

Northwest Nazarene University

Detecting Burn Severity in Post Wildland Fire Imagery through k-Dimensional Trees and
k-Nearest Neighbors Machine Learning Algorithms

THESIS

Submitted to the Department of Computer Science and Mathematics

in partial fulfillment of the requirements

for the degree of

Bachelor of Science

Llewellyn B. Johnston

2017

THESIS

Submitted to the Department of Mathematics and Computer Science

In partial fulfillment of the requirements

for the degree of

BACHELOR OF SCIENCE

Llewellyn B. Johnston

2017

Detecting Burn Severity in Post Wildland Fire Imagery through k-Dimensional Trees and
k-Nearest Neighbors Machine Learning Algorithms

Author:



Llewellyn Boscawen Johnston

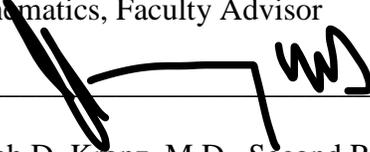
Approved:



Dale B. Hamilton, Associate Professor, Department of Computer Science and

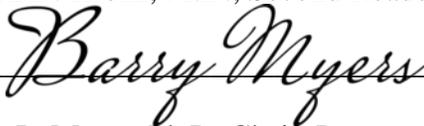
Mathematics, Faculty Advisor

Approved:



Joseph D. Kronz, M.D., Second Reader

Approved:



Barry L. Myers, Ph.D, Chair, Department of Computer Science and

Mathematics

Abstract

A kNN Classification using KD Trees

JOHNSTON, LLEWELLYN (Department of Mathematics and Computer Science), HAMILTON, DALE A. (Department of Mathematics and Computer Science)

This is a system for analyzing post-fire imagery to determine wildland fire severity. The system is written for the FireMAP project. It is written in C++ and C using the open source image processing library OpenCV. The system primarily is comprised of a k-Dimensional binary tree, for storing training data, along with a k-Nearest Neighbors algorithm to quickly classify imagery based on the training data. The algorithm utilizes parallel processing to fully utilize the CPU greatly increase the classification speed. Most of the system is written in house to provide a unique and modifiable algorithm implementation for use by the FireMAP project. This effort is important to the FireMAP project because it provides the ability to automate the severity determination process that would previously take weeks for a Burn Area Emergency Response (BAER) team to do now takes several minutes or several seconds. So, the solution provided by this project is cheaper, faster, requires less manpower, and is overall a safer approach to the issue of burn severity analysis. Several areas of research and work were not included in this project and will require further attention. These areas are object based analysis of imagery, further optimization and training for the classifier, more robust system testing, and full integration of the classifier into the full FireMAP project.

Acknowledgements

I would like to acknowledge Dr. Joe Kronz for giving his time and energy into tutoring me on the details of prostate cancer detection and for providing sample imagery to work with for that part of the project. I would also like to thank Northwest Nazarene University for providing a research position for me over the summer of 2016 with the FireMAP project. This research project and publication was made possible by an Institutional Development Award (IDeA) from the National Institute of General Medical Sciences of the National Institutes of health under Grant #P20GM103408. In addition, I would like to acknowledge Dr. Barry Myers for the work he performed to provide me with the opportunity to pursue the IDeA Network of Biomedical Research Excellence Grant (Idaho INBRE). Dr. Myers was my mentor both in this project and outside of it. Lastly, I would like to thank Prof. Dale Hamilton for his mentorship in this project and his work to provide me with this opportunity.

Table of Contents

Title Page	i
Signature Page	ii
Abstract	iii
Acknowledgements	iv
Table of Contents	v
Project Background	1
Project Overview	2
Testing and Results	11
Future Work	17
Conclusion	18
References	19
Appendix	20

List of Figures

Figure 1 – 2D kNN Algorithm	2
Figure 2 – Binary Tree	3
Figure 3 – k-D Tree	4
Figure 4 – Initial Traversal to Bottom of the k-D Tree	5
Figure 5 – Reverse Traversal of Tree to Find Possible Other Neighbors	6
Figure 6 – Custom Stack	9
Figure 7 – Parallelization and Mutex Locks	10
Figure 8 – Reynolds Creek Prescribed Burn Base Image	12
Figure 9 - Reynolds Creek Prescribed Burn Base Image with Training Data Sections ...	13
Figure 10 – Reynolds Creek Burn Post Classification with 5 Neighbors	14
Figure 11 – Kane Burn Orthomosaic Base Image	15
Figure 12 – Kane Burn Orthomosaic Post Classification	16

Project Background

The Fire Monitoring and Assessment Platform (FireMAP) is a large research project headed up by Dale Hamilton for his Ph.D. research. The purpose of FireMAP is to address wildland fire analysis and mapping which are both primarily done by hand and by Burn Area Emergency Response (BAER) teams. This work is time consuming, expensive, and potentially dangerous with team members working in a recently active fire zone with possible hotspots and unstable wildland. FireMAP proposes to perform the data collection and analysis through Unmanned Aerial Systems (UAS) with attached high definition cameras. Actionable knowledge is extracted from the imagery.

FireMAP utilizes machine learning classifiers to classify all of the pixels within an image based on a pixels' spectral values, the levels of red, green, and blue present in the pixel, in reference to previously classified data called training data. Machine Learning classification methods are applicable to far more than just wildland fire imagery. One field of interest that has been addressed previously in a project by Joshua Benton attempted to classify and analyze prostate cancer in histological imagery. His approach utilized MATLAB as his working base. FireMAP already had an implementation of a k-Nearest Neighbors (kNN) algorithm which provided automated analysis of post fire imagery but it heavily relied on an open source library called OpenCV which has several key features already developed. This resulted in fairly slow classification times and the lack of control of the code and the overall process.

The purpose of this project is to improve on the previous methods of classification, MATLAB and the OpenCV dependent FireMAP kNN. The main improvements this project aimed to accomplish was to develop a customized kNN algorithm using a k-

Dimensional tree and to improve on current classification speeds. This improved classifier is then to be applied to both post fire imagery and to prostate cancer slides to test its effectiveness. The last part of the project was to implement object based classification using the work of Patrick Richardson, an NNU senior working on the FireMAP project.

Project Overview

This project required the creation and implementation of a C++ implementation of a KD-Tree and a kNN Algorithm. A k-Nearest Neighbors algorithm uses training data which is points of data that have been assigned a classification by a user prior to the operation of the algorithm. These pieces of training data are then used to determine the most likely classification of a new data point based on their spatial proximity to the new piece of data. Figure 1 provides a visualization of the basic workings of a kNN algorithm within a two dimensional space. In Figure 1 the kNN algorithm is running on a k of five, meaning that it uses the five closest points of training data to determine the most likely classification of the new data point – the black point. In this case the new piece of data is most likely blue because three of the five closest data points are blue.

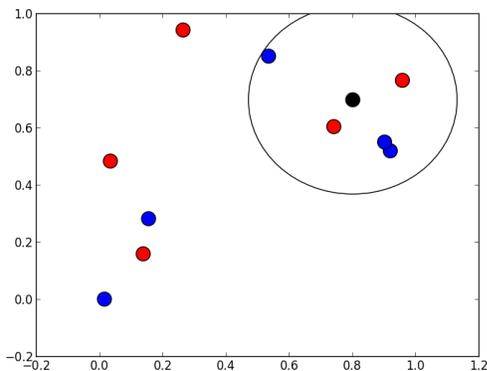


Figure 1 – 2D kNN Algorithm

For the kNN to work, the training data has to be stored in a manner that is quick to access and maintains the spatial aspects of the data. The approach taken in this project is to store the data in a k-Dimensional tree. Though there are other data structures available to use with the kNN algorithm the k-D tree is a commonly used data structure and is one of the most reliable methods for storing spatial data, which is data that deals with multiple dimensions (Otair 98).

Binary trees are a common method for storing data because they provide very fast access to data and are flexible in their set up and active usage. Binary trees work by repeatedly dividing the desired data in half and sorting it based on a midpoint – all data less than the midpoint is placed on the left, all data greater is placed on the right. This process is repeated until you are dealing with single points. Figure 2 gives a visual representation of a simple binary tree.

