Learning Weights for Weighted Distance Functions

Daniel Vournazos
Department of Computer Science
Department of Mathematics
California State University, Channel Islands

Christian A. Rodríguez
Department of Computer Science
University of Puerto Rico, Río Piedras

Abstract—Some applications of machine learning often require that data be classified by defining a distance function between datapoints. Moreover sometimes these distance functions are defined using weights, or costs. Setting these weights in a meaningful way is not a trivial task. In this report we present our contribution to an existing framework [2] which learns these weights using data. We introduce parallelization to improve the runtime of this framework, and present results on how to optimize weights. We benchmarked the parallelization on a dataset whose distance matrix is created using a weighted Levenshtein distance function, and we can see that there is a noticeable decrease in the framework’s runtime.

I. INTRODUCTION

A. Motivation

Often, applications of machine learning require the definition of a distance measure, or distance function, between datapoints. In many cases these distance functions can be parameterized by a set of weights, most of the time though they are defined as unweighted or with a set of handpicked weights. Picking these weights in a meaningful way is not a trivial task, but in many cases it makes sense to introduce these weight coefficients into the distance measure as can be seen in example 1.

Example 1. Consider a dataset of strings broken into 2 classes, each string being of equal length 4. The letters of these strings are sampled from the alphabet \( \Sigma = \{a, b, c, d, e\} \) in the following manner: letters in strings of class 1 have a 95% probability of being from \( \{a, b\} \) and 5% probability of being from \( \{c, d, e\} \), letters in strings of class 2 have a 5% probability of being from \( \{a, b\} \) and 95% probability of being from \( \{c, d, e\} \). We define a distance function \( d(S_1, S_2) \) as the count of letter substitutions required to change \( S_1 \) into \( S_2 \). Now let \( S_1 = (acba) \) and \( S_2 = (baab) \) be strings in class 1 and \( S_3 = (ddhe) \) be a string in class 2. Notice that \( d(S_1, S_2) = 4, d(S_1, S_3) = 3, d(S_2, S_3) = 4 \). Classification algorithms based on this distance function will not be very efficient. Strings \( S_1 \) and \( S_3 \), even though they are in different classes, have a lower distance than strings \( S_1 \) and \( S_2 \) which are in the same class. However if we introduce a lower cost of 0.5 for the substitutions \( (a \rightarrow b) \) and \( (b \rightarrow a) \), then \( d(S_1, S_2) = 2.5 \). A classifier can now say that strings that are from the same class have lower distance between them.

In the previous example we handpicked a cost hereby known as a weight, such that the distance function helped us classify the dataset. In practice though, handpicking weights is not ideal as it is not systematic. In this report we talk about our work on improving a framework that learns these weights in a meaningful and systematic way based on data. The framework depicted in algorithm 1 works in a 2 step fashion. In the first step we calculate a distance matrix which is defined as follows: the \((i, j)\)th entry of the distance matrix is the distance between datapoints \( x_i \) and \( x_j \). In the second step we use this distance matrix to optimize the set of weights. We improve the first step by introducing parallelization to speed up the runtime of calculating the distance matrix. We also present results on one way to optimize weights.

B. Problem Definition

In this section we will formally introduce the framework, the problems it poses, and briefly describe the way we worked on these problems. First, we will define some terminology that will be used throughout this report.

Definition 1. Let \( D = \{x_1, ..., x_n\} \) be our training set of data. Our interest is in distance functions parameterized by weights \( d(\cdot, \cdot; W) \) that operate over our dataset \( D \). Using these definitions we can define a distance matrix \( K \) as a matrix that contains the pairwise distances of all points in our dataset. This is, the \( i, j \)th entry of this matrix will be \( d(x_i, x_j; W) \).

