Classification using Parallel Locality Sensitive Hashing Data Structures

Final Project for CS 598 Geometric Data Structures

Christian Howard
howard28@illinois.edu

Abstract

Within the world of Machine Learning, being able to look at an input and decide what to classify it as fits within many application domains such as computer vision, signal processing, financial signals, and more. With the number of classification problem domains being extensive, building algorithms to handle such problems efficiently is an interesting area to consider. Within this project, a classification code is built around the idea of k-nearest neighbor (k-NN) classifiers by using approximate k-nearest neighbor data structures based on Locality Sensitive Hashing (LSH). This work builds out the classical LSH variant as well as the CoveringLSH variant which removes the ability for false negatives. Parallel implementations of these data structures are built in C++14 using a threading approach via OpenMP with good scaling performance. These data structures are then applied to a sample classification problem to observe the accuracy of these approximation based k-NN classifiers.

Contents

1 Problem Outline

1.1 Multi-label Classification

From a young age, every person out there learns to navigate through the world by first learning how to classify things they see around them. Classification, that is being able to identify what an object is, is a crucial part of how we in turn interact with the world around us. Of course, being able to identify objects generalizes past what humans can see with their senses but indeed finds its way into all sorts of automation and algorithmic related tasks. Whether it is identifying objects within images or identifying behavior on a server indicative of a hacker, the range of areas where such techniques could be useful are extensive.

To treat this problem mathematically, a simple way to describe the problem of classification is to start off with a given dataset $\mathcal{D} := \{(x_i, y_i)\}_{i=1}^n$ with some metric $d(\cdot,\cdot)$ where $x_i \in \mathbb{R}^d$ is a point representing some information about an object and $y_i \in L$ is the corresponding label for the object, with L being some finite set of labels. Using this dataset, we wish to come up with some predictor that, when given some point $x \in \mathbb{R}^d$, can find a label for x that makes sense, given the dataset $\mathcal{D}$. A specific choice for the predictor that makes sense is

$$\hat{f}(x) = \arg\max_{y \in L} P(Y = y | X = x, D = \mathcal{D})$$

which means to make the predictor choose the most likely label, conditioned on the dataset and input x. How one aims to construct this sort of predictor, however, leads to different techniques, one of which we will mention as it relates to *similarity*.

1.2 Classification through Similarity

As mentioned previously, given some dataset $\mathcal{D} := \{(x_i, y_i)\}_{i=1}^n$, we can construct a classifier that makes predictions via the predictor

$$\hat{f}(x) = \arg\max_{y \in L} P(Y = y | X = x, D = \mathcal{D})$$

Of course, one can relax this model by approximating the probability $P(Y|X, D)$ in some manner. One simple way to think about approximating this probability is to look at all points from $\mathcal{D}$ within the ball $B(x, \epsilon)$ (induced by the choice of metric) and approximate the probability as

$$P(Y = y|X = x, D = \mathcal{D}) \approx \frac{|\{(x_i, y_i) \in \mathcal{D} \mid y_i = y, x_i \in B(x, \epsilon)\}|}{|\{(x_i, y_i) \in \mathcal{D} \mid x_i \in B(x, \epsilon)\}|}$$

Interestingly, since the denominator of this approximation does not depend on the input value y, we can reduce our predictor to a simplified form that just counts the number of data points that are *similar* (ϵ-close with respect to the metric) to x that have the label y and then chooses the label that has the highest count. More precisely, this gives us for some fixed $\epsilon > 0$ that

$$\hat{f}(x) = \arg\max_{y \in L} |\{(x_i, y_i) \in \mathcal{D} \mid y_i = y, x_i \in B(x, \epsilon)\}|$$

Now suppose that given $x \in \mathbb{R}^d$, $\exists \epsilon_x > 0$ such that exactly k points of $\mathcal{D}$ intersect $B(x, \epsilon_x)$. If we use this as our ϵ, which means it carries a dependence on the input x, we can instead choose our predictor such that it looks at the k-nearest neighbors to x and chooses the label that occurs most often within this set of points. Thus, this gives us a *k-nearest neighbor (k-NN) classifier*, a classifier we aim to build within this project.

1.3 Problem Instance for Project

To tackle classification using a k-NN classifier, we ultimately depend on our metric and in turn that may impact the sort of data we can work with. For the purposes of this project, we will work with the Hamming metric $d_H(u, v)$ that takes as input two bit vectors $u, v \in \{0, 1\}^d$ and effectively counts the number of bits that differ between u and v. We will make use of the ability for a finite input space $[U]^d = \{0, 1, \cdots, U\}^d$ to be transformed into higher dimensional Hamming space where l_1 distances in the original space are equivalent to Hamming distances in the new space for corresponding vectors.

