Four Results in Matching Data Distributions

Carlos R. González, Yaser S. Abu-Mostafa
California Institute of Technology

June 21, 2014

1 First Result - Targeted Weighting Algorithm

1.1 Main Problem Being Addressed

In Machine Learning, systems are trained with data that is assumed to have the same distribution as the data that will be used for testing later on. In Recommender Systems and other application domains, this assumption does not hold, as the time component of data, which can be seen in changes in fashion, trends, opinion, etc., alters the distribution of the data. Other effects like sampling bias or simply a change in conditions can also alter the distributions. There are various methods that propose to match through weighting mechanisms on the training data, so that the training set looks similar to the data that the system will be used on, hoping to improve performance. Generally, the data the system will be used on may be available, but not labeled, as finding the labels is precisely the function of the Recommender System. This means that there is no direct way of validating if the use of weights improved performance of the system since we do not have labeled points from the test set.

It turns out that using weights to match training and test distributions, can be helpful, but can also do harm, as the use of weights results in an effective loss of samples. The smaller the sample size, the worse the learned function produced by the system. Therefore, we propose a method that allows us to find if using weights is beneficial or harmful in a particular scenario. The advantage of our solution is that it does not require the data in the test distribution to be labeled, as it is the case in real applications.

1.2 Summary

In Machine Learning sometimes weighting mechanisms are used to match training and test distributions. We have concluded that the effect of matching can be decomposed into two terms, a positive term due to training with the appropriate distribution, and a negative term due to the sample loss. These quantities cannot be computed exactly, due to the lack of labeled points distributed as the test distribution and because the target function is unknown. Therefore, we created a model to estimate the gain (or

loss) in matching through quantities that can be estimated with the available data. All we need to know is the difference between the learned function with and without weights, and through the use of artificial target functions, we are able to estimate fully if there will be a gain or a loss in using the weighting mechanism chosen. Previously there were no methods to validate if weighting was useful in each particular case, so practitioners used these methods without knowing if weighting was aiding or hurting the solution.

1.3 Description of the Solution

The solution we provide is an algorithm that is implemented in software in order to answer the question, should one use weights or not to match training and test distributions in a particular machine learning problem. We assume a learning algorithm has been implemented to solve the particular problem. To be clear, a machine learning algorithm takes some inputs and predicts some output which usually results in solving a classification problem or a regression problem. An example of a classification problem is a Recommender System. Here the learning algorithm predicts the rating of a user for a given item, and hence the algorithm learns to classify items by order of importance to the user. As stated previously, to build such learning systems data is needed to train the algorithm so that it is able to learn the classification or prediction rule. Our solution assumes both a learning algorithm is in place, and a weighting mechanism has been chosen. To clarify, having a learning algorithm available means that given a set of training examples of ordered pairs (x,y) where x is the input and y the desired output or labels, there is a software routine that trains the learning algorithm, that is, finds all the parameters needed to output a function that when evaluated at new inputs, constitutes the prediction of the learning algorithm. These new inputs without labels where predictions are desired is referred to as the test data.

To clarify the setup we give an example. Assume we had the task of recommending products to users in Amazon or any other e-commerce shop. Also assume there was already a learning algorithm in place that given as inputs a user and an item, would be able to determine if the user would give a high or low rating to the given item. The system would have been trained with previously acquired data, consisting of rated items by different users. With such a system in place, it would be possible to recommend items to the user, as this would simply require ranking the desired items according to this learned function, and showing the top ranked items to the user. However, assume now that many of the items we trained with, are subject to changes in fashion and trends. That is, some items may become very unpopular after a few months. Then, the system might be trained so that it uses weights to change the importance that it gives to some of the items that users rated previously have during the training state. The system can gather the statistics of what items it wants to predict ratings for, and from these it can quickly find out which items are no longer popular in this test set. With this information then the learning system may use weights to make the training distribution look like the test distribution and output a

new solution. In this particular example, we are assuming the training distribution is different to the test distribution because of popularity of items, reflected in the amount of times an item is rated in the store, but the difference between the two distributions may occur along many other dimensions of the data or coordinates. Then, our solution answers the question: will using weights improve or harm the performance of the learning algorithm?

The above question is very relevant as we were able to show that the use of weights is both beneficial and harmful. It is beneficial in the sense that we learn under the same conditions that the system will be tested on. It is harmful in the sense that the use of weights effectively reduces the sample size of the training set, and with a reduced set, the function learned is further from the true target. These findings were confirmed both in synthetic data sets as well as in a real data set like the Netflix Data Set. Finding which of the two terms is larger is the key to our solution.