Our interest is to learn weights \( W \) that will help us classify future datapoints using the distance function \( d(\cdot, \cdot; W) \), based on our training data \( D \). We present an abstract sketch of the framework in algorithm 1. In the function \( calculateDistance(d, D, W) \) we calculate the distance matrix \( K \) using a specific set of weights \( W \). The function \( optimizeWeights(d, K, D, W) \) is used to optimize the weights as mentioned previously based on a distance matrix. It is an abstract function that can be specified by a user of the framework, however we present results on one optimization function that looks promising.

A cumbersome limitation of the framework is the fact that we need to recalculate the distance matrix \( K \) on each iteration. Depending on the computational complexity of calculating \( d \) and the size of our dataset this might limit the framework substantially. Since the entries in \( K \) are independent, we can introduce parallelization to speedup the computation of \( K \). It is not clear how to optimize weights based on a distance matrix. We present our work on an
Data: $D, d$
Result: Weights $W$

$W \leftarrow \text{rand}()$

\begin{algorithm}
\caption{Abstract Sketch of the Framework}
\begin{algorithmic}
\State $W \leftarrow \text{rand}()$
\While{} \Do
\State $K \leftarrow \text{calculateDistance}(d, D, W)$
\State $W \leftarrow \text{optimizeWeights}(d, K, D, W)$
\EndWhile
\State return $W$
\end{algorithmic}
\end{algorithm}

optimization function that modifies $W$ in such a way that datapoints in the same class have lower distance, while datapoints in different classes have higher distances. This optimization function is biased towards one of it’s terms and in our work we study ways to unbias it.

The rest of the paper is structured the following way: In section II we talk about the process of parallelizing the distance matrix computation in the framework. We talk about the parallelization scheme utilized and why it is efficient. In section III we talk about one possible optimization function. First we present the function, talk about some of it’s limitations and talk about our work on overcoming these limitations. In section IV we present our results on benchmarking the parallelization and also show a simulation ran on our optimization problem from the previous section. Finally we we conclude in section V with a summary on our report and talk about possible future work in section VI.

II. PARALLELISM AND VALIDATION

In this section we talk about parallelizing the computation of distance matrix $K$. We expand the previous psuedocode to reflect this new parallelization. Recall that $K$ is populated by a distance function $d(\cdot, \cdot; W)$ where $W$ is a set of weights. See figure 1 for an example of a simple distance matrix. This matrix is symmetric and the diagonal is always 0 because $d(x_i, x_i; W) = 0 \forall i$. The trivial way of computing the distance matrix $D$ is the following:

\begin{algorithm}
\caption{Distance Matrix calculation}
\begin{algorithmic}
\For{}{i = 1 \to n}
\For{}{j = i + 1 \to n}
\State $K_{i,j} = d(x_i, x_j; W)$
\State $K_{j,i} = K_{i,j}$
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|}
\hline
  & s1 & s2 & s3 \\
\hline
s1 & 0 & 2 & 3 \\
s2 & 2 & 0 & 1 \\
s3 & 3 & 1 & 0 \\
s4 & 8 & 5 & 4 \\
\hline
\end{tabular}
\caption{Example of a distance matrix}
\end{table}

A. C program

Our first objective is to translate the existing calculateDistance code into a format that can be parallelized. To do this a C program is created that imitates the existing code written in mex [4]. The existing code uses matlab’s API which is not available for normal compilation. Thus it cannot be run by itself, the correct environment is needed to pass it input and recive the output. The C program mirrors the existing matlab program but with a few differences. Because it cannot be passed to, nor can it directly return data objects like matlab can, it must have some IO operations to deal with retrieving and later distributing input and output. After this code is created it is important to validate the correctness of the new implementation.

B. Validation

The validation consists of two processes. The first asserts that the code runs correctly, and the second verifies that the jobs are divided and recombined correctly.

1) Code Translation: The first step is to create inputs for our program in the specified format. Because this is sent along with the code in the parallelization step, the input will contain a location for the data instead of the data itself. This avoids sending huge amounts of data potentially hundreds of times.