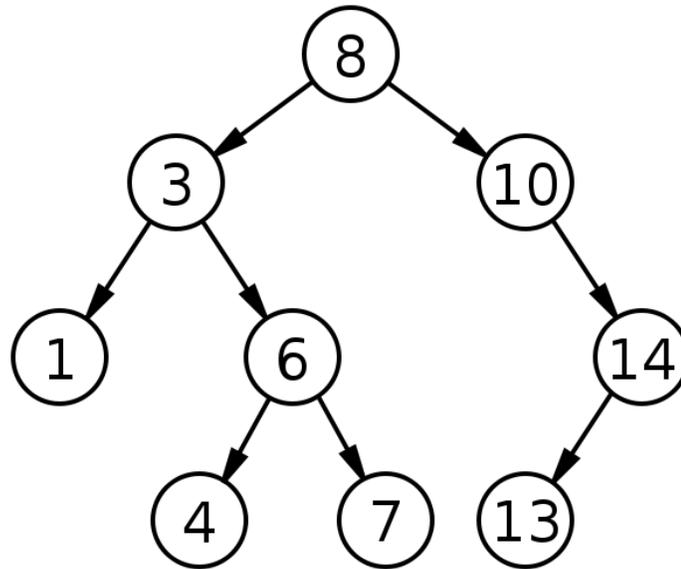


Figure 2 – Binary Tree

For this project, however, binary trees do not work because the subject matter has multiple dimensions that need to be considered – the spectral values red, green, blue, infrared, in

addition to a variety of texture metrics are all examples of dimensions being considered by FireMAP. Additionally, there is the possibility that more dimensions will be considered in the future work of the project. So, the storage system used had to be able to handle multiple dimensions. A basic binary tree cannot handle this requirement. To fulfill this requirements a k-Dimensional (k-D) tree was implemented. A k-D tree is similar to a binary tree in its underlying functionality and form but it differs from a binary tree because the dimension that the data is sorted on changes depending on which level of the tree you are on. The first level will be sorted based on your first dimension, your second level will be sorted on your second dimension and so on until you have sorted on each of your dimensions at which point your start again with the first dimension (Moore 62-64). Figure 3 gives a representation of this process with two dimensions: x and y.

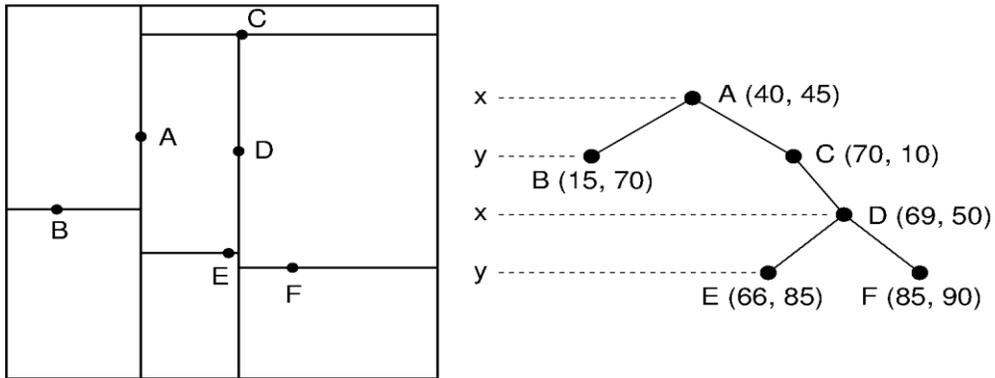


Figure 3 – k-D Tree

The k-D Tree and kNN are the two main structures that are used within this project. The k-D tree stores the training data and the kNN algorithm uses the stored data to classify unknown data.

With the k-D tree established it is now possible for the kNN to utilize it for the classification process. Visually the kNN looks at the closest training data to determine the

classification of a new data point but the process behind that is more complicated as it requires moving through the k-D tree to find those training points.

```
PixelNode* currentNode = head;
...
...
bool bottom = false;
...
...
while (!bottom)
{
    if (newObjectNode[kLvl] < currentNode->getValue(kLvl) && currentNode->getLeft() != 0)
    {
        if (currentNode->getLeft() != 0)
        {
            currentNode = currentNode->getLeft();
            ...
        }
        else
            bottom = true;
    }
    else
    {
        if (currentNode->getRight() != 0)
        {
            currentNode = currentNode->getRight();
            prevPath = 1;
        }
        else
            bottom = true;
    }
    kLvl = (kLvl + 1) % numberOfValues;
}
...
...
```

Figure 4 – Initial Traversal to Bottom of the k-D Tree

To begin the process the kNN traverses the tree to the bottom of the k-D Tree. Figure 4 shows this process. The starting point (currentNode) is set to the top of the tree (head). From there the algorithm compares the value of the new data to the one stored in the tree and moves left or right depending on if the value is less than or greater than/equal to accordingly. This process is then repeated, using the next dimension as explained previously, until the training node being considered is at the bottom of the tree. The indicator that the bottom has been reached is that the necessary path is not populated – which means that the path is 0 rather than another node.

```

while (!pixelStack.empty())
{
    currentNode = pixelStack.top();
    pixelStack.pop();

    prevPath = previousPath.top();
    previousPath.pop();

    distance = currentNode->getDistance(newObjectNode);

    if (neighbourList->isNotFull() || distance < neighbourList->getMax())
    {
        neighbourList->insert(currentNode, distance);
        ...
        childStack.push(child) //PUSH THE NEW CHILD ONTO THE CHILDSTACKS
        ...
        //TRAVERSE THE ENTIRE SUBBRANCH LOOKING FOR CLOSER POINTS
        while (!childStack.empty())
        {
            currentNode = childStack.top();
            childStack.pop();
            if (currentNode != 0)
            {
                distance = currentNode-
                >getDistance(newObjectNode);
                neighbourList->insert(currentNode, distance);

                childStack.push(currentNode->getRight());
                childStack.push(currentNode->getLeft());
            }
        }
    }
}

```

Figure 5 – Reverse Traversal of Tree to Find Possible Other Neighbors

The next part of the kNN is to backtrack from the bottom of the tree and check the parent nodes and branches for other neighbors and possibly better neighbors. Figure 5 shows this process. This section of the code loops until all the desired nodes within the k-D tree have been checked. The desired nodes are stored in pixelStack. The algorithm moves from the bottom of the tree to look at the parent node, which is the training data point directly above the current one within the tree. If the kNN has not identified the proper number of neighbors to the data point then the parent is added onto the list of neighbors (neighbourList) or if the parent node is closer in distance to the data point than the current least likely neighbor then the parent replaces that neighbor in the list. In both cases the parent node is added to the neighbor list then the kNN checks all the children of that parent node. This means that the

kNN looks at all of the nodes below the parent node to see if they are possible closest neighbors to the data point being classified. Only one side has to be checked, either left or right, as the previous side has been checked before looking at the parent node. The process of backtracking and checking parents and other branches continues until the end of the kNN. If neither of the conditions are met at any point, the parent node is farther away than the farthest current neighbor and the kNN has identified the proper number of neighbors, then no actions are taken and the kNN is complete. This new list of neighbors is then used to assign a class to the new data point being classified (Lavrenko).

At this point both the k-D tree and the kNN algorithm are created and function as expected. Both are fully customizable in-house solutions and both are no longer heavily tied to OpenCV, as the previous edition of the project was. With the k-D Tree and kNN established the next steps in the project are to optimize the algorithm to run as quickly as possible and to begin classifying imagery – which involves training data selection and refinement of the kNN algorithm.

There are several steps in the optimization of the system. The main steps are the balancing of the k-D tree, the implementation of a custom stack to avoid recurrence and the use of the default system stack, and the parallelization of the kNN process. One issue with data trees is that they can become very imbalanced to the point that they are not providing much improvement in run times over a straight linear approach. What this means is that the tree can be strongly skewed to one side based on where the data is divided. One side of the data tree could be much longer than the other side which means that getting at information on the longer side takes longer which leads to an overall slowdown of the system. Balancing a normal binary tree is possible during run time but balancing a k-D

tree, because it deals with multiple dimensions, is difficult and sometimes almost impossible. To solve this issue the k-D tree first reads in all of the data and sorts it based on the first dimension, it then sorts the data on the two halves based on the second dimension, and so on (Brown 51-55). This is done before loading all the data into the tree and it allows the tree to be created in a balanced form so that each data access is as quick as possible.