Through simulations we were able to show empirically that the gain in weighting can be decomposed into the two terms mentioned above as

$$Gain = MatchBenefit - SampleLoss,$$

with the three above quantities defined in terms of the out of sample gain or loss by changing the training conditions. The three quantities can only be found in a hypothetical scenario where we know what function we are predicting. This of course is never known in practice, and for this reason we found a model that estimates this Gain. The Gain is actually equal to the equation of Step 9 in the algorithm below. Here the Gain is estimated with quantities that are always available in a practical set up. The figures below show how this model estimates the Gain in simulations, and the clear linear fit obtained shows how the model is a very accurate proxy for the Gain.

So assuming a setup similar to the above, where a routine for training the learning algorithm is available, and a set of weights has been already chosen, then the solution we report is a software implementation of the following steps to determine if the learning algorithm works better if trained with or without weights.

- Train algorithm with available data. Call the learned function g.
- Train algorithm with available data and weights desired. Call the output function g_w
- Evaluate g and g_w in the test set, to find the $E[(g - g_w)^2]$ where $E[]$ is the expected value
- Generate n artificial target function, denoted f_i for $i = 1$ to n (below we give details of how to do this)
- For each artificial target function, evaluate it on the training set points to obtain a new set of labels y_i
- With each artificial set of labels train algorithm to learn functions g_i

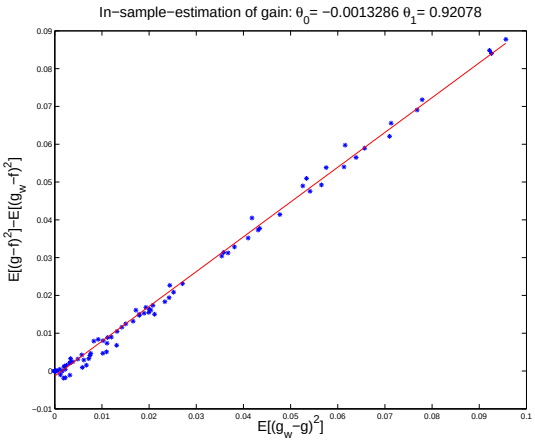

Figure 1: Gain vs Proxy: The Linear Fit with slope very close to 1 shows how our proxy is able to estimate the Gain when weighting is done. Plot consists of simulations with thousands of runs for different target functions and distributions of the data used to find each of these points.

- With each artificial set of labels train algorithm with weights to learn functions g_{wi}
- For each artificial target function compute: $T_{0i} = E[(g_i - f_i)^2] - E[(g_{wi} - fi)^2] - E[(g_i - g_{wi})^2]$
- Count for how many artificial targets $E[(g - g_w)^2] + T_{0i} > 0$. Call this count $Counts$
- If $Counts > n/2$ solution concludes using weights for training improves performance. Else it concludes using weights hurts performance

The above steps outline the steps that a software implementation should take in order to obtain the desired answer. The only remaining detail to specify is how to generate the artificial target functions. The way to do this is in the following way:

- Choose random parameters for the function that the learning algorithm wants to learn. This means that some artificial target function is already found, call this f_{0i}.
- With this artificial function, compute the output on each of the points of the training set.
- Find the mean and standard deviation of the above set, call these m_i and s_i
- Find the mean and standard deviation of the original training set, call these m and s

- Artificial target function $f_i(x) = t((f_{0i}(x) - m_i)s/s_i + m_i)$,

where $t(\cdot)$ is a non-linear threshold function, whose choice depends on the range of values that the target function takes. If the target function is binary, a good choice for $t()$ is the sign() function. If the target function takes multiple discrete values (multi class problem) a good choice is the round() function. Finally if the target function is continuous (regression problem) then noise should be added to the function such that: $f_i(x) = (f_{0i}(x) - m_i)s/s_i + m_i + \epsilon$, where ϵ is Gaussian noise with standard deviation of the order of the usual regression error of the problem and no threshold function is used.

2 Second Result - Hard Weight Matching Algorithm

2.1 Main Problem Being Addressed

In Machine Learning, systems are trained with data that is assumed to have the same distribution as the data that will be used for testing later on. In Recommender Systems and other application domains, this assumption does not hold, as the time component of data, which can be seen in changes in fashion, trends, opinion, etc., alters the distribution of the data. Other effects like sampling bias or simply a change in conditions can also alter the distributions. There are various methods that propose to match through weighting mechanisms on the training data, so that the training set looks similar to the data that the system will be used on, hoping to improve performance. Generally, the data the system will be used on may be available, but not labeled, as finding the labels is precisely the function of the Recommender System. Yet this data can be used to find the desired weights.

Finding weights is not a trivial problem since the actual probability distributions are not known and estimating them is an even harder problem. Also, the fact that these probability distributions lie in high-dimensional spaces makes it impossible to match exactly the two distributions. Here we propose a computationally simple algorithm to find weights that can match two distributions along any number of coordinates. .