To verify the code translation the Levenshtein distance function [1] was used. Initially, random strings are generated which will be used as the dataset. Our input is formatted in the following way: The number of arrays on the first line. After this there is a repeating segment that consists of how many rows a data point will have as well as the data point itself. The input files will have the path to the data specified and flags for utilizing the weight matrix and printing out the paths. This weight matrix is generated in matlab based off the largest number in the dataset previously created, this allows the matrix to be returned to matlab for future use as well as written to the input file for the C program’s use.

Our C program can’t return an object to matlab directly to be used in comparison with the existing code, so an object is created afterwards from the output. In the output the size
<table>
<thead>
<tr>
<th></th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>s2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>s3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>s4</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 2. Highlighted are an example of rows calculated in a job of the distance matrix and the indices are calculated. The distance matrix is printed out as well as histograms for inner class operations and outer class operations along a string path. They record the normalized number of how many times a specific transformation is used along paths with strings in different classes as well as strings in the same class. These outputs can be read to create a structure that allows the output of the C code to be compared to the output of the existing code.

2) Code Parallelization: The next step configures the new program for parallelization and collects each output to recombine them into one whole output. We want to compute each entry of the distance matrix in parallel. Since the distance matrix consists of pairwise distances between strings, it is a symmetric matrix with entries along the diagonal zero. This means that only the upper triangulation of the matrix needs to be calculated. By creating jobs that compute the $ith$ and $n-ith$ rows of $K$, we can create jobs where loads are evenly distributed as seen in Figure 2.

Now to tell our workers to calculate only these rows, the input is modified to specify indices to calculate. Since we have partial outputs created by calculating specific rows for a job, we need to combine them into a whole output for testing against the code that already works. The outputs are read and structures with fields: distance matrix dimension, indices, distance matrix itself, outer class histogram, and inner class histogram are created. These are then combined into a structure similar to one created from calculating all the entries of the matrix, except this contains all the data. After checking to make sure this structure has the same data in each field as if running the data without jobs, we can move along to actually running the code in parallel.

C. Parallelization

To parallelize the code the Declarative Load framework [3] previously created for jobs such as this is used. There are 3 parts to this parallelization framework.

* Dispatcher: This is what gathers together the jobs and sends them to the server. It also pings statistics on it’s jobs.

* Server: This sends jobs to workers as well as collects outputs and sends information to the dispatcher.

* Worker: This takes the jobs from the server, executes them, and sends output back to the server.

The server is set up in such a way as to balance job execution, which can be read more about in the cited paper.

III. Optimizing Weights

In this section we present our work on a specific optimizeWeights function from algorithm 1. First we will describe how we modify weights such that data points are correctly classified. Then turn this intuition into an optimization problem [2], state why this optimization problem is biased and present one way to unbias this optimization problem in subsection III-A. This unbiasing however requires that we solve for some constants. In subsection III-B we show results on solving for these unbiasing constants under some assumptions and present our hypothesis on the results while dropping the assumptions.

A. Defining an objective function

Recall that we are interested in the classification of data points, by using a weighted distance function $d$. For a distance matrix $K$ we refer to an entry of $K$ corresponding to data points in the same class as intraclass distances and
we refer to entries corresponding to datapoints in different classes as interclass distances. The optimization function proposed \cite{2} is a heuristic that attempts to adjusts the weights \(W\) such that intraclass distances are minimized and interclass distances are maximized. We now define the optimization problem:

\[
\arg\min_W \left\{ \sum_{l(x_1)=l(x_2)} d(x_1, x_2; W) - \sum_{l(x_1)\neq l(x_2)} d(x_1, x_2; W) \right\} \tag{1}
\]

The optimization problem 1 suffers from a biasing problem as for any dataset with more than 2 classes there are more interclass distances than there are intraclass distances. This is a problem as one side of our objective function will dominate the other and this will lead to a bad choice of \(W\) that maximizes interclass distances but fails to minimize intraclass distances. To solve this problem and unbias our objective function we need to scale down the interclass distances sum by the appropriate term.

