the training set contains an individual output value for each input vector, but this output value is an aggregate over multiple measurements [13]. Yang et. al. look at learning with aggregates only in order to preserve privacy [21]. An approach by Chen et. al. examines learning from multiple aggregate tables each derived from an underlying common dataset [4]. This probabilistic approach is different from our scenario in this paper in that we do not assume that the input vectors within an individual aggregation collection share feature values. This last difference also contrasts our approach with a number of ideas from the statistical literature such as iterative proportional fitting [12].

We now move on to provide a formal framework for the problems that we consider.

3. Problem Formulations

3.1. Regression

Table 1 shows a sample dataset for this framework. Suppose that we are given a training set that consists of input vectors $\{\mathbf{x}_{ci}\}$, where each $\mathbf{x}_{ci}$ has an *unknown* real-valued output value y_{ci}. We are given, however, a set of aggregate output values y_c, where we know that for a fixed c, $y_c = \sum_i y_{ci}$. (The subscript c indicates an aggregate collection, and the subscript i identifies a particular data point within that collection.) The goal is the same as for the traditional regression problem. We wish to learn a predictor f, using the training set only, that performs well on unseen test data drawn from the same source. In other words, for a test input vector $\mathbf{x}$ with output value y, we desire $f(\mathbf{x}) \approx y$. (The precise criterion varies with the learning algorithm.) Note that f operates on a single input vector and produces a single output value, though the training set contains output values aggregated over multiple input vectors. The test set contains unaggregated output values.

3.2. Classification

Though regression is perhaps the more natural approach for thinking about the aggregate output learning problem, we present a classification version of it as well.

First, we point out that the traditional classification problem, in general, allows each input vector to belong to one of

Age	Income	Weight	Beer over Wine?
50	75000	220	
30	56000	180	3 Yes
50	60000	170	1 No
19	2000	150	
32	60000	160	
35	90000	180	1 Yes
60	85000	165	3 No
53	92000	190	

Age	Income	Weight	Beer over Wine?
40	48000	170	?
29	60000	180	?
57	18000	195	?

Table 2. Sample classification training and test sets. In the training set, classifications are known only in aggregate.

an arbitrary number of classes. We constrain ourselves in this paper to the binary classification problem, where each input vector belongs to one of two possible classes.

Table 2 shows a sample dataset for our framework. Suppose that we are given a training set of input vectors where each has an *unknown* output value (also known as the class label). This output value is a "yes" or a "no," depending on to which class its corresponding input vector belongs. The training set is divided into collections of input vectors where aggregate output values are known for each collection. More precisely, we suppose that our training set consists of input vectors $\mathbf{x}_{ci}$, where the subscript c indicates to which collection the input vector belongs, and the subscript i identifies a particular input vector within that collection. We further suppose that we are given a set of aggregate output values y_c and $\bar{y}_c$, where for an individual collection c, y_c is the number of "yes" values and $\bar{y}_c$ is the number of "no" values. The goal is the same as for the traditional classification problem. We wish to learn a predictor f, using the training set only, that performs well on unseen test data presumably drawn from the same source. In other words, for a set of test input vectors $\{\mathbf{x}_j\}$ with unknown output values $\{y_j\}$, we desire $f(\mathbf{x}_j) = y_j$ often. (The precise definition of "often" varies with the learning algorithm.) Note that f operates on a single input vector and produces a single output value, though the training set contains output values aggregated over multiple input vectors.

4. Algorithm Updates

We now describe in detail how three popular supervised learning techniques can be updated for the aggregate output learning problem, for classification and regression scenarios. k-nearest neighbor is an exceedingly simple algorithm to use, and its adaptation is similarly straightforward. Support vector machines require the formulation of a quadratic programming optimization problem, and so we present new reformulations of these optimization problems to address our scenario. Neural networks traditionally use the backpropagation algorithm [14, 17] to find the local minimum for a quadratic loss optimization problem. We present modifications to the classic backpropagation algorithms to handle our new problem, and similarly show how radial basis networks can be adapted for the regression problem.

In each case, we present classification first, then regression. This is for purposes of readability: for most algorithms, the classification approaches are more well known. However, for our particular application (described in Section 1), the regression problem is more critical.

It should be noted that we make no claims that these new approaches are optimal techniques for solving the aggregate output learning problem. We believe that one of the main contributions of this paper is the formulation of this new problem for the community as well as for our particular application, and therefore we present a first effort at solving this problem via modifications to traditional algorithms.

4.1. k-Nearest Neighbor: Classification

The k-nearest neighbor algorithm is a remarkably simple yet effective approach for classification [9, 18, 20]. The algorithm requires the choice of a parameter k, representing the number of neighbors to be used, and a distance metric for quantifying the difference between input examples. In the traditional supervised learning approach, for a particular test input vector its k nearest neighbors in the training set are determined. As each of these nearest neighbors already has an output value, the test point is assigned the output value represented by a majority of the nearest neighbors.

Our adaptation of k-nearest neighbor, in both the classification and regression case, is exceedingly straightforward and perhaps obvious. We present it here as an instructive example to help reinforce our general approach before addressing the more complex algorithms. We also present it for experimental comparison purposes.