2.2 Summary

As described above, there are many situations in Machine Learning where training and test distributions are different. Learning in such a scenario is in principle less beneficial than learning in a scenario where the two distributions are equal, which is the set up where all learning theory has been developed in. One way to correct this mismatch is by using weights in the training samples, so that the distributions are matched in some sense. We propose to match along coordinates or projections of the data, and do so by discretizing the domain in each of the desired coordinates. Matching along projections avoids computing exponentially large sums, while discretizing

the coordinates allows accounting for the fact that even if the training and test sets come from the same distribution, they can have differences due to the finite sample size. With these simplifications, the problem reduces to solve a convex optimization problem, for which we derive an iterative solution.

2.3 Description of the Solution

The solution we provide is an algorithm that is implemented in software in order to find the weights that should be assigned to each of the points in the training set, before carrying out the learning process. That is, finding the weights is a pre-processing stage, before actually using a learning algorithm to be trained with the data available. We assume that a learning algorithm is in place and is able to take the weights as inputs. To be clear, a machine learning algorithm takes some inputs and predicts some output which usually results in solving a classification problem or a regression problem. An example of a classification problem is a Recommender System. Here the learning algorithm predicts the rating of a user for a given item, and hence the algorithm learns to classify items by order of importance to the user. As stated previously, to build such learning systems data is needed to train the algorithm so that it is able to learn the classification or prediction rule. Our solution assumes a learning algorithm is in place, but that a weighting mechanism needs to be chosen. To clarify, having a learning algorithm available means that given a set of training examples of ordered pairs (x,y) where x is the input and y the desired output or labels, there is a software routine that trains the learning algorithm, that is, finds all the parameters needed to output a function that when evaluated at new inputs, constitutes the prediction of the learning algorithm. These new inputs without labels where predictions are desired are referred to as the test data. The learning algorithm can take as input weights so that some points in the training set are given more importance than others.

To clarify the setup we give an example. Assume we had the task of recommending products to users in Amazon or any other e-commerce shop. Also assume there was already a learning algorithm in place that given as inputs a user and an item, determines if the user would give a high or low rating to the presented item. The system would have been trained with previously acquired data, consisting of rated items by different users. With such a system, it would be possible to recommend new items to the user, as this would simply require ranking the desired items according to this learned function, and showing the top ranked items to the user. However, assume now that many of the items we trained with are subject to changes in fashion and trends. That is, some items may become very unpopular after a few months. Figure 3 shows an example on real data where such a mismatch occurs. The figure shows histograms of popularity of ratings in the training set, the test set, and finally, a transformation of the training set through the use of weights, that makes the data in the training set look just like the data in the test set.

The algorithm used to make the training set look like the test set is the solution we describe below. Once those weights have been found, the system might be trained

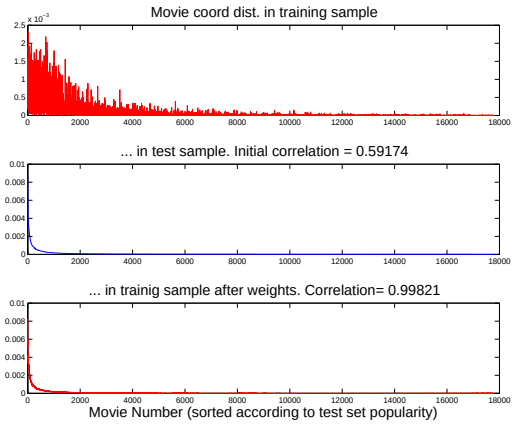

Figure 2: Histograms of movie popularity in the Netflix Data Set. The first plot shows the histogram in the training set, the following one shows the histogram in the test set. The final plot shows a weighted training set that results in the same distribution as the test set.

so that it uses weights to change the importance that it gives to some of the items that users rated previously, during the training stage. To find the weights, the system can gather the statistics of what items it wants to predict ratings for, and from these it can find out which items are no longer popular in this test set. In this particular example, we are assuming the training distribution is different to the test distribution because of popularity of items, reflected in the amount of times an item is rated in the store, but the difference between the two distributions may occur along many other dimensions of the data or coordinates. The process for finding such weights constitutes the solution reported.

Assume we have picked which coordinates to match from. Let these coordinates be 1 to C. Then, in words, what the algorithm does is that it matches the frequency of points in equally spaced bins along the chosen coordinate, for the training and test sets. Now if we picked to match along C coordinates, the algorithm will match the frequency of points in each bin along all of the C projections. Since this set of constraints gives many different solutions for the weights, we pick those weights that are closest to 1. That is, the weights that require the least deviation from their usual weight, as we explain that the use of weights can result in a harmful effect (see First Result).

Formally, this can be written as the following optimization problem.