Note that if we have a balanced dataset of \(k\) classes, where each class has the same amount of datapoints, then the ratio between pairs of distances between datapoints in the same class versus pairs of distances between datapoints in different classes is \(1:k-1\). Scaling down the sum of interclass by \(k-1\) will lead to unbiassing our objective function. We are interested however in the general case of datasets with unbalanced classes that have a different amount of datapoints per class. In this case, we add unbiassing constants to the objective function 1, in order to account for the imbalance of class sizes. Our objective function \(X\) now becomes:

\[
X = \sum_{l(x_1)=l(x_2)=i} \alpha_i \cdot d(x_1, x_2; W) - \sum_{i,l(x_1)\neq l(x_2)=j} \beta_{i,j} \cdot d(x_1, x_2; W) \tag{2}
\]

B. Solving for unbiassing constants \(\alpha_i\) and \(\beta_{i,j}\)

Our problem now consists of finding \(\alpha_i\) and \(\beta_{i,j}\) such that both intraclass and interclass distance are treated equally by our optimization function. To study possible solutions to this problem, we define \(W\) as a random variable. If \(X\) is our objective function, then \(X\) is now a random variable under the process of picking set of random weights \(W\). Unbiassing objective function 2 requires that both the sum of intraclass distances and the sum of interclass distances be equal on average. Therefore, we require that the expected value of both sides of the sum are equal. From this requirement we obtain the following equation:

\[
\mathbb{E}[X] = \sum_{l(x_1)=l(x_2)=i} \alpha_i \cdot \mathbb{E}[d(x_1, x_2; W)] - \sum_{i,l(x_1)\neq l(x_2)=j} \beta_{i,j} \cdot \mathbb{E}[d(x_1, x_2; W)] = 0 \tag{3}
\]

To study \(X\) as a random variable under this requirements consider the simpler case where the entries of our distance matrix \(K\) are IID random variables. Under this assumption \(\mathbb{E}[d(x_1, x_2; W)]\) is known for any \(x_1\) and \(x_2\). The only unknown variables in equation 3 are \(\alpha_i\) and \(\beta_{i,j}\). If we have \(k\) classes then equation 3 has \(k^2-k\) variables, implying that it has infinite solutions for \(k>1\). We define \(\theta_i = \{\alpha_i, \beta_{i,j}\}_{i,j=1}^k\) as the \(i\)th solution for equation 3 and \(X_i\) the random variable defined by \(\theta_i\). For this case we have proven a theorem that says any two solutions \(\theta_1\) and \(\theta_2\) define random variables \(X_1\) and \(X_2\) with the same distributions up to a known constant.

**Theorem 1.** Let \(K_{n\times n}\) be a matrix such that \(K_{i,i} = 0\) \(\forall i\), and \(K_{i,j}\) are IID random variables \(\forall i,j\). Let \(\theta_1 = \{\alpha_{1,i}, \beta_{1,i,j}\}_{i,j=1}^k\) and \(\theta_2 = \{\alpha_{2,i}, \beta_{2,i,j}\}_{i,j=1}^k\) be two solutions for Equation 3 defining two random variables \(X_1\) and \(X_2\), respectively. Then, there exists a constant \(C \in \mathbb{R}\) such that the probability density of \(X_1\) approaches the probability density of \((C \cdot X_2)\) as \(n\) approaches \(\infty\).

**Proof.** Let \(K, \theta_1\) and \(\theta_2\) be the random distance matrix and the two solutions from the theorem. Since (non-diagonal) entries of \(K\) are IID random variables, \(X_1\) is defined, using \(\theta_1\), as follows

\[
X_1 = \sum_q \alpha_{1,q} \cdot K_{d_1,d_2} - \sum_q \beta_{1,q} \cdot K_{d_1,d_2}
\]

and similarly, \(X_2\) defined using \(\theta_2\).