For the classification version of the aggregate output learning problem, for a particular test input vector one can find a set of k nearest neighbors in the training set in precisely the same way as one does for the traditional algorithm. The difference is that each nearest neighbor $\mathbf{x}_{ci}$ belongs to a set of training input vectors for which only an aggregate y_c and $\bar{y}_c$ is known. We resolve this by creating an artificial output value for each nearest neighbor $\mathbf{x}_{ci}$, which is defined as the proportion of "yes" classifications for the entire collection to which it belongs. In other words, if we define n_c to be the number of input vectors $\mathbf{x}_{ci}$ in aggregate collection c, we define for each input vector $\mathbf{x}_{ci}$ an artificial output value of $\tilde{y}_{ci} = \frac{y_c}{n_c}$. We then average $\tilde{y}_{ci}$ over all k nearest neighbors to produce our estimated y for our test point. If this is greater than one-half, we classify the test point as a "yes"; otherwise, we classify it as a "no." Note that if one has knowledge that leads one to believe that the test set has a different ratio of "yes" to "no" than the training set does, one can change the threshold accordingly.

This approach is mathematically equivalent to thinking of "yes" as a 1, "no" as a 0, and applying the regression technique described below.

4.2. k-Nearest Neighbor: Regression

The k-nearest neighbor algorithm can also be used for traditional regression problems. Instead of using a majority rule amongst the neighbors, the output values from all nearest neighbors are averaged together.

Adapting the regression form of k-nearest neighbor for our aggregate output learning problem is quite similar to the adaptation we do for classification. Again, we point out that k-nearest neighbor is a particularly simplistic approach: we demonstrate this for instructional purposes before proceeding to more complex algorithms. For a particular test input vector one can find a set of k nearest neighbors in the training set in precisely the same way as one does for the traditional algorithm. The difference is that each nearest neighbor $\mathbf{x}_{ci}$ belongs to a set of training input vectors for which only an aggregate y_c is known. We resolve this by creating an artificial output value for each nearest neighbor $\mathbf{x}_{ci}$, which is defined as the average of the output value across the entire aggregate collection to which it belongs. In other words, if we define n_c to be the number of input vectors $\mathbf{x}_{ci}$ in aggregate collection c, we define an artificial output value $\tilde{y}_{ci} = \frac{y_c}{n_c}$. We then average $\tilde{y}_{ci}$ over all k nearest neighbors to produce our estimated y for our test point.

This approach is mathematically equivalent to preprocessing the training set by assigning to each point an output value equal to the average output value for its aggregate collection, and proceeding with the traditional k-nearest neighbor algorithm. A considerable deficiency with using k-nearest neighbor in this way is that the algorithm does not allow the learner flexibility in distributing the aggregate output value for a collection unevenly throughout the points in that collection. This issue is addressed well, however, by our neural network and support vector machine approaches.

4.3. Neural Network: Classification

We first describe the traditional neural network for classification [14, 17] to establish our notation. We will be changing some of the notation that we used in the k-nearest neighbor case as we define the neural networks, as our primary goal is to make each algorithm as simple and understandable as possible. We are given an input vector $\mathbf{x}$, where x_i represents the i-th component of the vector $\mathbf{x}$. Each input dimension connects to a series of hidden nodes with weights w_{ij}. An activation function $g(\cdot)$, typically a sigmoid in practice, is used to process the output of each node. The output of each hidden node is denoted as a_j. Each hidden node, in turn, connects to a series of output nodes with weights v_{jk}. The output from each output node o_k is similarly preprocessed with the same activation function $g(\cdot)$. A sigmoid typically has a constant threshold parameter. We adopt the usual strategy of representing that parameter via an artificial input dimension of constant value [14, 17]. More precisely, the output for a particular output node o_k is determined as $o_k = g(\sum_j v_{jk}a_j)$, and the output for a particular hidden node a_j is determined as $a_j = g(\sum_i w_{ij}x_i)$.

For the aggregate output learning problem, we wish to retain this traditional neural network. The test set will consist of individual points, so a network of this form makes sense. The training procedure must differ, however, since the training set contains outputs for aggregate collections instead of for individual points. We therefore derive an update to the traditional backpropagation algorithm [14, 17].

The traditional backpropagation algorithm requires, for a particular input vector $\mathbf{x}$, the comparison of an output o_k with the actual output y_k. Neural networks allow for multiple outputs, which we have not addressed in our framework or other algorithms in this paper. Since including multiple outputs is straightforward for neural networks, we leave this possibility in as we work though our derivations. We therefore define y_k to be the k-th desired output. The key difference we face from the traditional approach is that we only have an output y_k for an aggregate collection that contains a number of input vectors. For a particular aggregate collection, we will use the notation $x_{\ell i}$ to represent the ℓ-th input vector's i-th component. Similarly, we represent the output of each hidden node for each input vector as $a_{\ell j} = g(\sum_i w_{ij}x_{\ell i})$, and output k for an entire aggregate collection as $o_k = \sum_\ell g(\sum_j v_{jk}a_{\ell j})$. This is similar to the traditional approach, but the presence of the summation over the subscripts ℓ requires us to develop modifications to backpropagation to handle this new learning problem.