Specifically, for some vector $(u_1, \cdots, u_d) \in [U]^d$, its corresponding bit vector is $\text{bitvec}(u) = 1^{u_1}0^{U-u_1}1^{u_2}0^{U-u_2} \cdots 1^{u_d}0^{U-u_d}$, where 1^p corresponds to p continuous 1 bits and similarly for 0^p. This implies that a vector $u \in [U]^d$ is mapped to a unique bit vector in $\{0, 1\}^{dU}$ Hamming space. Notice that this transformation does indeed ensure that the l_1 metric in the original space is equivalent to the Hamming space distance of the transformed result. To see this, note that the l_1 norm definition we have that for $u \in [U]^d$ that

$$\|u\|_1 := \sum_{i=1}^{d} |u_i|$$

with the l_1 metric being $d_1(u,v) = \|u-v\|_1$. If we take the transformed result as defined before, we have that the Hamming norm for the bit vector associated with $u \in [U]^d$ is

$$\begin{aligned} \|\text{bitvec}(u)\|_H &= |\{i : \text{bitvec}(u)_i = 1\}| \\ &= |\{i : (1^{u_1}0^{U-u_1}1^{u_2}0^{U-u_2}\cdots 1^{u_d}0^{U-u_d})_i = 1| \\ &= \sum_{i=1}^{d} u_i \\ &= \sum_{i=1}^{d} |u_i| \\ &= \|u\|_1 \end{aligned}$$

The above implies that our transformation conserves distances and so in turn allows us to focus on approaches to solving the k-NN problem in Hamming space datasets.

2 Algorithms and Software

2.1 Exact Algorithms for k-NN Classification

Given the description for the k-NN classifier described before, it seems the main algorithmic hurdle that needs to occur is actually obtaining the k-nearest neighbors, given the dataset $\mathcal{D}$ and input point x. The most basic algorithm one can think of is to compute the distances $(d(x,x_1),\cdots,d(x,x_n))$, sort this list from smallest to largest in $O(n\log n)$ time, and then pick the first k samples from the list and use their information to get the label the predictor will return in an overall complexity of $O(dn\log n+k) = O(dn\log n)$, since $k \leq n$ and assuming $d(x,x_i)$ can be computed in $O(d)$ for any i. This basic algorithm is a common first implementation since it is simple, but suffers a lot when the dataset size is large.

An improvement would be to push points into a max-heap data structure that maintains its structure using the distances $d(x, x_i)$ and whenever the heap has a size of $k+1$, we pop off the point with the maximum distance from x at that moment. This algorithm in turn allows us, using typical $O(\log m)$ updates for a heap with m elements, to find the k-nearest neighbors in $O(dn \log k)$ time. While this is better than the previous result, this still feels too slow for large datasets since you still must check every data point before getting the k-nearest neighbors. If we decide to use a kd-tree to store the points and allow for k-nearest neighbor queries, we can perform such queries in $O(n^{1-1/d} + k)$ time, which provides a sublinear result in n, a key improvement over the other techniques, but has diminishing returns for higher dimensions. To get something that is sublinear without a dependence on d, we will instead allow for approximately finding the k-nearest neighbors for some point x.

2.2 Approximate Algorithms for k-NN Classification

Within this discussion, readers can visit [Pagh16][1] to get a more thorough discussion of the details. Now to go about approximately tackling the k-nearest neighbor problem, we first consider a reduction of exact k-nearest neighbor problem to something simpler. Problem 2.1 refers to the approximate k-nearest neighbor problem of interest and Problem 2.2 refers to a problem the k-nearest neighbor problem can be reduced to. We can reduce Problem 2.2 to Problem 2.1 by building data structures that tackle Problem 2.2 for a range of radii we care about and querying the data structure from the smallest radii to the largest, compiling their resulting outputs of points until we have a set of k distinct points.

[1][Pagh16] paper link: https://arxiv.org/pdf/1507.03225.pdf

Problem 2.1: c-approximate k Nearest Neighbors

Assume a d-dimensional metric space (X, d_x). Given point set $S \subseteq X$ of size $|S| = n$, we wish to construct an efficient data structure that when given a query point $q \in X$, the data structure returns up to k points $\mathcal{P} \subseteq S$ such that $d_X(q, p) \leq cr$ for all $p \in \mathcal{P}$ where r is minimum distance that ensures $|B(x, r) \bigcap S| = k$.