Since (non-diagonal) entries of \(K\) are IID random variables, by the Central Limit Theorem, distributions of both \(X_1\) and \(X_2\) approach normal random distributions, \(N(\mu_1, \sigma_1^2)\) and \(N(\mu_2, \sigma_2^2)\) as \(n\) approaches infinity, respectively. It is given that \(\theta_1\) and \(\theta_2\) are solutions to Equation 3. From that, we know that \(\mu_1 = \mathbb{E}[X_1] = 0\) and \(\mu_2 = \mathbb{E}[X_2] = 0\) for any \(C \in \mathbb{R}\).

If we choose \(C = \frac{\sigma_2}{\sigma_1}\), then \(\text{var}(C \cdot X_2) = \sigma_1\). Clearly, \(\mathbb{E}[C \cdot X_2] = 0\) and the distribution of \((C \cdot X_2)\) remains normal. Subsequently, we get \(X_1 \sim N(0, \sigma_1^2)\) and \((C \cdot X_2) \sim N(0, \sigma_2^2)\) as \(n \to \infty\). \(\square\)

From theorem 1 we know that if one has \(\theta = \{\alpha_i, \beta_{i,j}\}_{i,j=1}^k\) a solution to equation 3 any other solution will be equivalent. We solve for one solution \(\theta\) as follows: First we set \(\alpha_1 = 1 \forall i\), next we assume that \(\beta_{i,j} = \beta \forall i,j\), transforming equation 3 into an equation of one variable:

\[
\mathbb{E}[X] = \sum_{l(x_1)=l(x_2)=i} \mathbb{E}[d(x_1, x_2; W)] - \beta \cdot \sum_{i,l(x_1)\neq l(x_2)=j} \mathbb{E}[d(x_1, x_2; W)] = 0 \tag{4}
\]

Solving equation 4 for \(\beta\) gives us \(\beta = \frac{\sigma_1^2}{\sum_i n_i^2 - n_i n_j}\) where \(n_i\) denotes the number datapoints in class \(i\).

Based on numerical studies shown in our Evaluation we do not expect that our theorem still holds when we drop the assumptions of \(K\) being composed of IID random variables. Therefore we need to find ways to relax theorem 1 in this case.
Fig. 4. Execution time compared to number of jobs

Fig. 5. Execution speedup factor with 500 jobs

IV. EVALUATION

Here we will show the evaluation of our contribution to this framework. First we will show the benchmarking results on parallelizing the computation of the distance matrix $K$. This will show us the speedup gained by introducing parallelization. Then we will show simulations of our objective function $X$ that provide insight and intuition on the way we unbias our objective function. We also present a simulation of $X$ using a distance function that has dependencies, and see, intuitively, that our theorem does not hold once we drop our assumptions.

A. Parallelization

- 6 Machines
- OS: CentOS 6.4 Kernel version 2.6.32-358
- CPU: i7-2620M (2.7GHz, 1333MHz FSB, 4MB L3 cache), 4 cores/8 threads
- RAM: 8GB DDR3 - 1333MHz
- Filesystem: ramfs - 1GB, no paging occurred
- Network: 10/100M Ethernet connection (Workers connected to server over the internet)
- 4 workers per machine, each ran on a distinct CPU core

Running the C code without jobs produced a runtime of about 38 seconds. As evident in figure 5 we can see that there is linear speedup as more workers are added to the job computation. The speedup is calculated by taking the average computation time using 1 worker and dividing it by the average time taken with $n$ workers. This shows the overall impact of increasing the amount of workers has on the total execution time. From figure 4 one can see that using the pushCompute framework increases the runtime when utilizing only 1 worker. There are two reasons for this. The first being that there is some inherent overhead in dealing with the pushCompute framework. The second, bigger reason, is that when the jobs are split as mentioned above. There are about 500 jobs to execute. Each of these requires reading in that data file, meaning those 1000 strings are being read 500 times as opposed to one time. This presents some serious time increase. To verify this as well as see the effect reducing the amount of reading has, we create 250 jobs and 50 jobs from the 1000 strings. These still have the same row balancing mentioned previously. Now we see a significant decrease in starting runtime. The best being when 50 jobs are created. But making smaller number of jobs only works if there is enough workers to make use of the job division. It would be useless to create 15 jobs for 24 workers.