In order to determine the optimal weights, we start with the output layer. For a particular aggregate collection, the error is defined as:

$$E = \tfrac{1}{2}\sum_k (y_k - \sum_\ell g(\sum_j v_{jk}a_{\ell j}))^2 = \tfrac{1}{2}\sum_k e_k^2 \quad (1)$$

where we define e_k to be the difference between the expected output and the actual. Denoting fixed indices by capital letters, we find the gradient by taking the partial derivative with respect to a particular weight v_{JK} as:

$$\begin{aligned}\frac{\partial E}{\partial v_{JK}} &= -e_K \sum_\ell \frac{\partial}{\partial v_{JK}} g(\sum_j v_{jK}a_{\ell j}) \\ &= -e_K \sum_\ell g'(\sum_j v_{jK}a_{\ell j})a_{\ell J}\end{aligned} \quad (2)$$

We therefore define the propagation update rule for v_{JK} as:

$$v_{JK} := v_{JK} + \alpha e_K \sum_\ell g'(\sum_j v_{jK} a_{\ell j}) a_{\ell J} \tag{3}$$

where α is a learning rate parameter [17]. Similarly, we can derive an update rule for each weight w_{IJ} as:

$$\begin{aligned}
&\tfrac{\partial E}{\partial w_{IJ}} = -\sum_k [e_k \sum_\ell \tfrac{\partial}{\partial w_{IJ}} g(\sum_j v_{jk} a_{\ell j})] \\
&= -\sum_k [e_k \sum_\ell g'(\sum_j v_{jk} a_{\ell j}) \tfrac{\partial}{\partial w_{IJ}} \sum_j v_{jk} a_{\ell j}] \\
&= -\sum_k [e_k \sum_\ell g'(\sum_j v_{jk} a_{\ell j}) \tfrac{\partial}{\partial w_{IJ}} \sum_j v_{jk} g(\sum_i w_{ij} x_{\ell i})] \\
&= -\sum_k [e_k \sum_\ell g'(\sum_j v_{jk} a_{\ell j}) \sum_j v_{jk} g'(\sum_i w_{ij} x_{\ell i}) \\
&\qquad\qquad \times \tfrac{\partial}{\partial w_{IJ}} \sum_i w_{ij} x_{\ell i}] \\
&= -\sum_k [e_k \sum_\ell g'(\sum_j v_{jk} a_{\ell j}) v_{Jk} g'(\sum_i w_{iJ} x_{\ell i}) x_{\ell I}]
\end{aligned} \tag{4}$$

We therefore define the propagation update rule for w_{IJ} as

$$\begin{aligned} w_{IJ} := w_{IJ} + \alpha \sum_k [e_k \sum_\ell g'(\sum_j v_{jk} a_{\ell j}) \\ \times \quad v_{Jk} g'(\sum_i w_{iJ} x_{\ell i}) x_{\ell I}] \end{aligned} \tag{5}$$

With these update rules, we proceed in a similar fashion to traditional neural network backpropagation. For each aggregate collection, we determine the output from the neural network for each point, aggregate, and compare with the expected output for that collection. The backpropagation update rules then indicate how to update the weights. We iterate over the dataset repeatedly until the error changes by less than a predefined threshold. The key difference between these update rules and the traditional ones is the appropriate usage of the summations over ℓ, which represent the multiple points in an aggregate collection.

We note that in contrast with the k-nearest neighbor approach described earlier, this approach does not require the total aggregate output for each collection to be averaged evenly across that collection in some fashion. Instead, this approach only attempts to constrain the total output across a collection to approximate that in the training set.

4.4. Neural Network: Regression

In order to use neural networks for regression, we adopt the radial basis function network approach [10]. In this scenario, under the traditional approach we are given an input vector $\mathbf{x}$, where x_i represents the i-th component of the vector $\mathbf{x}$. Each input dimension connects to a series of hidden nodes with weights w_{ij}. The output of each hidden node (denoted as a_j) is the distance between the input vector $\mathbf{x}$ and the associated weights $\mathbf{w_{\cdot j}}$, post-processed by a radial basis function. A linear combination of the outputs from the hidden nodes (with weights v_j) yields the output from the network. We adopt the usual strategy of representing the threshold term in this linear combination via an additional hidden node that always outputs a value of 1 [14].

More precisely, the output o associated with a particular input vector $\mathbf{x}$ is determined as $o = \sum_j v_j a_j$, where the output for a particular hidden node a_j is determined as

$$a_j = e^{\frac{-\sum_i (x_i - w_{ij})^2}{2\sigma^2}} \tag{6}$$

where σ is a parameter. For the aggregate output learning problem, we wish to use this same network structure. As in the classification case, the test set will consist of individual points, so a network of this form makes sense. The training procedure must be reexamined, however, since the training set contains outputs for aggregate collections instead of for individual points.

The traditional approach for training RBF networks is to first choose the weights for the hidden nodes via a clustering algorithm. The output of each hidden node is effectively a measure of how close an input vector is to the point represented by the weights for that node. Therefore, choosing hidden nodes whose weights represent "prototypes" for points in the training set makes sense [10]. This means that this stage of the training process does not need to change at all for our aggregate output learning problem: the difference in our training set and a traditional one lies in the fact that the output values are aggregated, but not the input vectors. It is precisely because clustering is an *unsupervised* procedure that we can leverage it unchanged.

The second stage of training an RBF network, once the weights for the hidden nodes have been determined, is to find optimal values for the weights v_j. We seek to minimize the error over all aggregate collections. This total error can be represented as

$$E = \tfrac{1}{2} \sum_c (y_c - \sum_{\ell \in c} \sum_j v_j a_{\ell j})^2 \tag{7}$$

where y_c is the output value for aggregate collection c, $a_{\ell j}$ is the output from hidden node a_j associated with the ℓ-th input vector in aggregate collection c, and (with a slight abuse of notation), $\ell \in c$ represents the indices of the data associated with aggregate collection c. The summation over c is understood to run over all aggregate collections.