In order to accomplish all of the prior described sorting there is generally a large amount of recurrence, which is where one section of the program is called multiple times by itself to accomplish a task. The general method for doing this results in slow run times as each time the part of the program is called all the information currently held by the program has to be stored into the system stack to be restored at a later point. This process is rather slow and has lots of overhead. To address this a custom stack was created which has faster store and load speeds than the system stack and erases the need for recurrence within the program. Instead of a section of the program calling itself now instead the section of the program will loop several times until the stack is empty. This optimization is primarily used when balancing the data for the creation of the tree as there is an enormous amount of sorting that has to happen here but it is also used in the kNN classification process described previously when doing the backtracking within the k-D tree.

```
stack<int> qSortStack;

int left;
int right;
int kLvl;
int midPoint;

qSortStack.push(0);
qSortStack.push(objectNodeVector.size() - 1);
qSortStack.push(0);
do
{
    kLvl = qSortStack.top();
```

```

qSortStack.pop();
right = qSortStack.top();
qSortStack.pop();
left = qSortStack.top();
qSortStack.pop();
...
...
} while (!qSortStack.empty());

```

Figure 6 – Custom Stack

Figure 6 provides a brief look into how the custom stack is implemented and works within the sorting function. The variables left, right, kLvl, and midpoint are important to the sort – in this case quick sort is being used. The stack is initialized with the default values and then within the do-while loop the required information is pulled from the stack, the sort is completed and the information for the left and right sections are then pushed onto the stack. This allows the multi-level sort that balances the k-D tree to run as quickly as possible.

The last major optimization is the parallelization of the classification process. The kNN algorithm is a pixel based classification, classifying each pixel individually within the desired image. In the case of this project the desired images are often orthomosaics of large fires – these are images that are composed of multiple smaller images stitched together to display a larger region than visible in a single image. The orthomosaics are very large, the Kane Fire orthomosaic (Figure 7) is 18725 pixels by 14302 pixels. This means that there are 267,804,950 pixels to classify. The run time of this can be enormous especially when you can only classify one pixel at a time. This is where parallel processing becomes very useful. Instead of handling one pixel at a time the program can process multiple pixels at a time which reduces the run time enormously from minutes to seconds and hours to minutes. Within this project each column of the picture each row of the image is handled separately in parallel. What this means is that separate rows are being classified simultaneously. This does cause some issue, however, as it makes it possible to override information being dealt

with by a separate parallel process. The information being written in this section are the classifications for the pixel so to handle this issue the program utilizes a mutex lock to control access to the output variable between parallel processes.

```
mutex threadLock;
int pixelVal;

parallel_for(0, inputImage.rows, 1, [&](int row)
//Parallel For loop cycling through each of the rows
{
    string className;
    for (int col = 0; col < inputImage.cols; col++)
    //For loop cycling through each of the columns
    {
        ...
        Vec3b& bgr = inputImage.at<Vec3b>(row, col);

        ...
        ...

        pixelTree->kNearestNeighbourNodes(heap, newPixel);

        pixelVal = voting(heap, neighbourCount);

        threadLock.lock();

        outputImage.setPixelClasses(row, col, pixelVal);

        threadLock.unlock();

        ...
    }
});
```

Figure 7 – Parallelization and Mutex Locks

Figure 7 outlines the parallel process used within the project. The mutex lock is established as threadLock prior to the parallel for loop. The parallel for loop is established to run on each row within the image being classified and hands off each loop to a separate processor as they become available on the computer running the kNN program. Each of these parallel loops then have a for loop to analyze each pixel within the row and assign a class to each pixel using the kNearestNeighbourNodes function. The usage mutex lock is seen near the end of the figure where the mutex locks the parallel loops before setting the pixel class for the pixel with the setPixelClasses function. The mutex lock is then released so that the other parallel processes can write set the classes

for their pixels. The parallelization is the main optimization carried out and is the main reason for the enormous speed improvement over previous iterations of the kNN classifier.

The main components of the project are the kNN algorithm and the k-D tree. The k-D tree is necessary for the kNN algorithm to function properly and both components have been written for this project. Both the k-D tree and kNN algorithm are optimized to be as efficient as possible which allows for very quick runtimes and efficient use of resources. This section has given the overview of the project's components and what has been accomplished by the project.

Testing and Results

The next step to the process is to actually perform the classifications and create training data and resulting imagery. Because this individual project is part of the larger FireMAP project selecting the training data to use was not part of the functionality of this project but it will be covered briefly for the purpose of explaining the process of obtaining resulting imagery. All of the following results were obtained using a Windows 8.1 machine with an Intel Quad Core i7-3630QM CPU, 12 GB DDR3 RAM, and 4GB Nvidia GeForce GT650M.



Figure 8 – Reynolds Creek Prescribed Burn Base Image

Figure 4 shows one of the three primary fires used for the testing of the kNN classifier. It shows the base image prior to any training data is extracted and prior to any of the classification. Within this image, there are two main burn regions visible surrounded by dry unburnt regions.

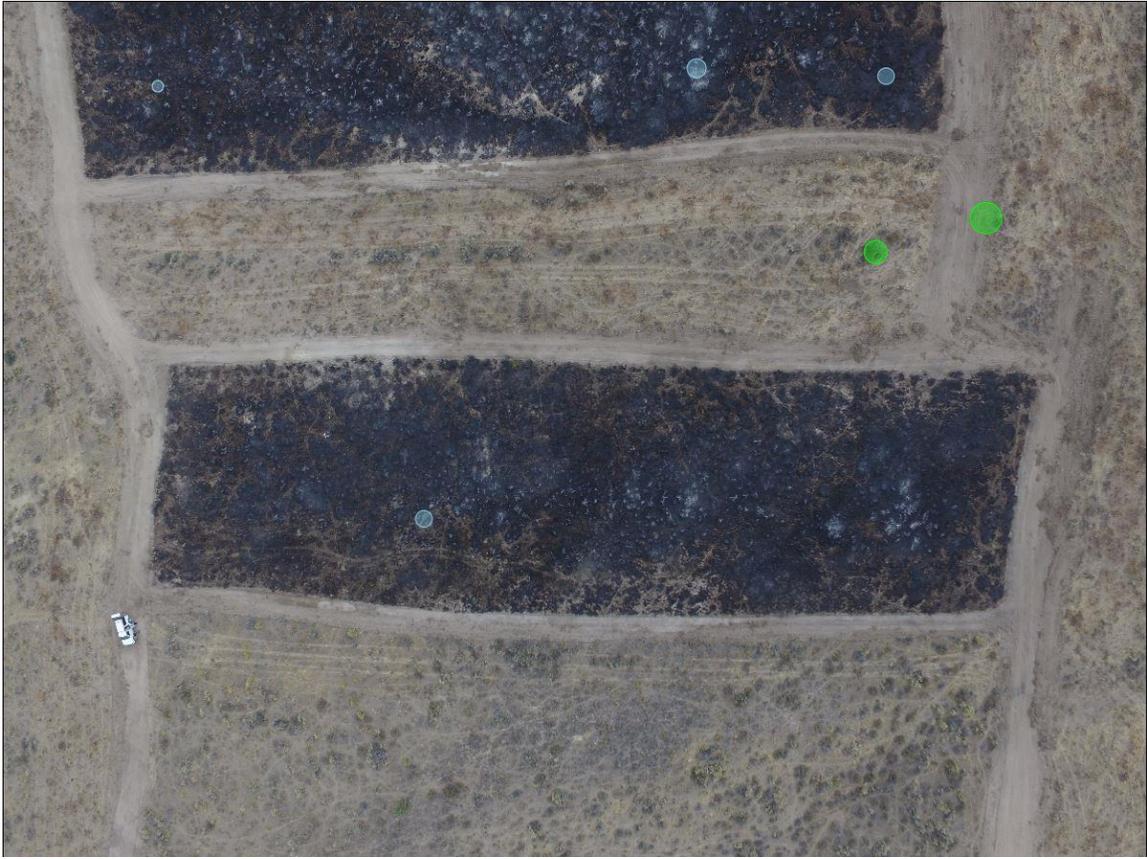


Figure 9 - Reynolds Creek Prescribed Burn Base Image with Training Data Sections, Green is Unburn, Blue is Burn

Figure 5 shows the training data selection process. The blue circles are labelled as burnt regions. The green circles are labelled as unburnt regions. Both of these groupings of regions are exported as training data with the appropriate labels to be classified.