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\sum_{i\in R}(w_i - 1)^2 \\ \text{subject to} \quad & \frac{1}{N_R}\sum_{i:\theta_c(i)=t} w_i = \nu_c(t), \quad \text{for } 0 \le t < T_c, 1 \le c \le C \end{aligned}$$

where w_i are the weights, R is the set of training points, N_R is the number of points in the training set, ν_c is a vector of frequency of points falling into each of the bins along coordinate c, θ_c is a function that determines in which bin of coordinate c point i falls into, and T_c is the number of bins along coordinate c.

Our solution consists of the equations that find these weights. The solution is made of two equations that implemented iteratively find the correct weights.

$$\begin{aligned} w_i &= 1 - \sum_{c=1}^{C} \mu_c(\theta_c(i)) \\ \mu_c(\tau_c) &= \frac{1}{n_c(\tau_c)} \left(n_c(\tau_c) - N_R \nu_c(\tau_c) - \sum_{\substack{i \in R \\ \theta_c(i)=\tau_c}} \sum_{\substack{k \neq c \\ k=1}}^{C} \mu_k(\tau_c) \right) \end{aligned}$$

In the above equations, n_c is a vector holding the count of points in the training set in each of the bins, of coordinate c. So the algorithm is the following

- Initialize all $\mu_c(\tau_c)$ to 0.
- Compute all $\mu_c(\tau_c)$ for each bin and each coordinate using the second equation.
- Go back to step 2 until convergence. By convergence we mean that the values of $\mu_c(\tau_c)$ do not change anymore.
- Use the first equation to compute the weights.

In practice we were able to see that for stability it is better to update these values with the following standard technique in numerical methods. If ?old is the value used to compute the ?new, we recommend that in the next iteration ?new is used, where

$$\mu'_{new} = \alpha \mu_{new} + (1 - \alpha) * \mu_{old}$$

with $\alpha = 0.1$ or $\alpha = 0.01$. This requires using around 100 and 1000 iterations respectively to achieve convergence.

The above solution can be implemented in any desired programming language.

Two questions remain to be answered when using the above solution. The first one is which coordinates to project the data along. The second one is how to choose the number of bins in each coordinate. The first question depends on the particular application. When the system is being designed, the practitioner may be able to identify potential mismatches in certain coordinates. Going back to the Recommender Systems example, typical coordinates may include item popularity, user load (i.e. amount of times users rate items), absolute time when ratings were made, time between ratings for each user, time since the first rating of users, among others. These of course are typical coordinates that in a Recommender System come up. In a different application where perhaps thousands of features constitute each data point,

a more practical approach would be to obtain the first few Principal Components of the data, and match along these projections. This makes the solution practical, as otherwise it would be necessary to project along every single feature of the data which is not reasonable for data in very high dimensions.

The second question regarding the number of bins is a free parameter that can be adjusted through cross-validation, as it is the case with many hyper-parameters in Machine Learning. However, a good rule of thumb is to have bins large enough so that no bins end up empty or with very few points. If this would happen, the weights would have to be very big to accommodate for those scenarios and this would hurt the learning algorithm substantially. In the Recommender System example given, if ratings are stored with a time stamp indicating the day, hour, and minute, it would probably not be beneficial to bin items by hour. Yet, using bins that group items rated in the same day, or perhaps even items rated during the same week, can lead to better results. In any case, this decision is application dependent and cross-validation can help adjust these parameters.

3 Third Result - Soft Weight Matching Algorithm

3.1 Main Problem Being Addressed

In Machine Learning, systems are trained with data that is assumed to have the same distribution as the data that will be used for testing later on. In Recommender Systems and other application domains, this assumption does not hold, as the time component of data, which can be seen in changes in fashion, trends, opinion, etc., alters the distribution of the data. Other effects like sampling bias or simply a change in conditions can also alter the distributions. There are various methods that propose to match through weighting mechanisms on the training data, so that the training set looks similar to the data that the system will be used on, hoping to improve performance. Generally, the data the system will be used on may be available, but not labeled, as finding the labels is precisely the function of the Recommender System. Yet this data can be used to find the desired weights.

Finding weights is not a trivial problem since the actual probability distributions are not known and estimating them is an even harder problem. Also, the fact that these probability distributions lie in high-dimensional spaces makes it impossible to match exactly the two distributions. However since the use of weights can also be harmful in weighting (see First Result) we propose a computationally simple algorithm to find weights that can match two distributions along any number of coordinates up to a desired point. Not matching exactly, or soft matching as we refer to the algorithms below, reduces the negative effects of weighting.