The weights from the first layer of the network remain fixed, so the terms y_c and $a_{\ell j}$ in the above error are fixed. Optimizing for the best values of v_j is therefore a straightforward unconstrained quadratic optimization problem.

As in the classification case, this new version of an RBF is quite similar to the traditional methodology, and the algorithms for using it are similar. But again, it should be pointed out that this new approach differs from the original in that it does not constrain the output from each individual input vector to match a predetermined output, but rather constrains sums of the outputs from collections of input vectors to match given training set values.

4.5. SVM: Classification

In developing an SVM approach for solving the classification version of the aggregate output learning problem, we observe that the problem is similar in some ways to the *semi-supervised learning problem* [2, 3]. The semi-supervised learning problem consists of both labeled and unlabeled training data, and the goal is to use the unlabeled data to improve classification accuracy over using just labeled data. In our problem, none of the data is labeled individually, but a count of the number of labels of each type is provided for each aggregate set. These two problems are not the same, but work by Bennett and Demiriz on the semi-supervised problem [2] yields insights on how to appropriately adapt linear SVMs for use with unlabeled data. Our approach heavily leverages their ideas. Therefore, we only consider linear support vector machines in this work.

We again shift our notation slightly for clarity of exposition. The standard SVM for classification [6, 19] is

$$\begin{array}{rll} \min_{(\mathbf{w},b,\boldsymbol{\xi}\geq 0)} & \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_i \xi_i & \\ \text{s.t.} & y_i(\mathbf{w}\cdot\mathbf{x}_i - b) + \xi_i & \geq \quad 1 \end{array} \tag{8}$$

where the subscript i ranges over all training rows, $\mathbf{x}_i$ represents a particular training input vector, and the vector $\mathbf{w}$ and scalar b represent the coefficients of the separating hyperplane. ξ_i is a measure of the error associated with the output from input vector $\mathbf{x}_i$, and C is a user chosen parameter that balances the tradeoff of accuracy against overfitting. y_i is a 1 or a -1, depending on to which class the training

For the aggregate output learning problem, we do not know to which class each training point belongs. We do know how many points from each class that there are supposed to be in each aggregate collection, though. Therefore, similar to Bennett and Demiriz [2], we modify the above quadratic program to the following mixed integer program. Here, we use the indicator variable d_i to be a 1 if the point is in class 1 and 0 if the point is in class -1:

$$\begin{array}{rc} \min\limits_{(\mathbf{w},b,\boldsymbol{\xi}\geq 0,\mathbf{z}\geq 0,\boldsymbol{\eta}_c)} & \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_i(\xi_i + z_i) + D\sum_c \eta_c \\ \text{s.t.} & \mathbf{w}\cdot\mathbf{x}_i - b + \xi_i + M(1-d_i) \geq 1 \\ & -(\mathbf{w}\cdot\mathbf{x}_i - b) + z_i + Md_i \geq 1 \\ & -\eta_c \leq y_c - \sum_{l\in c} d_l \leq \eta_c \end{array} \tag{9}$$

The constant M is chosen to be sufficiently large so that if $d_i = 0$, then $\xi_i = 0$ satisfies the first constraint. Similarly, if $d_i = 1$, then $z_i = 0$ satisfies the second constraint. ξ_i and z_i represent the misclassification errors for each point, measured as a traditional SVM would. In this case, however, since we do not know in advance to which class each point belongs, the error is effectively taken to be the minimum error for either of the two classes. The subscript c is used to represent an individual aggregate collection; $\ell \in c$ represents the indices of all input vectors associated with collection c, and y_c represents the actual number of points associated with class 1 for collection c. The term η_c works to ensure that the number of points assigned to each class is consistent with the aggregates provided for the training set. η_c is the difference between the predicted and the actual count of the number of points in class 1 for an aggregate collection c. We therefore sum this error over all points and add it to the objective function, multiplying it by a parameter D to balance the importance of matching the desired aggregate accuracy level for each collection.

The solution to this optimization problem can be found via any mixed integer quadratic programming solver. This approach is somewhat slow, however, and we acknowledge that a faster algorithm can likely be constructed. For example, the semi-supervised work by Bennett and Demiriz was sped up in two fashions. First, they used the popular substitution of $||w||_1$ instead of $\frac{1}{2}||w||_2^2$ in the objective function. This transforms this mixed integer quadratic program into a mixed integer linear program. Furthermore, Fung and Mangasarian [8] reformulated this problem as a concave minimization problem and used a successive linear approximation algorithm. Such an approach might work here as well. That is outside the scope of this particular paper, however, whose role is to present the framework for our new learning problem and to look at some initial efforts in solving it. We have therefore chosen to present a formulation as similar as possible to traditional SVMs. Nonetheless, leveraging approximation approaches similar to the ones described above would be worthy of examining in future work.