Figure 10 – Reynolds Creek Burn Post Classification with 5 Neighbors and an 18 Second Runtime

The training data previously generated is loaded into the k-D tree and balanced. The Reynolds Creek image is loaded into the kNN application as the input image and the kNN runs through the image classifying each pixel. Figure 6 shows the resulting image from the kNN run with a neighbor count of 5. There are a total of 12,000,000 pixels within this image. The entire classification process, including the construction of the k-D tree, took around 18 seconds.

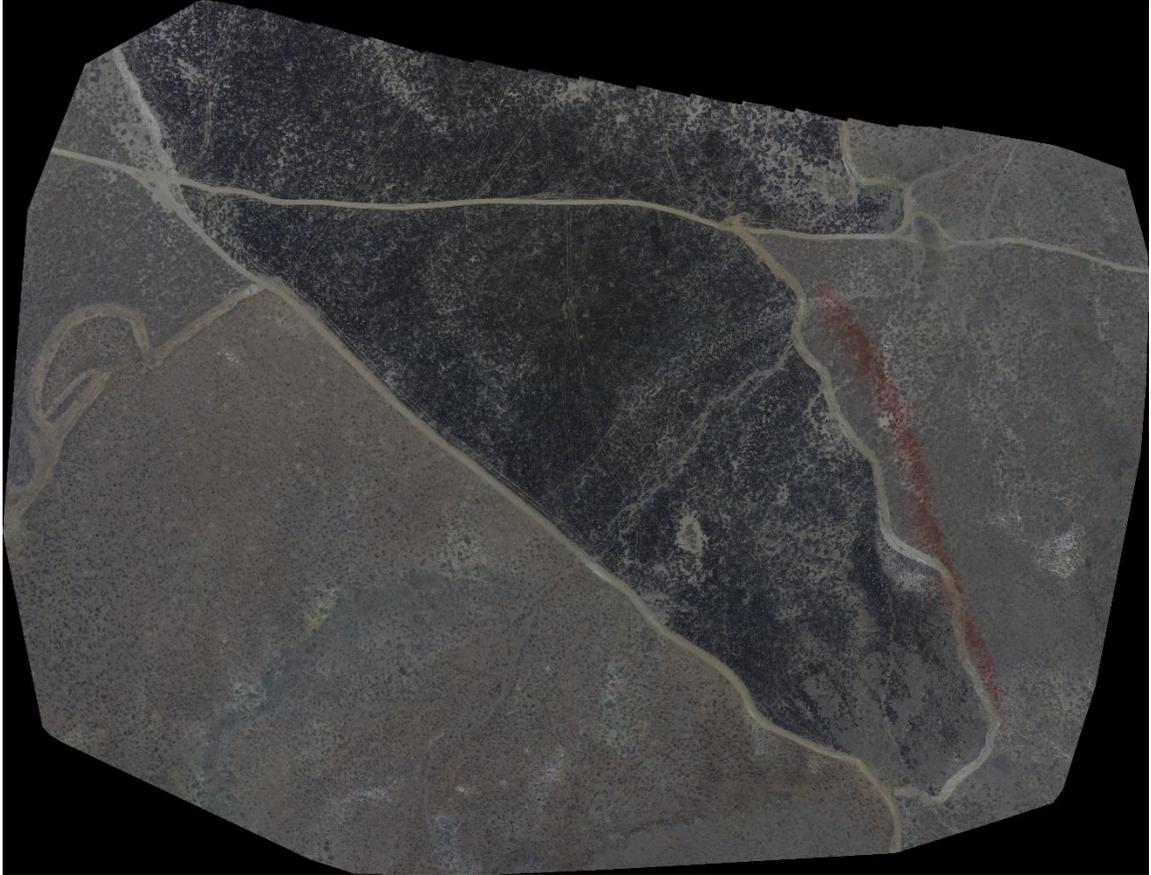


Figure 11 – Kane Burn Orthomosaic Base Image

Figure 7 show the largest image used to test the kNN project. It shows the Kane Fire orthomosaic which is 18725 pixels by 14302 pixels. The Kane Fire was a wildland fire over the summer, 2016 in the Owyhee Mountains in southern Idaho. Multiple separate images were taken using a UAS and the images are stitched together using a third party application, called Pix4D, to create one larger image of the entire fire region. For the classification of this image the training data was pulled from the central burnt section and from the top right unburnt regions. Using the same methods as used in Figure 5.

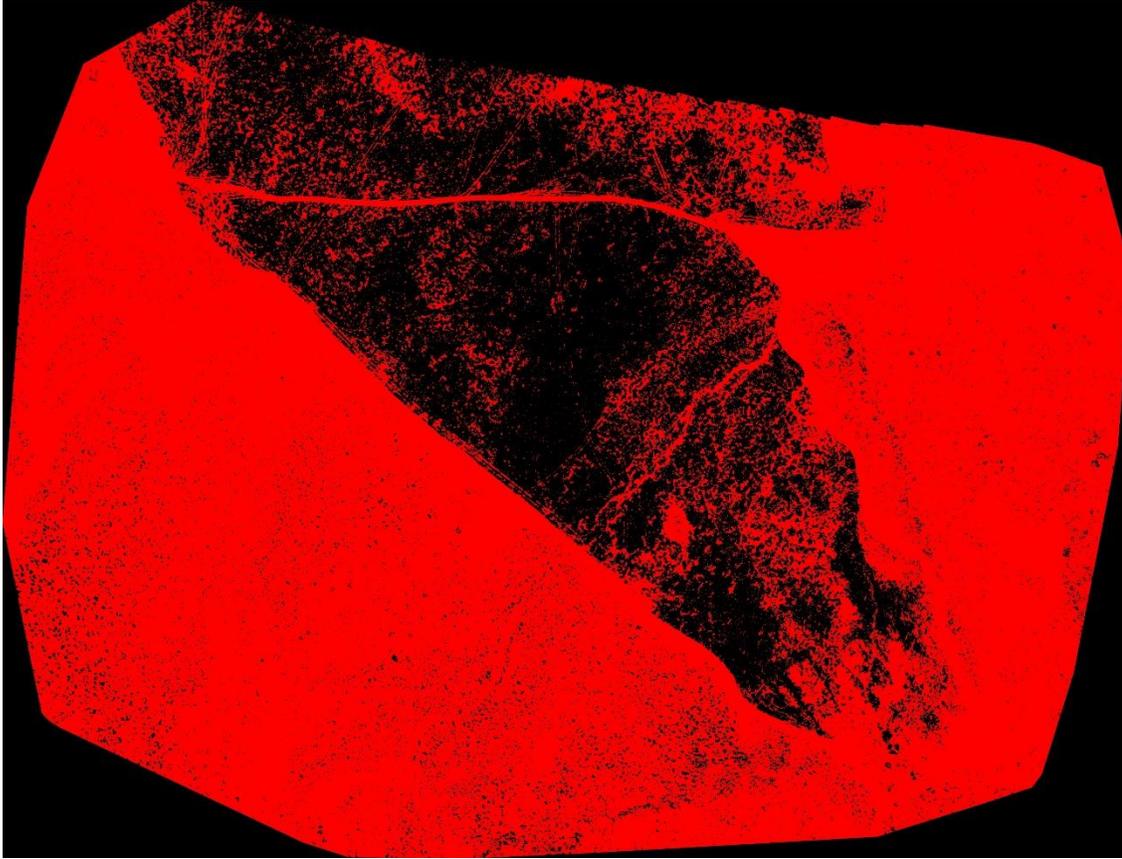


Figure 12 – Kane Burn Orthomosaic Post Classification, Red is Unburnt, Black is Burnt, 4 Minute 52 Second Runtime

Figure 8 shows the resulting classification of the Kane Fire imagery. The red represents unburnt pixels and the black is burnt pixels. The entire classification of this image took 4 minutes and 52 seconds including the construction of the k-D tree. This resulting image and Figure 6 both show an issue faced in the process.