This reduction allows us to gain an approximate k-nearest neighbor solution using the data structure that solves Problem 2.2, while making the space and time complexity only a multiplicative factor based on the number of radii we choose to use. If we ensure we use only a polylog(n) number of radii, any sublinear query solution Problem 2.2 will remain sublinear while also not contributing more than a polylog(n) factor to the space complexity. Thus, we need to only consider data structures for handling Problem 2.2.

Problem 2.2: c-approximate k r-near Neighbors

Assume a d-dimensional metric space (X, d_x). Given point set $S \subseteq X$ of size $|S| = n$ and fixed $r > 0$, we wish to construct an efficient data structure that when given a query point $q \in X$, the data structure returns up to k points $\mathcal{P} \subseteq S$ such that $d_X(q, p) \leq cr$ for all $p \in \mathcal{P}$, returning INCOMPLETE if less than k points are returned.

For the purposes of this project, two data structures were considered to solve Problem 2.2 given a dataset in some d-dimensional Hamming space: *Classical Locality Sensitive Hashing (LSH)* and *CoveringLSH*. These two data structures make use of Locality Sensitive Hashing to efficiently map a query point to a bucket of points that should, generally speaking, have some points that are nearby. In the context of LSH, we define a false negative query result as one where we query for points within a distance cr away from the query point x and get back that there are no points within that distance away from x, even though there exists at least one in the dataset that does satisfy the distance requirement. One of the key differences is that the Classical LSH approach allows for false negatives while CoveringLSH does not.

The idea behind the Classical LSH is to define a hash function family $\mathcal{H}_{\mathcal{A}} := \{h(x) = x \wedge a \mid a \in \mathcal{A}\}$ induced by a random subset $\mathcal{A} \subseteq \{0, 1\}^d$ of

L i.i.d. bit vectors and then construct a hash table using the dataset and each hash function within the family. To query for our k r-near neighbors, we loop though the L hash tables checking the bucket our input x is hashed to in search of at most k points within distance cr away from x. For large enough L, we have with high probability that the collisions with an input x will cover k points within a distance cr away from x, if such points exist, with a small false negative rate.

For the CoveringLSH, the idea is to similarly define a hash function family as $\mathcal{H}_{\mathcal{A}(m)} := \{h(x) = x \wedge a \mid a \in \mathcal{A}(m)\}$ where m is some function used to construct correlated vectors within $\mathcal{A}(m)$ such that $\mathcal{H}_{\mathcal{A}(m)}$ is r-covering, as defined in Definition 2.1.

Definition 2.1: r-covering Function Family in Hamming Space

For $\mathcal{A} \subseteq \{0,1\}^d$, the hash function family $\mathcal{H}_{\mathcal{A}}$ is r-covering if for any $x \in \{0,1\}^d$ with $\|x\|_H \leq r$, $\exists h \in \mathcal{H}_{\mathcal{A}}$ such that $h(x) = \mathbf{0}$.

An r-covering construction is shown in work by [Pagh16][2] to exist for any choice of $m : \{1, \cdots, d\} \mapsto \{0,1\}^{r+1}$. For simplicity we randomly construct m and use this to construct our hash function family as defined by

$$\mathcal{H}_{\mathcal{A}(m)} := \left\{h(x) = x \wedge a(v) \mid a_i(v) = \langle m(i), v\rangle \text{ for } v \in \{0,1\}^{r+1} \setminus \{\mathbf{0}\}\right\}$$

Then, as with the Classical LSH approach, we construct hash tables for each hash function in the family $\mathcal{H}_{\mathcal{A}(m)}$ and query the data structure by checking all hash tables until we retrieve exactly k distinct points that are within a distance of cr away from the input x, given they exist. This CoveringLSH construction ensures by the choice of our hash function family that there will be no false negatives with probability 1 at the expense of requiring $2^{r+1} - 1$ hash functions for a fixed r.

Thus, if we choose $L = \Theta(n^{1/c})$ as is used in the software implementation, then the Classical LSH technique requires $O(Ln) = O(n^{1+1/c})$ space and a worst case $O(dL) = O(dn^{1/c})$ time complexity for querying. For CoveringLSH, we have that the space is $O(2^r n)$ and the query time is in the worst case $O(d2^r)$. For small constant values of r, we in turn have an $O(n)$

[2][Pagh16] paper link: https://arxiv.org/pdf/1507.03225.pdf

space and $O(d)$ worst case query. If we allow for $r \leq \log(n)/c$, then we also have $O(n^{1+1/c})$ space and $O(dn^{1/c})$ query time complexity, matching up to the Classical LSH approach.