4.6. SVM: Regression

Developing a version of support vector regression (SVR) for the aggregate output learning problem is considerably simpler than for classification. The standard linear SVR approach [6, 19] is expressed as:

$$\begin{array}{rl} \min_{(\mathbf{w},b,\boldsymbol{\xi}\geq 0,\mathbf{z}\geq 0)} & \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_i(\xi_i + z_i) \\ \text{s.t.} & \mathbf{w}\cdot\mathbf{x}_i + b - y_i \leq \varepsilon + \xi_i \\ & y_i - \mathbf{w}\cdot\mathbf{x}_i - b \leq \varepsilon + z_i \end{array} \tag{10}$$

where the subscript i represents a particular training set row, $\mathbf{x_i}$ represents a particular training input vector, and the vector $\mathbf{w}$ and scalar b represent the coefficients of the regression surface. y_i represents the desired output value for each input vector. ξ_i and z_i serve to measure how far the predicted output value is from the actual; the optimization problem ensures that for each i, either ξ_i or z_i is zero depending on whether the predicted value is too small or too large. C is a user chosen parameter that balances the tradeoff of accuracy against overfitting, and ε is a user chosen parameter representing the size of the "zone of insensitivity" within which errors do not contribute.

The aggregate output version of SVR does not have an individual y_i for each input vector $\mathbf{x}_i$. Instead, each aggregate collection contains an individual aggregate output value y_c. Therefore, we can modify the SVR formulation to constrain (with slack) the outputs from all points within an

	randomness						
size	0	25	50	100	200	500	2000
auto-mpg							
2	0.15	0.22	0.23	0.32	0.36	0.46	0.40
5	0.15	0.24	0.31	0.47	0.63	0.70	0.75
10	0.15	0.26	0.33	0.52	0.72	0.85	0.90
20	0.15	0.27	0.35	0.57	0.83	0.92	0.95
housing							
2	0.25	0.28	0.32	0.41	0.47	0.50	0.51
5	0.25	0.30	0.38	0.53	0.70	0.80	0.79
10	0.25	0.32	0.41	0.58	0.77	0.89	0.88
20	0.25	0.33	0.43	0.60	0.81	0.93	0.94
cpu-small							
2	0.25	0.31	0.37	0.40	0.52	0.50	0.52
5	0.26	0.34	0.43	0.60	0.76	0.76	0.76
10	0.29	0.38	0.48	0.73	0.88	0.88	0.87
20	0.42	0.52	0.63	0.82	0.92	0.95	0.95

Table 3. MSE for three datasets using aggregate k-nearest neighbor algorithm.

aggregate collection to sum to the appropriate total:

$$\begin{array}{rl} \min_{(\mathbf{w},b,\boldsymbol{\xi}\geq 0,\mathbf{z}\geq 0)} & \frac{1}{2}||\mathbf{w}||_2^2 + C\sum_c(\xi_c + z_c) \\ \text{s.t.} & \sum_{\ell\in c}(\mathbf{w}\cdot\mathbf{x}_\ell + b) - y_c \leq \varepsilon + \xi_c \\ & y_c - \sum_{\ell\in c}(\mathbf{w}\cdot\mathbf{x}_\ell + b) \leq \varepsilon + z_c \end{array} \tag{11}$$

The subscript c is used to represent an individual aggregate collection; $\ell \in c$ represents the indices of all input vectors associated with collection c.

As in the classification case, this quadratic program can be solved by any off-the-shelf quadratic programming solver. Considerably faster algorithms for SVR are well known [5, 16], and so one or all of them might be adaptable to work with our formulation here. Similarly, our approach could likely be adapted to work with nonlinear SVMs.

5. Experimental Results

The main contributions of this paper are in proposing our new machine learning problem, and in posing some initial attempts at solving it. All three of the algorithms proposed here are natural generalizations of well-known algorithms, namely variations to k-nearest neighbor, neural networks, and support vector machines. Each of these new variations applies exactly the same philosophy to the aggregate output learning problem that the original algorithms do to the traditional supervised learning problem. In some sense, it is somewhat unclear as to what purpose experiments would serve. Since we pose a new learning problem, there are no other algorithms in the literature which are appropriate for comparison purposes. We can compare these algorithms with each other, but it is well known that different algorithms perform better on different datasets. However, we do see (at least for regression) a simpler "experimental control" technique which might be appropriate for comparison purposes, which we describe later. We therefore focus our experiments on the regression case, especially as this is the one which is most appropriate for our application.

Since we are unable at the moment to release our SPMS data, we present in this paper data from three well-known publicly available datasets. The first, *auto-mpg* [15], predicts miles per gallon based on seven variables including cylinders, origin, model year, and acceleration. Although the dataset included a car name variable, we ignored it because it was not numeric. The second dataset, *housing* [15] is the classic "Boston Housing Data." This dataset is used to predict the median value of owner-occupied homes in Boston based on thirteen variables including the crime rate, the non-retail business acreage, the average number of rooms, and so on. The final dataset, *cpu-small* [7], measures the portion of time (%) that cpus run in user mode based on fourteen variables. Variables include number of reads between system memory and user memory, number of writes between system memory and user memory, number of system read calls per second, and so on. There were 398 instances in the auto-mpg dataset, 506 in the housing dataset, and 300 in the cpu-small dataset (we chose a small subset of size 300 from the original, which had 8192 examples).