Both of the images have speckling and noise which is false classification. This means that there is a pixelization effect visible within the unburnt regions due to data points being falsely classified as burnt. There are two main ways of handling this issue. The first is to refine the training data used. A big difficulty with assisted machine learning algorithms, like the kNN algorithm, is the selection of training data. The second main way of handling the issue is to perform data cleanup through processes like blurs or morphology – which is handled by a different section of the FireMAP project.

Future Work

With the limited time available for the development of this project and it being tied to the larger FireMAP project there were some limitations placed on what could be accomplished. The largest section of the project that was not completed is object based classification. An original objective was also to implement the object based image classification, where spatio-spectral clusters are classified as a group. This objective was not accomplished. A vital part of that process was developed in a separate section of the FireMAP project and proved to be far more difficult than first anticipated. So, it was not completed in time to be used with the kNN algorithm. Enhancement of the kNN classifier to enable object based classification is an important avenue to pursue in the future. The resulting fire imagery is fairly accurate but the resulting prostate cancer imagery is not very accurate and that is because the processes are fairly dissimilar. Fire is able to be analyzed pretty accurately based on the color spectrum and other dimensions, like texture. Prostate cancer analysis is more complicated and quite a bit of it is dependent on proximity of different objects. So, performing classification on prostate cancer without the object oriented classification methods proved fairly ineffective. This means that another goal for future work is to work more closely with prostate cancer imagery with the object oriented classification mentioned previously.

Overall the core of the project was completed. The kNN algorithm was created to function with a flexible number of dimensions. The k-D tree was created to function with the same dimensional flexibility. Results were obtained and proved to be fairly accurate. However, those results can be improved especially through better training data selection. The development of more accurate training data is another goal to be pursued in the

future of this project. Additionally, more testing is necessary with more rigorous testing parameters to develop a more accurate assessment of the efficiency of this classifier.

There are several other dimensions that should be considered in the future like texture and shape, which will likely improve the prostate cancer results tremendously.

Conclusion

In conclusion, the overall project was successful. Despite there being fairly extensive future work to be taken and possible improvements on the system a functional kNN algorithm and k-D tree were developed. A far greater understanding of the algorithms and of machine learning overall was gained through the process of the project. Additionally a greater understanding of working within a larger project and with a group of other developers were also gained. This project has laid the initial foundation for the custom kNN for the FireMAP project and created a very fast and optimized system.

References

- Benton, Joshua. Detecting Prostate Cancer in Histological Images via Image Segmentation and Supervised Machine Learning. Thesis. Northwest Nazarene University, 2016. Nampa: NNU, 2016. Print.
- Brown, R.A. (2015). Building a Balanced k-d Tree in $O(kn \log n)$ Time. *Journal of Computer Graphics Techniques*, 4(1).
- Gupta, Sachin. "Iterative traversals for Binary Trees." HackerEarth. N.p., n.d. Web. 01 Apr. 2017.
- Hamilton, Dale (2015). Prototyping machine learning classifiers for mapping wildland fire extent and severity. Northwest Nazarene University, Nampa, ID.
- "Intuitive Classification using KNN and Python." *ŷhat | Blog*. N.p., 25 July 2013. Web. 01 Apr. 2017.
- Lavrenko, Victor. "kNN. 15 K-d tree algorithm." Youtube. University of Edinburgh. 15 Sep 2015. Web.
- Moore, C.W. (1991). An Introductory Tutorial on KD-Trees. *Efficient Memory-based Learning for Robot Control*.
- "OpenDSA: All Modules TODO List." 8.2. k-d Trees. OpenDSA, n.d. Web. 01 Apr. 2017.
- Otair, Mohammed, Dr. "Approximate K-Nearest Neighbour Based Spatial Clustering Using K-D Tree." *International Journal of Database Management Systems* 5.1 (2013): 97-108. Web.

Appendix

k-D Tree

PixelTree.h

```
#include <string>
#include <vector>
#include "BPQHeap.h"

using namespace std;

#ifdef PIXEL_LINKED_TREE
#define PIXEL_LINKED_TREE

/*****
    CLASS: PixelNode

    DEFINITION OF THE PIXELNODE CLASS WHICH WILL MAKE UP THE KD TREE AND BE USED
    IN THE KNN ALGORITHM
    The basic premise is that each of the spectral values of the pixels are stored
    in the PixelNode which will then be used to run the kNN Algorithm.

    Llewellyn Johnston - May 2016
    *****/

class PixelNode
{
private:
    PixelNode* left;
    PixelNode* right;
    PixelNode* parent;
    int pxCount = 1;        //used for pruning, if exact pixels exist
    string nodeType;
    int kLvl = 0;
    vector<float> PixelValues;

public:
    PixelNode(vector<float>, string, int, int);
        PixelNode(vector<float>, string, int, int, PixelNode*);    //Overloaded
                                                                    if Parent is
                                                                    known

    void setLeft(PixelNode* newLeft) { left = newLeft; }
    void setRight(PixelNode* newRight) { right = newRight; }
    void setParent(PixelNode* newParent) { parent = newParent; }
    void incrementCount() { pxCount++; }

    PixelNode* getLeft() { return left; }
    PixelNode* getRight() { return right; }
    PixelNode* getParent() { return parent; }
    string getType() { return nodeType; }
    bool isMatch(PixelNode*);
    double getValue(int k) { return PixelValues[k]; }
    int getCount() { return pxCount; }
    float getDistance(vector<float>);
};
```

```

/*****
CLASS: PixelTree

DEFINITION OF THE PIXELTREE CLASS WHICH WILL BALANCE THE INPUTS AND BE USED IN
THE KNN ALGORITHM
Basically, it has the public function balanceVector() which is a two step
recursive function that balances the users input training data vector so that
the tree created from it is balanced and is functional on O(log(n)). This is the
most lengthy part of the tree creation though, balanced vector is then to be
saved so that this process can be skipped next time the user uses the training
data. Additionally, there is the pruneBalanceVector which will prune the
training data (get rid of duplicate entries and increment the counters for those
pixels) and then use this pruned vector to construct the tree.

Llewellyn Johnston - May 2016
*****/

class PixelTree
{
private:
    PixelNode* head;
    int numberOfValues = 0;
    //Implement a quicksort function (kind of) to construct the tree
    void quickPixelSort(vector<PixelNode*>&, int, int, int);

public:
    void printTypes(PixelNode*);
    PixelTree(int K)
    {
        head = 0;
        numberOfValues = K;
    }
    ~PixelTree();

    PixelNode* getHead() { return head; }
    int isEmpty() { return head == 0; }
    void setValueCount(int newCount) { numberOfValues = newCount; }
    int getValueCount() { return numberOfValues; }
    void deleteNode(PixelNode*, bool);

    void createPrunedVector(vector<PixelNode*>&, PixelNode*);
    void createPruningTree(vector<PixelNode*>&, int);
    vector<PixelNode*> pruneBalanceVector(vector<PixelNode*>&, int);
    void balanceVector(vector<PixelNode*>&);
    void constructTree(vector<PixelNode*>&, PixelNode*, int, int, int = 0);
    void kNearestNeighbourNodes(BPQHeap*, vector<float>);
};

#endif