2.3 Software Implementation

The software implementation, located on Github[3], of the approximate k-nearest neighbor data structures and the related k-NN classification algorithms is in C++14 and makes use of OpenMP to perform shared memory threading to boost the efficiency of the construction and querying of the various LSH data structures. The LSH data structure variants are built to support input datasets defined within some finite space $[U]^d$ and automatically transforms them into a Hamming space representation using the transformation discussed earlier. This representation is supported through a custom bit vector class that minimizes space by actually compressing bits of the representation into the bits of a list of unsigned integers and defines methods to perform various key computations, reads, and writes on the underlying bit vector data.

3 Experimental Results

3.1 Hardware Setup

The hardware setup for performing the experiments is a 2.3 GHz 8-core Intel Core i9 CPU along with 16 GB 2667 MHz DDR4 RAM. Software is compiled using the latest Clang C++ compiler with OpenMP support and compiled with the fastest and smallest executable setting.

3.2 Parallel Scalability of Data Structures

To test the parallel implementation scalability for the ClassicalLSH and CoveringLSH data structures, we consider a strong scaling analysis by solving two approximate k r-near neighbors test problems for a different number of processors. The first problem will be with $r = k = 5$, $U = 8192$, $d = 1$, an approximation factor of $c = 2$, and the dataset exhaustively covering the

[3]Codebase link: https://github.com/choward1491/classiferLSH

whole space of possible values, implying that $|\mathcal{D}| = U + 1$. The second problem will be with $r = k = 10$, and the rest of the problem defined similarly to problem one.

Under these examples, Figure 1 showcase the raw change in time to build the data structures for each problem and 2 showcases the strong scaling results as a function of the number of threads used. The strong scaling results show that up through 8 threads, the scaling behaves pretty well being between 85% and 100% scaling. When the number of threads is set to 16, to take advantage of having two hardware threads per each of the 8 cores, the strong scaling performance drops a bit, which is unsurprising. The raw results in Figure 1 show that even at the 16 thread case, the runtimes are about as good as the 8 thread case, which implies it does not hurt performance but that number of threads is unnecessary. One further comment is that threading is used to handle queries as well, but it was found that queries tended to be so fast that threading did not really buy much relative to the overhead of setting up threads, with each k r-near neighbor query taking on an order of 10 to 100 microseconds.

3.3 Classification Experiment

The classification problem we will tackle is a 2D binary classification problem with a l_1 norm induced disk in the center of the domain being labeled 1 and the surrounding region being labeled 0. This test case will use a universe size of $U = 256$ and the label set will be $L = \{0, 1\}$. We will generate a corresponding dataset with a random sample of size $\lceil |[U]^2|/6 \rceil = 10923$ pieces of data, labeled based on the disk configuration. Figures 3 and 4 represent visuals for the classification results for the ClassicalLSH and CoveringLSH approaches using a nearest neighbor set size of $k = 5$ and a radii set of $\{1, 2, \cdots, 6\}$, where black regions represent areas where each k-NN models mislabeled the points. Just eye-balling the results, it appears the ClassicalLSH and CoveringLSH based k-NN models perform similarly. Looking at the data, the ClassicalLSH based k-NN model mislabeled 0.53% of the test data while the CoveringLSH based k-NN model mislabeled 0.54% of the data. Both models did indeed perform similarly and quite well, though interestingly the ClassicalLSH based k-NN model performed slightly better.