None of the above datasets have aggregate outputs. They are traditional regression datasets in that they contain an output value for each input vector. Therefore, we use them for experiments by creating aggregate training sets. After separating training data from test data under a cross-validation framework, we group together multiple input vectors in the training set and aggregate their output values together. This transforms the training set into one appropriate for the aggregate output learning problem, and leaves us with a traditional test set for measuring success. This technique for creating artificial aggregate datasets gives us the capability to run multiple experiments, each with different characteristics. Specifically, we vary the dataset in two different ways, each of which could potentially influence the performance of an aggregate output learning algorithm.

First, we vary the *size* of the aggregate sets, i.e., the number of rows in the original dataset whose outputs are summed to form each aggregate set. For the traditional supervised learning problem, all aggregate sets are of size 1. Note that the set size is essentially an upper bound on how much information is lost due to aggregation. For simplicity, all aggregate sets that we generate for a particular dataset have the same size (except for possibly the last one).

Second, we vary the amount of *randomness* in the aggregation. In order for us to be able to learn anything from aggregate data, we have been making the assumption that points within an aggregate set are somewhat related. If this assumption were invalid, the problem would seem unsolvable; we would end up with aggregate sets where each col-

	randomness						
size	0	25	50	100	200	500	2000
auto-mpg: aggregate algorithm							
2	0.53	0.74	0.46	0.48	0.50	0.74	0.59
5	0.41	0.74	0.64	0.65	0.73	0.69	0.68
10	0.43	0.49	0.44	0.48	0.96	0.84	1.04
20	0.42	0.45	0.45	0.74	0.81	1.04	1.02
auto-mpg: control algorithm							
2	0.56	0.76	0.52	0.58	0.62	0.82	0.71
5	0.43	0.77	0.73	0.77	0.86	0.83	0.83
10	0.53	0.63	0.61	0.73	1.01	0.92	0.90
20	0.51	0.52	0.61	0.87	0.87	1.02	1.04
housing: aggregate algorithm							
2	0.67	0.76	0.64	0.75	0.87	0.85	0.89
5	0.77	0.81	0.88	0.94	1.02	0.87	0.90
10	0.76	0.80	0.84	0.84	0.79	1.05	0.95
20	0.81	0.88	0.79	0.82	0.92	1.22	1.23
housing: control algorithm							
2	0.69	0.76	0.68	0.79	0.90	0.88	0.91
5	0.75	0.79	0.92	0.96	0.95	0.91	0.92
10	0.81	0.77	0.92	0.86	0.88	0.99	0.99
20	0.82	0.85	0.95	0.86	0.95	1.05	1.07
cpu-small: aggregate algorithm							
2	0.86	0.87	0.93	0.75	0.90	0.99	0.93
5	0.90	0.69	0.82	0.86	0.96	0.99	1.01
10	0.74	0.77	0.91	0.89	0.94	0.97	1.06
20	1.01	0.68	0.74	0.80	0.92	1.01	0.97
cpu-small: control algorithm							
2	0.85	0.88	0.94	0.87	0.91	1.03	0.96
5	0.90	0.79	0.93	0.90	0.97	1.00	1.01
10	0.79	0.89	0.93	0.92	0.97	0.97	1.06
20	0.91	0.91	0.90	0.98	0.94	1.01	0.98

Table 4. MSE for three datasets using aggregate and control neural network algorithms.

lection of input vectors varied over the range of the dataset, and each aggregate output value would therefore be approximately the same. Said differently, we assume that each aggregate set is different from the others in a way that provides structure to help us learn from the data. We therefore vary the level of disorder in the data for experimental purposes in order to measure this effect. To do so, the original data is first sorted by output value. Individual data points in the training set (input vectors and output values together) are randomly chosen in pairs and swapped. After a number of random pairs have been swapped, the points are taken in order starting from the top of the dataset to create the aggregate groups. Large numbers of swaps therefore correspond to relatively randomized aggregate groups, whereas low numbers of swaps correspond to highly ordered aggregate groups. The "randomness" value seen in our experimental results refers to the number of pairs that we randomly swapped before aggregating the data.

As stated above, there are no algorithms that we know of that make sense to compare our new algorithms with, since the aggregate output learning problem is new. We can, however, compare to the following simple technique. Assuming that we start with an aggregate training set (which is, of course, the problem which we are trying to solve), we create a new training set where the input vectors are the same, but each input vector is assigned its own individual output value which is the average of the known aggregate output value for the collection. This new dataset thus resembles a traditional supervised learning dataset, and thus traditional algorithms can be used on it. Of course, using traditional algorithms overconstrains the problem, as any such algorithm will try to find a predictor that matches each input vector in the training set individually to the average output value for its collection. Nonetheless, this technique requires no new algorithms, and so it seems as though it is a worthy experimental control. (We note that an alternative approach might seem to be to aggregate the input vectors within each aggregate set together in order to match the aggregated output values. This would work for training purposes, but not for testing, where the goal is to do prediction of output values for individual input vectors.)

For all experimental results, five-fold cross-validation was used. (We were running enough experiments that the savings in time over ten-fold cross-validation was convenient.) All fields in all datasets were normalized by subtracting the mean and dividing by the standard deviation; regression accuracy was measured via mean squared error.