```

PixelTree.cpp

```

#include "PixelTree.h"
#include <iostream>
#include <stack>

```

```

/*****
NAME: PixelNode (no parent)
CLASS: PixelNode

CONSTRUCTOR FOR THE PIXELNODE CLASS, THIS IS THE OVERLOADED VERSION THAT DOES
NOT TAKE A PARENT NODE
Takes a vector of ObjectValues and puts those into the PixelNode's PixelValues
vector, sets the kLvl and the node type - the classification like burn, white
ash, black ash, etc.

Llewellyn Johnston - May 2016
*****/

PixelNode::PixelNode(vector<float> nObjectValues, string type, int level, int kSize)
{
    nodeType = type;        //Set the node type
    kLvl = level;          //set the branch level for the object
    for (int j = 0; j < kSize; j++)    //Place all of the pathing information into
                                        the object class
    {
        PixelValues.push_back(nObjectValues[j]);
    }
}

/*****
NAME: PixelNode (with parent)
CLASS: PixelNode

CONSTRUCTOR FOR THE PIXELNODE CLASS, THIS IS THE OVERLOADED VERSION THAT DOES
TAKE A PARENT NODE
Takes a vector of ObjectValues and puts those into the PixelNode's PixelValues
vector, sets the kLvl and the node type - the classification like burn, white
ash, black ash, etc. Also sets the parent ptr.

Llewellyn Johnston - May 2016
*****/

PixelNode::PixelNode(vector<float> nObjectValues, string type, int level, int kSize,
PixelNode* nParent)
{
    nodeType = type;        //Set the node type
    kLvl = level;          //set the branch level for the object
    for (int j = 0; j < kSize; j++)    //Place all of the attribute values into
                                        the object class
    {
        PixelValues.push_back(nObjectValues[j]);
    }
    parent = nParent;
}

/*****
NAME: getDistance
CLASS: PixelNode

CALCULATES THE EUCLIDIAN DISTANCE OF A GIVEN NODE BASED ON THE NODE ITSELF AND
THE PARAMETER NODE

Llewellyn Johnston - May 2016
*****/

```

```

float PixelNode::getDistance(vector<float> newPixelNode)
{
    float distance = 0;
    for (int i = 0; i < PixelValues.size(); i++)
    {
        distance += pow(newPixelNode[i] - PixelValues[i], 2);
    }
    return distance;
}

/*****
NAME: isMatch
CLASS: PixelNode

COMPARES THE PIXELVALUES OF THE COMPARENODE AGAINST THOSE OF THE CURRENT NODE TO
CHECK IF EXACT MATCH
If they are an exact match then returns false, otherwise will return true.

Llewellyn Johnston - May 2016
*****/

bool PixelNode::isMatch(PixelNode* compareNode)
{
    bool match = true;
    for (int i = 0; i < PixelValues.size(); i++)
    {
        if (compareNode->getValue(i) != PixelValues[i])
        {
            match = false;
            break;
        }
    }
    if (match == true && compareNode->getType() != nodeType)
        match = false;
    return match;
}

/*****
NAME: deleteNode
CLASS: PixelTree

RECURSIVE FUNCTION THAT DELETES THE PROVIDED NODE AND ALL OF ITS CHILD (AND
GRAN-KIDS AND SO ON)
Checks to see if the node has children, if it does then the function is called
on those children, after that the node is deleted and set to 0. If the node is
the initial node and it has a parent (meaning that it is not the head) its
parent's child node that corresponds is set to 0.

Llewellyn Johnston - May 2016
*****/

void PixelTree::deleteNode(PixelNode* nodeToDelete, bool initial = 1)
{
    if (nodeToDelete->getLeft() != 0)
    {
        deleteNode(nodeToDelete->getLeft(), 0);
    }
    if (nodeToDelete->getRight() != 0)
    {
        deleteNode(nodeToDelete->getRight(), 0);
    }
}

```

```

    if (initial == 1 && nodeToDelete->getParent() != 0)
    {
        if (nodeToDelete->getParent()->getLeft() == nodeToDelete)
            nodeToDelete->getParent()->setLeft(0);
        else if (nodeToDelete->getParent()->getRight() == nodeToDelete)
            nodeToDelete->getParent()->setRight(0);
    }
    delete nodeToDelete;
    nodeToDelete = 0;
}

/*****
NAME: ~PixelTree (destructor)
CLASS: PixelTree

THE DESTRUCTOR FOR THE PIXELTREE CLASS
Calls the deleteNode function (which deletes all child nodes of the provided
node) on the tree head

Llewellyn Johnston - May 2016
*****/

PixelTree::~PixelTree()
{
    deleteNode(head);
}

/*****
NAME: quickPixelSort
CLASS: PixelTree

QUICK SORT FUNCTION FOR SORTING THE PIXEL NODES IN THE VECTOR BASED ON THEIR
DEFINED DIMENSION (KLVL) Functions off of the generic quicksort principle,
create a pivot point and swap around based on that then recursively sort the two
"halves" based on that pivot point. Only unique functionality is that it
performs the quicksort based on the provided kLvl.

Llewellyn Johnston - May 2016
*****/

void PixelTree::quickPixelSort(vector<PixelNode*> &pixelNodeVector, int left, int
right, int kLvl)
{
    stack<int> qSortStack;

    int midPoint;
    int i, j; //counters for the quicksort
    PixelNode* temp;
    float pivot;

    qSortStack.push(left);
    qSortStack.push(right);

    do
    {
        right = qSortStack.top();
        qSortStack.pop();
        left = qSortStack.top();
        qSortStack.pop();
    }
}

```

```

midPoint = ((left + right) / 2);
i = left;
j = right;
pivot = pixelNodeVector[midPoint]->getValue(kLvl);

while (i <= j)
{
    while (pixelNodeVector[i]->getValue(kLvl) < pivot)
        i++;
    while (pixelNodeVector[j]->getValue(kLvl) > pivot)
        j--;
    if (i <= j)
    {
        temp = pixelNodeVector[i];
        pixelNodeVector[i] = pixelNodeVector[j];
        pixelNodeVector[j] = temp;
        i++;
        j--;
    }
}

if (left < j)
{
    qSortStack.push(left);
    qSortStack.push(j);
}
if (right > i)
{
    qSortStack.push(i);
    qSortStack.push(right);
}
} while (!qSortStack.empty());
}

/*****
NAME: balanceVector
CLASS: PixelTree

RECURSIVE FUNCTION THAT CALLS QUICKPIXELSORT TO SORT CHUNKS OF THE VECTOR BASED
ON THE KLVL
Kind of works like a smaller quicksort except it does not actually move anything
around. It calls the quickPixelSort() function to sort the chunks of the vector
before calling the function again with a new kLvl and with the two halves of the
now "sorted" vector. The purpose is to organize the vector so that it can be
easily used to construct the tree and to be able to store the organized data so
that you can avoid having to do this slow sort again later on. Additionally the
midpoint is decremented until the kLvl values of the midpoint and its previous
do not match in order to properly sort with < on the left and >= on the right.

Llewellyn Johnston - May 2016
*****/

void PixelTree::balanceVector(vector<PixelNode*> &objectNodeVector)
{
    stack<int> qSortStack;

    int left;
    int right;
    int kLvl;
    int midPoint;

```

```

qSortStack.push(0);
qSortStack.push(objectNodeVector.size() - 1);
qSortStack.push(0);

do
{
    kLv1 = qSortStack.top();
    qSortStack.pop();
    right = qSortStack.top();
    qSortStack.pop();
    left = qSortStack.top();
    qSortStack.pop();

    if (left < right)
    {
        midPoint = ((left + right) / 2);
        kLv1 = kLv1 % numberOfValues;

        quickPixelSort(objectNodeVector, left, right, kLv1);

        while (midPoint > left + 1 && objectNodeVector[midPoint]-
>getValue(kLv1) == objectNodeVector[midPoint - 1]-
>getValue(kLv1))
        {
            midPoint--;
        }

        kLv1++;
        qSortStack.push(left);
        qSortStack.push(midPoint - 1);
        qSortStack.push(kLv1);
        qSortStack.push(midPoint + 1);
        qSortStack.push(right);
        qSortStack.push(kLv1);
    }
} while (!qSortStack.empty());
}

/*****
NAME: constructTree
CLASS: PixelTree

RECURSIVE FUNCTION THAT CONSTRUCTS THE PIXEL TREE BASED ON THE PROVIDED
PIXELNODE POINTER VECTOR
Additionally, the construction decrements the midpoint in order to keep all >=
values to the right of the midpoint. This means that all values < the current
node's kLv1 value go to the left, all >= go to the right.

Llewellyn Johnston - May 2016
*****/

void PixelTree::constructTree(vector<PixelNode*> &objectNodeVector, PixelNode* parent,
int left, int right, int kLv1)
{
    if (right != left)
    {
        static int counter = 0;
        int midPoint = ((left + right) / 2);
        objectNodeVector[midPoint]->setParent(parent);
    }
}