3.2 Summary

As described above, there are many situations in Machine Learning where training and test distributions are different. Learning in such a scenario is in principle less

beneficial than learning in a scenario where the two distributions are equal, which is the set up where all learning theory has been developed in. One way to correct this mismatch is by using weights in the training samples, so that the distributions are matched in some sense. We propose to match along coordinates or projections of the data, and do so by discretizing the domain in each of the desired coordinates. Matching along projections avoids computing exponentially large sums, while discretizing the coordinates allows accounting for the fact that even if the training and test sets come from the same distribution, they can have differences due to the finite sample size. Added to these simplifications, we propose to soft match the projections, to avoid the negative effects that the use of weights can introduce in a learning system.

3.3 Description of the Solution

The solution we provide is an algorithm that is implemented in software in order to find the weights that should be assigned to each of the points in the training set, before carrying out the learning process. That is, finding the weights is a preprocessing stage, before actually using a learning algorithm to be trained with the data available. We assume that a learning algorithm is in place and is able to take the weights as inputs. To be clear, a machine learning algorithm takes some inputs and predicts some output which usually results in solving a classification problem or a regression problem. An example of a classification problem is a Recommender System. Here the learning algorithm predicts the rating of a user for a given item, and hence the algorithm learns to classify items by order of importance to the user. As stated previously, to build such learning systems data is needed to train the algorithm so that it is able to learn the classification or prediction rule. Our solution assumes a learning algorithm is in place, but that a weighting mechanism needs to be chosen. To clarify, having a learning algorithm available means that given a set of training examples of ordered pairs (x,y) where x is the input and y the desired output or labels, there is a software routine that trains the learning algorithm, that is, finds all the parameters needed to output a function that when evaluated at new inputs, constitutes the prediction of the learning algorithm. These new inputs without labels where predictions are desired are referred to as the test data. The learning algorithm can take as input weights so that some points in the training set are given more importance than others.

To clarify the setup we give an example. Assume we had the task of recommending products to users in Amazon or any other e-commerce shop. Also assume there was already a learning algorithm in place that given as inputs a user and an item, determines if the user would give a high or low rating to the presented item. The system would have been trained with previously acquired data, consisting of rated items by different users. With such a system, it would be possible to recommend new items to the user, as this would simply require ranking the desired items according to this learned function, and showing the top ranked items to the user. However, assume now that many of the items we trained with are subject to changes in fashion and trends. That is, some items may become very unpopular after a few months.

Figure 2 shows an example on real data where such a mismatch occurs. The figure shows histograms of popularity of ratings in the training set, the test set, and finally, a transformation of the training set through the use of weights, that makes the data in the training set look just like the data in the test set.

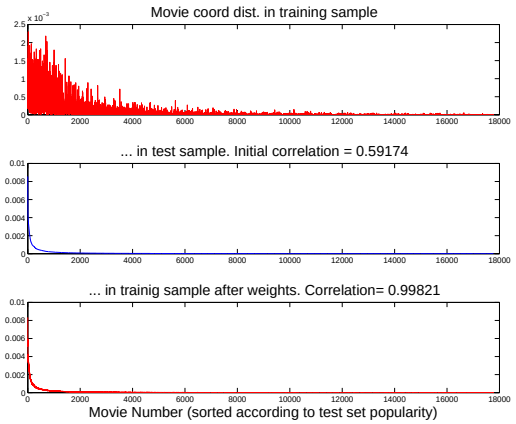

Figure 3: Histograms of movie popularity in the Netflix Data Set. The first plot shows the histogram in the training set, the following one shows the histogram in the test set. The final plot shows a weighted training set that results in the same distribution as the test set.

Here, we describe an algorithm that is used to make the training set look like the test set but that has the flexibility of allowing differences between the two distributions. The level of matching is controlled with a free parameter, and we refer to these methods as soft matching algorithms. Once those weights have been found, the system might be trained so that it uses weights to change the importance that it gives to some of the items that users rated previously, during the training stage. To find the weights, the system can gather the statistics of what items it wants to predict ratings for, and from these it can find out which items are no longer popular in this test set. In this particular example, we are assuming the training distribution is different to the test distribution because of popularity of items, reflected in the amount of times an item is rated in the store, but the difference between the two distributions may occur along many other dimensions of the data or coordinates. The process for finding such weights constitutes the solution reported.

Assume we have picked which coordinates to match from. Let these coordinates be 1 to C. Then, in words, what the algorithm does is that it minimizes the sum of the weights squared centered on an initial weight of one, while also trying to minimize another term that measures the difference of frequency of points in each bin in each coordinate of the test set, with the weighted sum of points in each bin and each coordinate in the training set. The second term, which guarantees matching, can have as much or as little weight as desired in the objective function, in order to reach a solution that minimizes the deviation of weights from their initial value of one.

This is done so that the trade off between the benefit of matching and the effective loss of samples that weighting brings gives a positive gain. Such gain can be actually quantified with the solution described in the First Result.