B. Optimizing Weights

In the section III we provided one set of parameters for our objective function 2: $\alpha_i = 1 \forall i$ and $\beta = \frac{\sum_{i \neq j} n_i n_j}{\sum_i n_i^2}$. We now present a simulation where we generated 10000 random variables $X$ in the following manner: First we fixed 4 classes each with 10 elements and 1 class with 40 elements. We solved for $\beta$ using these numbers. Now in each iteration we generated a distance matrix $K_{80x80}$ using random variables from the same distribution and assigned each datapoint to a class. We then computed $X$ using this $K$ as our distance matrix and saved the result in a vector. We normalized the entries in this vector by it’s standard deviation and plotted a histogram of it on top of a fitted normal distribution in Fig.
6. $X$ is a normal distribution with $\mu = 0$ in this simulation which implies that our objective function is unbiased and on average the sum of interclass distances is equal to the sum of interclass distances. We do not expect our theorem to hold using distance functions that introduce dependencies between datapoints. To study this hypothesis we ran another simulation using a weighted Levenshtein Distance [1] as our distance function $d(\cdot, \cdot; W)$ and a dataset of 120 strings generated in a smart fashion. We removed the parameters on $X$ turning equation 2 into:

$$X = \sum_{l(x_1)=l(x_2)=i} x_{1}, x_{2} \in D \quad d(x_{1}, x_{2}; W) - \sum_{i=l(x_1)\neq l(x_2)=j} x_{1}, x_{2} \in D \quad d(x_{1}, x_{2}; W)$$

The simulation ran 1000 iterations as follows: In each iteration we generated a random set of weights $W^*$. We calculated equation 5 using weights $W^*$ and saved the result in a vector. With these results we verified out theorem in the following fashion: We first introduced our previous set of parameters $\theta_1$ and plotted a histogram of the results vector. Also we solved for a new set of parameters $\theta_2$ by setting $\alpha_i = 1 \forall i$ and then empirically solving for $\beta$, we plotted a histogram of this solution as well.

As can be seen in 7, when using the solution $\theta_1$ the mean of our simulation is not 0, whereas for $\theta_2$ the mean of our simulation is 0. It is obvious from these results that when we let $d(\cdot, \cdot; W)$ be any distance function, our previous solution for $\beta$ does not satisfy our requirement of $E[X] = 0$. Moreover future work must be done on how to symbolically solve for $\theta$ in this general case.

V. CONCLUSION

In this work we saw a class of classification problems based on weighted distances, and the cumbersome task of handpicking weights for these functions. We presented our work on improving a framework that learns weights based on data. After introducing the framework and two of it’s challenges: high cost of calculating the distance matrix in every iteration, and picking an optimization function that will learn weights in a correct manner, we presented our work on both of these problems.Introducing parallelization into the computation of the distance matrix decreased the runtime of the framework substantially, and we presented an improvement to a specific way to optimize weights.

Translation of existing distance matrix computation code into C allowed us to parallelize and although there was some runtime added due to overhead, as we introduced more workers the runtime taken for computation decreased into a fraction of the original runtime. We present our work on one optimization problem that heuristically optimizes weights in a smart fusion. We saw that it had a biased objective function, and presented one way to unbiase the objective function by solving for a set of constants. Also under some assumptions we showed that all solutions for unbiasing the objective function are equivalent up to a constant.

VI. FUTURE WORK

When we optimize weights more work needs to be done on unbiasing the objective function when the distance between datapoints is not independent. This will allow our framework to work with a very broad range of distance functions. It is also of interest to find other objective functions that encompass our optimization problem and study them.

ACKNOWLEDGMENT

This work was supported in part by TRUST, Team for Research in Ubiquitous Secure Technology, which receives support from the National Science Foundation (NSF award number CCF-0424422)

We would also like to acknowledge:
- Daniel Aranki
- Ferda Ofli
- Aimee Tabor
- Mary Stewart
REFERENCES