```

```

    if (head == 0) //Check if the tree is empty
    {
        head = objectNodeVector[midPoint];
    }

    if (midPoint - left > 1) //Check if there are nodes to the left of
        the midPoint
    {
        objectNodeVector[midPoint]->setLeft(objectNodeVector[(left +
            midPoint - 1) / 2]);
        counter++;
        constructTree(objectNodeVector, objectNodeVector[midPoint], left,
            midPoint - 1, (kLvl + 1) % numberOfValues);
    }

    if (right - midPoint > 1) //Check if there are nodes to the right of
        the midPoint
    {
        objectNodeVector[midPoint]->setRight(objectNodeVector[(right +
            midPoint + 1) / 2]);
        counter++;
        constructTree(objectNodeVector, objectNodeVector[midPoint],
            midPoint + 1, right, (kLvl + 1) % numberOfValues);
    }
}

/*****
NAME: createPrunedVector
CLASS: PixelTree

READS THE VALUES FROM THE PRUNED TREE BACK INTO THE PIXELNODEVECTOR TO BALANCE
AND CREATE KD-TREE
Only meant to be used after the createPruningTree function has been run on a
tree.

Llewellyn Johnston - May 2016
*****/

void PixelTree::createPrunedVector(vector<PixelNode*> &pixelNodeVector, PixelNode*
currentNode)
{
    if (currentNode != 0)
    {
        pixelNodeVector.push_back(currentNode);
        createPrunedVector(pixelNodeVector, currentNode->getLeft());
        createPrunedVector(pixelNodeVector, currentNode->getRight());
    }
}

void PixelTree::printTypes(PixelNode* current)
{
    if (current != 0)
    {
        printTypes(current->getLeft());
        cout << current->getType() << endl;
        printTypes(current->getRight());
    }
}

```

```

/*****
NAME: createPruningTree
CLASS: PixelTree

ADDS THE PIXELNODES FROM THE VECTOR ONTO THE TREE AND IF AN EXACT MATCH IS FOUND
INCREMENTS THE COUNT
Importantly, this tree is not balanced, it is merely for pruning the tree for
use in the balanced tree for the kNN algorithm. It functions like a normal tree
using the first vector node as the head and then goes left or right depending on
if the value of the current kLvl.

Llewellyn Johnston - May 2016
*****/

```

```

void PixelTree::createPruningTree(vector<PixelNode*> &pixelNodeVector, int position =
0)
{
    while (position < pixelNodeVector.size())
    {
        if (head == 0)
        {
            head = pixelNodeVector[position];
        }
        else if (position < pixelNodeVector.size())
        {
            int kLvl = 0;
            bool match = false;
            PixelNode* currentPosition = head;
            PixelNode* previousPosition = currentPosition;
            while (currentPosition != 0 && match != true)
            {
                previousPosition = currentPosition;
                if (pixelNodeVector[position]->getValue(kLvl) ==
                    currentPosition->getValue(kLvl))
                {
                    if (pixelNodeVector[position]-
                        >isMatch(currentPosition))
                    {
                        match = true;
                        currentPosition->incrementCount();
                        continue; //CHECK WHETHER THIS IS FOR
                                ENTIRE FOR LOOP
                    }
                }
                if (pixelNodeVector[position]->getValue(kLvl) <
                    currentPosition->getValue(kLvl))
                {
                    currentPosition = currentPosition->getLeft();
                }
                else
                {
                    currentPosition = currentPosition->getRight();
                }
                kLvl = (kLvl+1)%numberOfValues; //USING 3 FOR NOW BECAUSE
                                                FORGOT ABOUT THIS, TODO
            }
            if (match == false)
            {
                if (pixelNodeVector[position]->getValue(kLvl) <
                    previousPosition->getValue(kLvl))
                {

```

```

        previousPosition-
        >setLeft(pixelNodeVector[position]);
    }
    else
    {
        previousPosition-
        >setRight(pixelNodeVector[position]);
    }
    pixelNodeVector[position]->setParent(previousPosition);
    }
    }
    position++;
}
}

/*****
NAME: pruneBalanceVector
CLASS: PixelTree

PRUNES THE PIXELNODE VECTOR TO GET RID OF DUPLICATE ENTRIES WITHIN THE TRAINING
DATA
Calls the functions createPruningTree to establish an unbalanced pruned tree,
clears the training vector, then uses the createPurnedVector function to take
all the pruned nodes from the tree and place them back into the training vector.
Reset all the links on the training nodes otherwise can cause infinite loops
during the kNN.

Llewellyn Johnston - May 2016
*****/

vector<PixelNode*> PixelTree::pruneBalanceVector(vector<PixelNode*> &pixelNodeVector,
int K)
{
    //Create tree and when adding elements to tree check if they already exist
    //If element already exists (all the pixelNodeVector[] values are the same)
    //Then increment the counter on that node but don't make new leaf
    //If element does not exist then add leaf to tree like normal
    //After creating tree (and pruning) place the tree back into the original
    //vector to be balanced and used to create actual tree for kNN
    PixelTree* pruningTree = new PixelTree(K);
    pruningTree->createPruningTree(pixelNodeVector);
    pixelNodeVector.clear();
    pruningTree->createPrunedVector(pixelNodeVector, pruningTree->getHead());
    for (int i = 0; i < pixelNodeVector.size(); i++)
    {
        pixelNodeVector[i]->setLeft(0);
        pixelNodeVector[i]->setRight(0);
        pixelNodeVector[i]->setParent(0);
    }
    return pixelNodeVector;
}

/*****
NAME: kNearestNeighbourNodes
CLASS: PixelTree

THE MAIN PURPOSE OF THIS CLASS IS THE KNN ALGORITHM WHICH USES A BOUNDED
PRIORITY QUEUE FOR RETURN
A recursive function that traverses through the tree looking for a leaf node.
Each node is inserted into the queue (which may or may not add based on
distance) before stepping farther down the tree. If the queue is not full (not

```

all k neighbours have been found) then it steps back up one level and goes the opposite direction from its previous path and checks that entire subtree for nearest neighbours.

Llewellyn Johnston - May 2016

/******

```
void PixelTree::kNearestNeighbourNodes(BPQHeap *neighbourList, vector<float>
newObjectNode)
{
    PixelNode* currentNode = head;
    stack<PixelNode*> pixelStack;
    stack<int> previousPath;    //0 = LEFT, 1 = RIGHT, 2 = NEITHER, 3 = BOTH
    stack<PixelNode*> childStack;
    int prevPath;

    float distance;
    bool bottom = false;
    int kLvl = 0;

    while (!bottom)
    {
        if (newObjectNode[kLvl] < currentNode->getValue(kLvl) && currentNode-
>getLeft() != 0)
        {
            if (currentNode->getLeft() != 0)
            {
                currentNode = currentNode->getLeft();
                prevPath = 0;
            }
            else
                bottom = true;
        }
        else
        {
            if (currentNode->getRight() != 0)
            {
                currentNode = currentNode->getRight();
                prevPath = 1;
            }
            else
                bottom = true;
        }
        kLvl = (kLvl + 1) % numberOfValues;
    }

    distance = currentNode->getDistance(newObjectNode);
    neighbourList->insert(currentNode, distance);

    pixelStack.push(currentNode->getParent());
    previousPath.push(prevPath);

    if (currentNode->getLeft() != 0)    //FOR THE OCCASSION WHERE A LEAF NODE WAS
                                        SELECTED BECAUSE IT DIDN'T HAVE TWO
                                        CHILDREN NODES, ONLY ONE
    {
        distance = currentNode->getLeft()->getDistance(newObjectNode);
        neighbourList->insert(currentNode->getLeft(), distance);
    }
}
```

```

else if (currentNode->getRight() != 0)
{
    distance = currentNode->getRight()->getDistance(newObjectNode);
    neighbourList->insert(currentNode->getRight(), distance);
}

//AT THIS POINT THE CURRENTNODE IS SET TO THE LOWEST POINT IT WILL GET ON THAT
//BRANCH OF THE TREE, NOW WE START TRAVERSING UPWARDS
//ALSO THE NODE'S PARENT NODE IS CURRENTLY ON THE TOP OF THE PIXELSTACK

while (!pixelStack.empty())
{
    currentNode = pixelStack.top();
    pixelStack.pop();

    prevPath = previousPath.top();
    previousPath.pop();

    distance = currentNode->getDistance(newObjectNode);

    if (neighbourList->isNotFull() || distance < neighbourList->getMax())
    {
        neighbourList->insert(currentNode, distance);
        if (currentNode != head)
        {
            pixelStack.push(currentNode->getParent()); //PUSH PARENT
                                                    POSITION ONTO
                                                    STACK FOR
                                                    LATER CHECK IF
                                                    CLOSER OR
                                                    QUEUE NOT FULL

            if (currentNode == currentNode->getParent()->getLeft())
                previousPath.push(0); //CHECKED LEFT OF
                                    PARENT NODE ALREADY

            else
                previousPath.push(1); //CHECKED RIGHT OF
                                    PARENT NODE ALREADY

        }
        if (prevPath == 0) //IF LEFT ALREADY CHECK THEN SET CURRENT
                           NODE TO RIGHT CHILD
            childStack.push(currentNode->getRight());
        else if (prevPath == 1) //IF RIGHT ALREADY CHECKED THEN SET
                                CURRENT NODE TO LEFT CHILD
            childStack.push(currentNode->getLeft());

        //TRAVERSE THE ENTIRE SUBBRANCH LOOKING FOR CLOSER POINTS
        while (!childStack.empty())
        {
            currentNode = childStack.top();
            childStack.pop();
            if (currentNode != 0)
            {
                distance = currentNode-
                    >getDistance(newObjectNode);
                neighbourList->insert(currentNode, distance);

                childStack.push(currentNode->getRight());
                childStack.push(currentNode->getLeft());
            }
        }
    }
}