The above description of the problem can be formally written as the following optimization problem:

$$\text{minimize} \frac{1}{2}\sum_{i\in R}(w_i-1)^2 + \sum_{c=1}^{C}\lambda_c \sum_{\theta=0}^{T_c-1}\left(\left(\frac{1}{N_R}\sum_{i:\theta_c(i)=t} w - i\right) - \nu_c(t)\right)^2$$

where w_i are the weights, R is the set of training points, N_R is the number of points in the training set, ν_c is a vector of frequency of points falling into each of the bins along coordinate c, θ_c is a function that determines in which bin of coordinate c point i falls into, and T_c is the number of bins along coordinate c. The parameters λ_c determine how much importance is given to matching. If they were set to an infinite value, we would recover the optimization problem that is described in Section **??**. If they were set to 0, all weights would remain equal to one.

Our solution consists of the equations that find these weights. The solution is made of two equations that implemented iteratively find the correct weights.

$$\begin{aligned} w_i &= 1 + \sum_{c=1}^{C}\mu_c(\theta_c(i)) \\ \mu_c(\tau_c) &= \frac{1}{n_c(\tau_c)}\left(N_R\nu_c(\tau_c) - n_c(\tau_c) - \sum_{\substack{i\in R \\ \theta_c(i)=\tau_c}}\sum_{\substack{k\neq c \\ k=1}}^{C}\mu_k(\tau_c)\right) \end{aligned}$$

In the above equations, n_c is a vector holding the count of points in the training set in each of the bins, of coordinate c. So the algorithm is the following

- Initialize all $\mu_c(t)$ to 0.
- Compute all $\mu_c(t)$ for each bin and each coordinate using the second equation.
- Go back to step 2 until convergence. By convergence we mean that the values of $\mu_c(t)$ do not change anymore.
- Use the first equation to compute the weights.

In practice we were able to see that for stability it is better to update these values with the following standard technique in numerical methods. If ?old is the value used to compute the ?new we recommend that in the next iteration ?new is used, where

$$\mu'_{new} = \alpha\mu_{new} + (1-\alpha) * \mu_{old}$$

with $\alpha = 0.1$ or $\alpha = 0.01$. This requires using around 100 and 1000 iterations respectively to achieve convergence.

The above solution can be implemented in any desired programming language.

Three questions remain to be answered when using the above solution. The first one is which coordinates to project the data along. The second one is how to choose the number of bins in each coordinate. The final question is what value to use for the c. The first question depends on the particular application. When the system is being designed, the practitioner may be able to identify potential mismatches in certain coordinates. Going back to the Recommender Systems example, typical coordinates may include item popularity, user load (i.e. amount of times users rate items), absolute time when ratings were made, time between ratings for each user, time since the first rating of users, among others. These of course are typical coordinates that in a Recommender System come up. In a different application where perhaps thousands of features constitute each data point, a more practical approach would be to obtain the first few Principal Components of the data, and match along these projections. This makes the solution practical, as otherwise it would be necessary to project along every single feature of the data which is not reasonable for data in very high dimensions.

The second question regarding the number of bins is a free parameter that can be adjusted through cross-validation, as it is the case with many hyper-parameters in Machine Learning. However, a good rule of thumb is to have bins large enough so that no bins end up empty or with very few points. If this would happen, the weights would have to be very big to accommodate for those scenarios and this would hurt the learning algorithm substantially. In the Recommender System example given, if ratings are stored with a time stamp indicating the day, hour, and minute, it would probably not be beneficial to bin items by hour. Yet, using bins that group items rated in the same day, or perhaps even items rated during the same week, can lead to better results. In any case, this decision is application dependent and cross-validation can help adjust these parameters.

The third question, picking the value of the c can be answered through cross validation, particularly using the method described in the first section. A way to quickly explore the value that should be used, c is to use powers of 10 for this parameter, from 0.01 to 1000, and using the method described in the first section, a set of weights that returns a gain can be found.

4 Fourth Result - Dual Distribution

4.1 Main Problem Being Addressed

In Machine Learning, systems are trained with data that is assumed to have the same distribution as the data that will be used for testing later on. Furthermore, in scenarios where the above assumption does not hold, there is significant work that introduces methods to match the training distribution to the test distribution. For example, the Second and Third Results serve this purpose.

However, up to now the literature assumes that systems will perform best if training and test distributions are matched. As we describe here, this is not necessarily the case. Instead, we introduce a new concept: dual distributions. A dual distribution

is defined as a training distribution that minimizes the out-of-sample error (performance) of a learning algorithm, given a test distribution. As we describe in this report, surprisingly the dual distribution is not necessarily equal to the test distribution. Furthermore, the report describes how to find a dual distribution in the case where we consider a discrete input space, and finally using these tools, it describes how to tackle problems where the input space is continuous.