In looking at our experimental results, it is tempting to compare test set accuracies across algorithms, i.e. to compare the results from neural networks with k-nearest neighbor. We emphasize that such comparisons are not valid. Our purpose in running these experiments is to show how our technique varies with different aggregate set sizes, and with varying amounts of randomness among the collections. Therefore, we pick a simple set of parameters for each algorithm, and generally leave them fixed throughout the experiments (we discuss these in more detail below). All three of these algorithms have considerable capability for being tweaked to improve the results. One could try a variety of values for k for the k-nearest neighbor algorithm, for example, or one could vary the number of hidden nodes in the neural network. In order for comparisons across algorithms to be valid, we would have had to make an attempt to optimize parameters across all algorithms to get the best possible results. *We have not done so.* Again, our purpose is to be able to look at each algorithm and observe its behavior on variations of the data, and to make comparisons with a control version of the algorithm with a similar set of parameters. Comparisons within the support vector machine results, for example, are completely valid and worthwhile. Comparing the SVM results with the neural network results does not make sense because we have not optimized the parameters for either appropriately.

First, we present the results from k-nearest neighbor, where we fixed $k = 5$ for all experiments. Note that for

k-nearest neighbor, the control method is mathematically equivalent to our algorithm (see the end of Section 4.2). We thus only provide one set of experiments for k-nearest neighbor, whereas for the other algorithms, we show two.

Table 3 shows the results from running k-nearest neighbor on our three datasets. We see that regression error increases as aggregate set size increases, which makes sense. As the aggregate set size increases, we are throwing more information out of the training set. Similarly, we see regression error increase as the amount of randomness in the aggregate sets increases. This makes sense as well. For highly ordered aggregate sets, within each aggregate set the (unknown) output values are quite similar to each other. Replacing each with the average for the aggregate set is a good approximation in this case. On the other hand, for highly random aggregate sets, assigning each point an output value which is the average of its aggregate set makes considerably less sense. As discussed earlier, these k-nearest neighbor results are a worthwhile benchmark for understanding our experimental techniques, but it is the results for neural networks and SVMs that illustrate the power of our approach.

We therefore present results from the neural network and support vector machine experiments. For the neural networks, we used a learning rate of 1×10^{-5} and a convergence tolerance of 0.001. The hidden nodes were determined via k-means clustering on the input vectors, and for each cluster σ was determined to be the average distance from each point in that cluster to the center. We arbitrarily fixed the number of hidden nodes to be 12. For the SVMs, we used the quadratic programming solver CPLEX [11] to handle the optimization. The loss insensitivity parameter ε was set to 0. We varied the parameter C in order to achieve the right balance of margin separation vs. data fitting. In principle, this should have been done on a tuning set, pulled out of the training set, for each individual experiment to optimize C. This opened up a complicated discussion as to how this should be done: in our scenario, the tuning set does not structurally mirror the test set. There are many approaches we might have tried. Conveniently, by varying C by orders of magnitude of 10, it was exceedingly clear for each dataset-algorithm pair that one particular value of C optimized nearly all experimental values. Since we looked at the results for a large number of experiments simultaneously and picked a single value of C for all of them, it was clear that we were not somehow picking C to optimize a single particular test set. More careful experiments might change the numbers slightly; they certainly would not change the broad conclusions we reach in understanding the nature of our algorithms.

By comparing the differences between the aggregate algorithm and the control algorithm for a given dataset and technique, the patterns are quite clear. For the neural network results shown in Table 4, we see that our aggregate algorithm outperforms the control algorithm for much of the table. The differences are most pronounced for moderate randomness values. For highly ordered data, our aggregate algorithm performs similarly to (and occasionally worse than) the control algorithm. As in the k-nearest neighbor experiments, this makes sense; when the data is highly ordered, assuming that the output value for each point is the average of the output values for the aggregate set is a good approximation. For exceedingly random data (randomness value of 2000), our algorithm again performs similarly to the control algorithm. This is likely because with highly random aggregations, there is a considerable loss of information. In this case, the aggregate algorithm does not have enough data to draw better conclusions than the control algorithms. For most of the cases, however, the aggregate algorithm performs considerably better than the control. The aggregate approach only rarely performs worse than the control, and not by much. This indicates that in general the aggregate algorithm is a stronger approach. We also observed that the aggregate algorithms tended to run faster than the control algorithms did, which makes sense since the amount of data used is reduced.

Finally, Table 5 shows the results for the SVM aggregate algorithm compared with its control. The comparison between the aggregate case and the control case is very similar to the one for the neural network algorithms, and again illustrates the outcome of our approach. One significant difference appears to be in the exceedingly random cases (randomness value of 2000) where the aggregate approach performs dramatically better than the control approach. This is likely due to the fact that SVMs inherently avoid overfitting by regularizing the separating surface, which has a more dramatic impact with noisy data.

6. Conclusions and Future Work

We have proposed a new machine learning problem, known as the aggregate output learning problem, that does not seem to have been previously examined in the literature. This problem, though inspired from atmospheric data analysis, could have broad ramifications in working with data masked for privacy purposes. We present a formal framework for this problem for both regression and classification, and provide adaptations of k-nearest neighbor, neural networks, and support vector machines (classification and regression for each) to handle the aggregate problem. We summarize a series of experiments for the regression framework that show our approach to be highly effective.

For SVM classification, we have shown a new connection between aggregate output learning and semi-supervised learning. The aggregate output learning problem may thus illustrate new insights into semi-supervised learning.