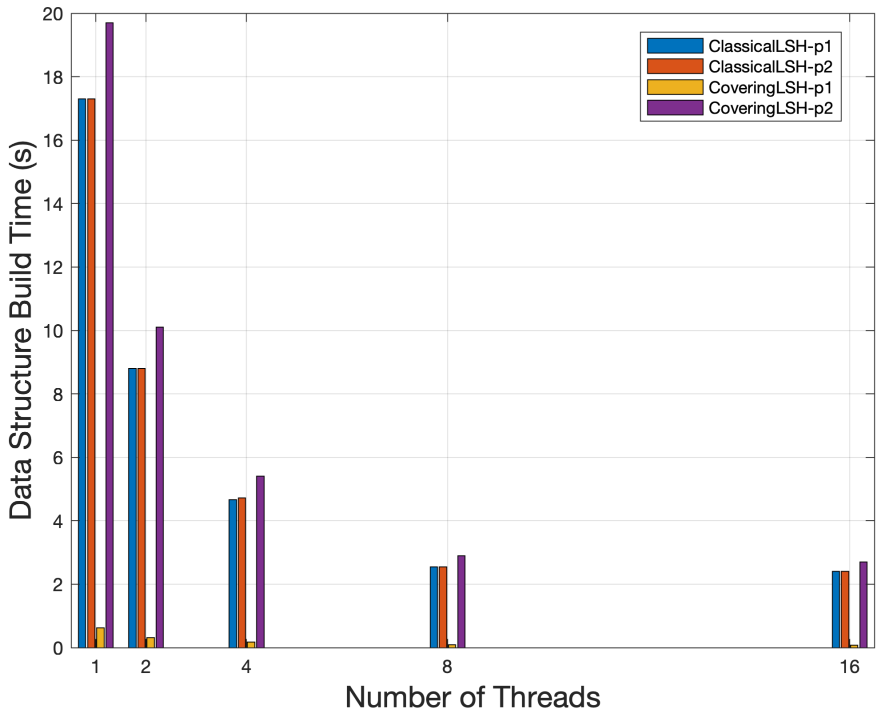

Figure 1: Raw build times as number of threads increase

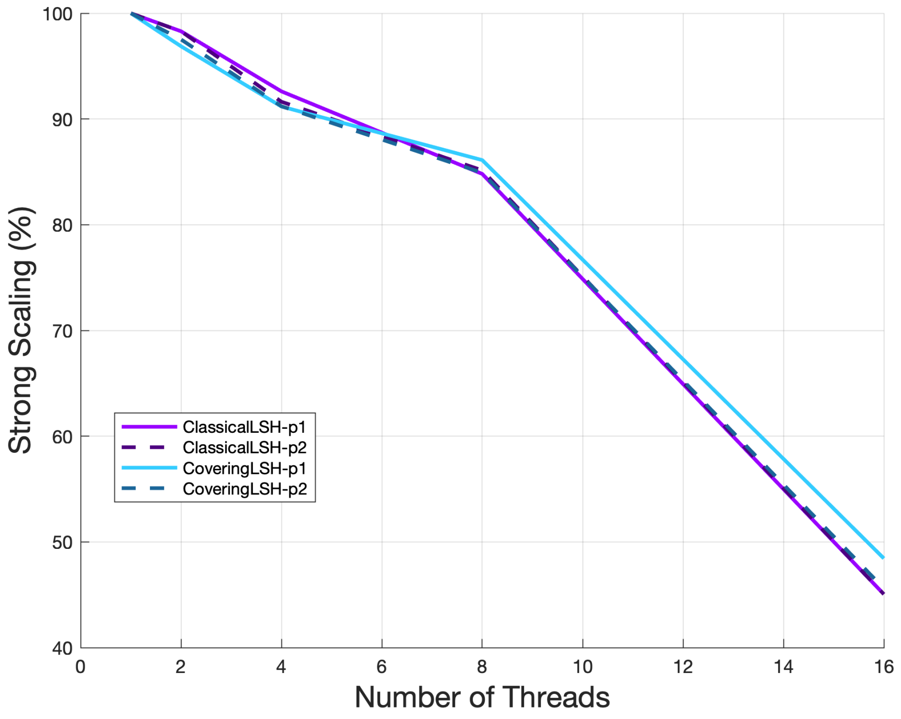

Figure 2: Strong scaling of data structures

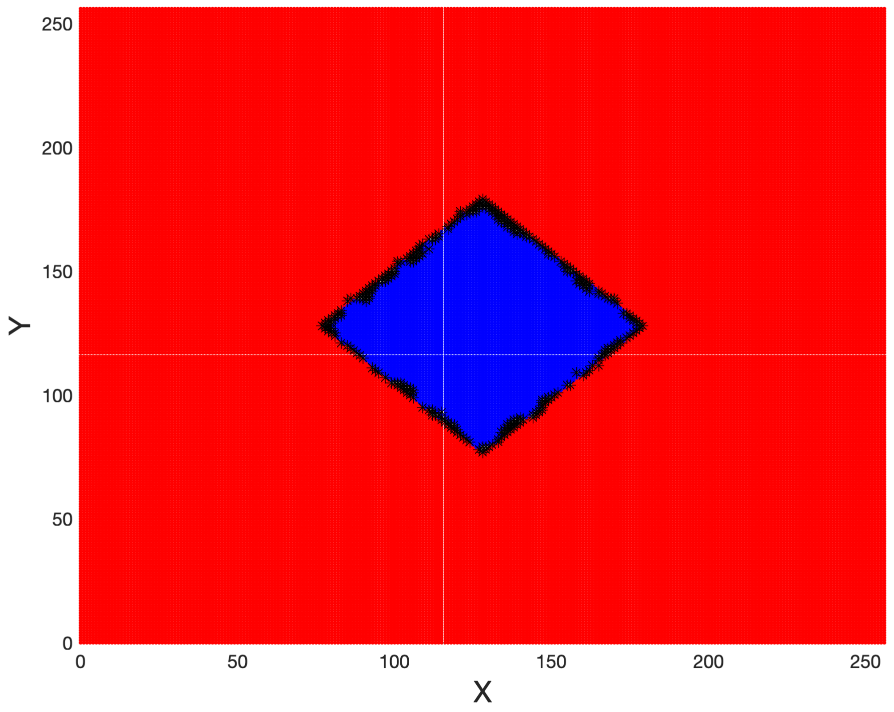

Figure 3: Classification Results using ClassicalLSH k-NN algorithm. **Red** corresponds to a correct label of 0, **Blue** corresponds to a correct label of 1, and **Black** corresponds to incorrect labeling.

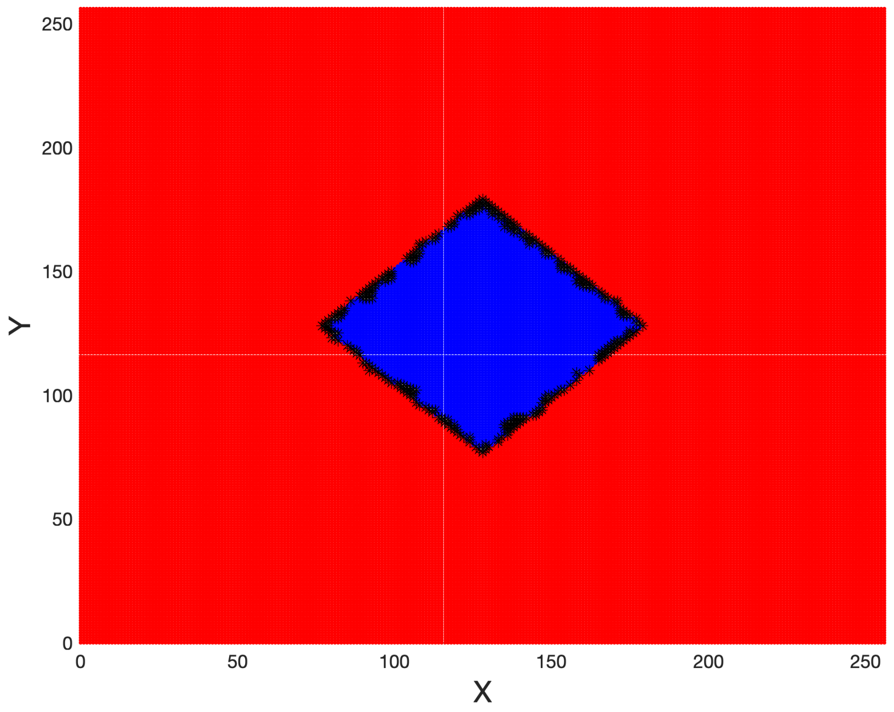

Figure 4: Classification Results using CoveringLSH k-NN algorithm. **Red** corresponds to a correct label of 0, **Blue** corresponds to a correct label of 1, and **Black** corresponds to incorrect labeling.

4 Conclusion and Future Work

Throughout this project, we have seen that it is feasible to make use of parallel approximate k-nearest neighbor data structures based on Locality Sensitivity Hashing to approach classification problems. These parallel data structures proved to scale quite well relative to the number of cores available on the hardware when it came to pre-processing part of building the data structures, though queries did not seem to benefit much on an individual query basis. Future work would be to make the data structures capable of batch queries using threading to improve the efficiency of those types of queries. Further, the software needs to be tested against higher dimensional classification problems with some more labels, which implies the need for larger number of nearest neighbors, to see how the approximation aspect of the data structures degrade as the number of labels increase.