4.2 Summary

The invention described introduces a new theoretical concept, namely dual distributions, and how to apply it. First of all we establish that given a test distribution there exists a training distribution that maximizes performance, which we define as the dual distribution. We then introduce a method to find such dual distribution in the case where the input space of the problem is discrete. Finally we describe how to apply these tools in the case there the input space is continuous.

The concept of dual distributions is new, as the literature had always assumed that the best distribution to use for training a learning system was equal to the test distribution. As we describe here, this is not always the case. Once a dual distribution is found, matching methods as the ones proposed in previous Results can then be used to improve performance of any learning system, even if there was no previous mismatch between the training and test distributions. Hence this concept can be applied to any learning scenario.

4.3 Description of the Solution

The first step in the invention is to realize that in learning systems, it is not necessarily the case that training with data coming from the same distribution as the data with which the system will be tested on will yield the best possible performance. This used to be a common unquestioned assumption in Machine Learning theory as well as in practice. Performance, in Machine Learning is measured using the out-of-sample error, that is, the error the algorithm makes on unseen data. As explained in previous reports, Machine Learning systems are algorithms that find the best possible approximation of a function or rule that needs to be learned, so that these algorithms can solve classification problems in the case where we want to learn discrete decisions, that can be either binary or multi-valued, or they can solve regression problems in the case where we want to learn real-valued functions.

However, an elaborate Monte Carlo simulation shows clear evidence that using unmatched distributions can outperform matched ones. Figure 4 shows the results of this simulation, where hundreds of thousands of classification learning scenarios were averaged. The figure shows in the y-axis a particular training distribution, while on the x-axis the corresponding test distribution used. We then found how many times using matched distributions was better than using unmatched distributions, given the test distribution. The decision of which scenario was better depended only on which resulting out-of-sample error was smaller. In the case of binary classification simulated, the error corresponded to the fraction of points that were incorrectly

classified by the learning algorithm, given a large enough test set to estimate it. As Figure 4 shows, in more than 36% of the cases, unmatched distributions yielded better out-of-sample performance (yellow, orange, and red regions).

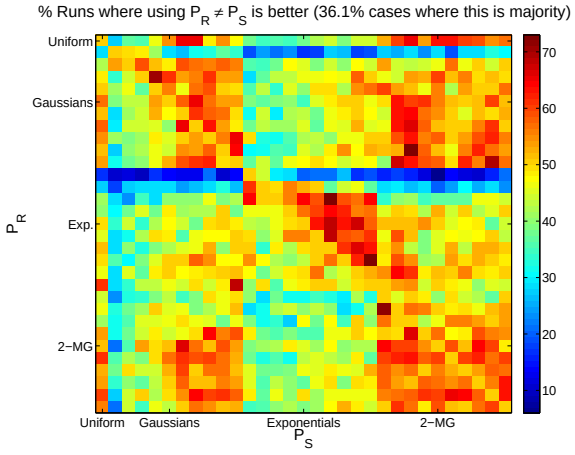

Figure 4: Monte Carlo simulation

This empirical observation can be further backed up analytically in a regression setting. In such a setting, the out-of-sample error can be computed analytically so that results like the ones shown in the Figure above can be computed without the need of a long computation time as the one required to produce the above figure. In a regression setting we are given a data set $R = \{x_i, y_i\}_{i=1}^{N}$ of ordered pairs, and we want to fit the model

$$y = \theta^T \Phi(x) + \epsilon$$

where θ is the vector of parameters to be found, Φ is a non-linear transformation of the input x, and ϵ is the noise. Furthermore, if we assume the non-linear transformation is made up of the Fourier harmonics, so that we know that any function that satisfies the Dirichlet conditions in a finite domain can be represented with such transformation, then we can find the out-of-sample error with respect to the training data set, the test set, the true target or function we want to learn, and the noise, that we denote $Eout(x, R)$.

This expression allows us to find a data set R such that $Eout(x, R) < Eout(x, R)$, if R comes from the same distribution as the test set, while R comes from a different distribution. As a concrete example, let the test distribution be a Gaussian with 0 mean and standard deviation of 0.5, and assume the domain is the interval [-1,1]. Then if we choose R from a Uniform distribution in the interval $[-a, a]$, with $a < 1$, then Figure 5 shows the error achieved. The dotted line shows the error when $R = R$, that is, when training and test distributions are matched. As it is clear, there are different values for a, that make the error smaller when training with R, which show

clearly that for the given test distribution, there is indeed a dual distribution that is not equal to the test distribution.