There is considerable future work that could be done. We

	randomness						
size	0	25	50	100	200	500	2000
auto-mpg: aggregate algorithm							
2	0.22	0.21	0.21	0.21	0.21	0.21	0.22
5	0.23	0.22	0.22	0.22	0.23	0.23	0.22
10	0.24	0.22	0.22	0.23	0.23	0.23	0.23
20	0.27	0.24	0.26	0.23	0.25	0.28	0.28
auto-mpg: control algorithm							
2	0.20	0.21	0.23	0.29	0.37	0.45	0.36
5	0.20	0.23	0.30	0.47	0.62	0.69	0.69
10	0.20	0.25	0.33	0.51	0.74	0.83	0.85
20	0.20	0.25	0.33	0.54	0.80	0.92	0.92
housing: aggregate algorithm							
2	0.32	0.31	0.31	0.32	0.32	0.32	0.33
5	0.32	0.31	0.32	0.32	0.32	0.35	0.34
10	0.36	0.33	0.33	0.35	0.34	0.38	0.39
20	0.36	0.34	0.38	0.41	0.40	0.44	0.43
housing: control algorithm							
2	0.31	0.31	0.34	0.38	0.45	0.48	0.48
5	0.31	0.32	0.37	0.53	0.69	0.76	0.74
10	0.31	0.33	0.39	0.57	0.78	0.86	0.84
20	0.31	0.33	0.39	0.58	0.79	0.93	0.91
cpu-small: aggregate algorithm							
2	0.34	0.30	0.28	0.28	0.29	0.28	0.28
5	0.26	0.33	0.23	0.21	0.34	0.31	0.32
10	0.34	0.35	0.40	0.32	0.38	0.39	0.29
20	0.65	0.59	0.51	0.44	0.50	0.42	0.48
cpu-small: control algorithm							
2	0.28	0.27	0.31	0.36	0.36	0.44	0.39
5	0.23	0.28	0.43	0.50	0.65	0.71	0.70
10	0.28	0.33	0.46	0.66	0.80	0.88	0.85
20	0.40	0.46	0.62	0.76	0.88	0.94	0.93

Table 5. MSE for three datasets using aggregate and control support vector machines.

have also proposed algorithms for the classification framework: experiments testing these techniques would be worthwhile. The support vector machine approaches that we have proposed are all linear, and we would like to develop nonlinear versions as well. The SVM community has developed fast algorithms for solving SVMs. Applying those ideas to speed up our algorithms would be worthwhile.

7. Acknowledgements

Most of the programming for our experiments, as well as significant assistance with the literature search, was done by a collection of undergraduates as part of a data mining course at Carleton College. We acknowledge and thank Rob Atlas, David Barbella, Deborah Chasman, Mark Dyson, Reid Gilman, Thomas Hagman, Bret Jackson, Daniel Lew, Brandy McMenamy, Peter Nelson, Charles Noneman, Jerad Phelps, Kevin Reschke, Ed Williams, Paul Wilmes, and Kelson Zawack. The efforts by these students were critical to the success of this project. Other portions of this work were funded by NSF ITR grant IIS-0326328.

References

[1] G. Arminger, N. Lijphart, and W. Müller. Die verwendung log-linearer modelle zur disaggregierung aggregierter daten. *Allgemeines Statistisches Archiv*, 3:273–291, 1981.

[2] K. Bennett and A. Demiriz. Semi-supervised support vector machines. In *Advances in Neural Information Processing Systems*, volume 12, pages 368–374. MIT Press, 1998.

[3] A. M. Bensaid, L. O. Hall, J. C. Bezdek, and L. P. Clarke. Partially supervised clustering for image segmentation. *Pattern Recognition*, 29:859–871, 1996.

[4] B.-C. Chen, L. Chen, R. Ramakrishnan, and D. R. Musicant. Learning from aggregate views. In *Proceedings of the 22nd International Conference on Data Engineering*, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[5] R. Collobert and S. Bengio. SVMTorch: Support vector machines for large-scale regression problems. *Journal of Machine Learning Research*, 1(1):143–160, February 2001.

[6] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, Cambridge, 2000.

[7] Delve. Data for evaluating learning in valid experiments. `http://www.cs.utoronto.ca/~delve`.

[8] G. Fung and O. Mangasarian. Semi-supervised support vector machines for unlabeled data classification. *Optimization Methods and Software*, 15:29–44, 2001.

[9] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, Cambridge, MA, 2001.

[10] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2nd edition, 1999.

[11] ILOG CPLEX Division, Sunnyvale, CA. *ILOG CPLEX 10.1*, 2007.

[12] R. Jiroušek and S. Přeučil. On the effective implementation of the iterative proportional fitting procedure. *Computational Statistics & Data Analysis*, 1995.

[13] N. McGrogan, C. M. Bishop, and L. Tarassenko. Neural network training using multi-channel data with aggregate labelling. In *Proceedings of the Ninth International Conference on Artificial Neural Networks*, volume 2, pages 862–867. IEE, 1999.

[14] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[15] P. M. Murphy and D. W. Aha. UCI repository of machine learning databases, 1992. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[16] S. Rüping. mySVM, September 2001. `http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM`.

[17] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc., New Jersey, second edition, 2003.

[18] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, May 2005.

[19] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[20] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.

[21] Z. Yang, S. Zhong, and R. N. Wright. Privacy-preserving classification of customer data without loss of accuracy. In *Proceedings of the Fifth SIAM International Conference on Data Mining*, pages 92–102, 2005.