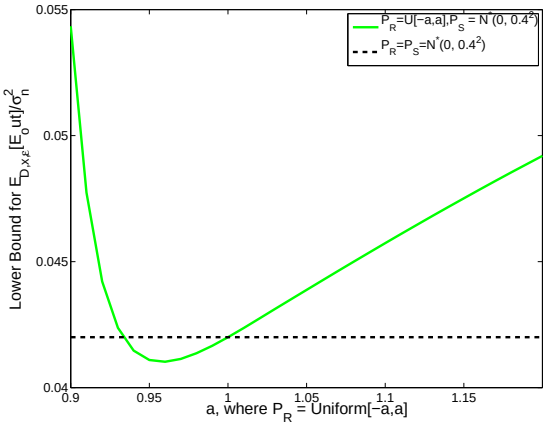

Figure 5: Out-of-sample error

Having established the existence of dual distributions, we now propose an algorithm to find these distributions. In the general case, we want to solve the following functional optimization problem:

$$P_R^\star = \arg\min_{P_R} \mathbb{E}_{R,x,f,\epsilon}[E_{out}(x,R)]$$

where $P_R^\star$ is the dual distribution.

We first solve this problem by considering a discrete input space. In such a scenario the dual distribution is no longer a function but a vector, so that we can solve the problem using standard optimization methods. In this case, the problem become

$$\begin{aligned} \arg \min_{p_1,p_2,\ldots,p_d} \quad & \mathbb{E}_{R,x,\epsilon,\theta}[E_{out}(x,R)] \\ \text{subject to} \quad & \sum_{i=1}^{d} p_i = 1 \\ & p_i \geq 0 \end{aligned}$$

where $P_R^\star = [p_1, p_2, , p_d]$ and d is the cardinality of the discrete input space. If the input space is discrete, there are only a finite number of possible data sets that can be generated if the training set is finite. In fact if the training set has N points, then the total number of possible data sets is given by

$$\sum_{i=1}^{N} \left(\begin{array}{c} d \\ i \end{array} \right)$$

Now for each possible data set we can find the corresponding out-of-sample error, which we denote

$$E_{i_1,i_2,\ldots,i_d}$$

where i_j indicates that the first element of the data set is the point j in the discrete input space. Hence, the objective function can be rewritten as a linear combination of these errors, as

$$\mathbb{E}_{R,x,\epsilon,\theta}[E_{out}(R)] = \sum_{i_1,i_2,\ldots,i_N} p_{i_1} p_{i_2} \ldots p_{i_N} E_{i_1,i_2,\ldots,i_N}$$

For the linear regression case, a closed form solution exists for each one of these errors, namely

$$\mathbb{E}_{x,\epsilon,\theta}[E_{out}(x,R)] = \sigma_N^2 \sum_{i=1}^{d} z_i^T (Z^T Z)^{-1} z_i P_S(x_i)$$

where $z = \Phi(x)$, and Z is constructed so that row i of Z is $\Phi(x_i)$, P_S is the test distribution, and ϵ is the noise. For other learning algorithms a closed form expression is not available, but these errors are estimated in the usual way. That is, using a held-out subset to test performance of the algorithm and then taking this as an estimate for out-of-sample performance.

Once these coefficients are found, the problem posed is a standard Geometric program. Geometric programs can be formulated as convex programs by using a standard transformation of the monomials in the objective function into a product of exponentials of which we can take the logarithm to recover the objective function. The monomials in our case are the products of the pi s, with their coefficients. Since the objective function is then a logarithm, and this is a convex function, and since the constraints of the program are affine functions, then this is indeed a convex program. Therefore, it can be solved using any standard convex optimization package.

Now we consider the case where the input space is continuous. In such situation, the optimization problem is a functional minimization problem, which requires the use of calculus of variations. We have not solved this problem analytically yet, but we can solve it approximately in the following way. Since in practice the test distribution is never known, it is usually estimated using the unlabeled samples given. The distribution is then estimated by binning the points, but this binning discretizes the distribution. That is, the estimate returns stepwise distributions, where we consider the densities constant in each bin.

If that is the case, then we have transformed our continuous input space into a discrete input space. Hence, the above geometric program can be solved to obtain a discrete training distribution that can then be transformed into a continuous distribution using the inverse process of the estimation. That is, starting from a discrete distribution, we assume then that in the continuous space, points in the same bin have the same density, and therefore construct in this way our continuous density.

Although this method solves the problem approximately, it is nevertheless important to find other methods to solve the continuous case directly, since the resulting Geometric program might be to large to solve, given the exponential number of terms

in the objective function, which can become intractable if the resolution used for binning is very small. For coarser binning, the program will be tractable but may not be accurate. Hence, this is the future direction of our work in dual distributions.

Once the dual distribution is found, the weighting methods like the ones proposed in Second and Third Results can be used to transform the distribution of the training set into a dual distribution. Finally, a method like the one proposed in "Deciding to weight or not to weight in a scenario with unmatched training and test distributions can be used to determine if the weights found to approximate the dual distribution are still beneficial, given the effective sample size loss that the use of weights can cause.