```

```
}  
}
```

Classification Voting System

BPQHeap.h

```
#include <string>  
  
class PixelNode; //class prototype to avoid recursive includes  
  
using namespace std;  
int const THRESHOLD = INT_MAX;  
  
#ifndef BPQ_HEAP  
#define BPQ_HEAP  
  
/*****  
    CLASS: BPQElement  
  
    THIS IS THE BPQELEMENT THAT IS CONTAINED IN THE BPQHEAP, IT CONTAINS A PIXELNODE  
    POINTER AND DISTANCE  
    The pixel is the one that is a nearest neighbour to the pixel being classified,  
    the distance is the Euclidian distance between the two pixels based on their  
    spectral values. Distance is not handled here but has to be passed in as the  
    intended value.  
  
    Llewellyn Johnston - June 2016  
  
*****/  
  
struct BPQElement  
{  
    PixelNode* node = 0;  
    float distance = THRESHOLD; //really large starting value by default  
};  
  
/*****  
    CLASS: BPQHeap  
  
    THIS IS THE BPQHEAP THAT CONTAINS THE ARRAY OF BPQELEMENTS AND ACTS AS A HEAP  
    (SPECIALIZED BINARY TREE)  
  
    Has the main function heapify which will place the element with the largest  
    distance at the top of the tree so that it can be compared to new possible  
    elements and replaced if a pixel with a smaller distance is found.  
  
    Llewellyn Johnston - June 2016  
  
*****/  
  
class BPQHeap  
{  
private:  
    int size = 0;  
    BPQElement* heapArray;  
    int left(int);  
    int right(int);  
    void heapify(int);  
};
```

```

public:
    BPQHeap(int newSize)
    {
        size = newSize;
        heapArray = new BPQElement[size];
    }
    ~BPQHeap()
    {
        delete[] heapArray;
    }

    PixelNode* getElement(int i) { return heapArray[i].node; }
    void insert(PixelNode*, float);
    float getMax() { return heapArray[0].distance; }
    bool isNotFull() { return heapArray[0].node == 0; }
    void reset();
};

#endif

```

BPQHeap.cpp

```

#include "BPQHeap.h"

/*****
    NAME: left
    CLASS: BPQHeap

    RETURNS THE LEFT CHILD OF THE GIVEN PARENT POSITION

    If the left child position falls outside of the maximum bound of the array then
    return -1 as a flag. This keeps the heap bounded to the initial size.

    Llewellyn Johnston - May 2016
    *****/

int BPQHeap::left(int parent)
{
    int left = 2 * parent + 1;
    if (left < size)
        return left;
    else
        return -1;
}

/*****
    NAME: right
    CLASS: BPQHeap

    RETURNS THE RIGHT CHILD OF THE GIVEN PARENT POSITION
    If the left child position falls outside of the maximum bound of the array then
    return -1 as a flag. This keeps the heap bounded to the initial size.

    Llewellyn Johnston - May 2016
    *****/

int BPQHeap::right(int parent)

```

```

{
    int right = 2 * parent + 2;
    if (right < size)
        return right;
    else
        return -1;
}

/*****
NAME: insert
CLASS: BPQHeap

REPLACES THE CURRENT MAXIMUM NODE WITH A NEW NODE AND THEN HEAPIFIES
Does swap the first and last elements of the array before calling the heapify
function.

Llewellyn Johnston - May 2016
*****/

void BPQHeap::insert(PixelNode* newNode, float newDistance)
{
    if (heapArray[0].distance > newDistance)
    {
        heapArray[0].distance = newDistance;
        heapArray[0].node = newNode;

        //Swap the first and last elements to prepare for a heapify
        BPQElement temp = heapArray[0];
        heapArray[0] = heapArray[size - 1];
        heapArray[size - 1] = temp;

        //heapify the array based on the first element
        heapify(0);
    }
}

/*****
NAME: heapify
CLASS: BPQHeap

HEAPIFIES THE HEAPARRAY RETURNS THE LEFT CHILD OF THE GIVEN PARENT POSITION
Find the left and right children's positions. Then, if both children fall within
the bounds of the array AND the distance of the right child is greater than the
distance of left child then set the left child to the max. Now childLPosition is
the position of the maximum distance element, if its distance is greater than
the distance of the current position (parent) then swap those two elements and
continue the heapify function on the swapped child node.

Llewellyn Johnston - May 2016
*****/

void BPQHeap::heapify(int position)
{
    int childLPosition = left(position); //get the position for the left child
    int childRPosition = right(position); //get the position for the right
                                         child
    if (childLPosition > 0 && childRPosition > 0 &&
        heapArray[childRPosition].distance > heapArray[childLPosition].distance)
        childLPosition = childRPosition; //set childLPosition to the larger
                                         of the two children
}

```

```

        if (childLPosition > 0 && heapArray[childLPosition].distance >
            heapArray[position].distance)
        {
            BPQElement temp = heapArray[position];
            heapArray[position] = heapArray[childLPosition];
            heapArray[childLPosition] = temp;
            heapify(childLPosition);
        }
    }
}

void BPQHeap::reset()
{
    for (int i = 0; i < size; i++)
    {
        heapArray[i].distance = THRESHOLD;
        heapArray[i].node = 0;
    }
}

```

Image Output System

Raster.h

```

#include "opencv2/highgui.hpp"
#include "opencv2/core.hpp"
#include "opencv2/imgproc.hpp"
#include <iostream>

using namespace cv;
using namespace std;

/*****
    CLASS: rasterImage

    THIS IS THE CLASS FOR THE OUTPUT IMAGE CONTAINS ABILITY TO SAVE GREYSCALE,
    COLOURIZED, AND BLURRED IMG
    In addition keeps track of the total number of classes within the image.

    Llewellyn Johnston - June 2016
    *****/

class rasterImage
{
private:
    Mat raster;
    int numberOfClasses = 0;
    int inputtedClassNumbers = 0;
    int type;
    void saveColourizedImage(string path);
    void saveGreyscaleImage(string path);

public:
    void saveBlurredImage(string path, int blurRatio);
    void printFirstElement();
    void setSize(int rows, int columns, int nType);
    void setPixelClasses(int x, int y, int cls);
    void saveImage(string path);
}

```

```

        void saveImage(string path, int time);
        Mat getRaster() { return raster; }
};

```

Raster.cpp

```

#include "Raster.h"

using namespace cv;
using namespace std;

/*****
    NAME: setSize
    CLASS: rasterImage

    SETS THE SIZE OF THE RASTER IMAGE AND THE TYPE OF IMAGE BEING CREATED (GREYSCALE
    OR COLOUR)
    Pretty self explanatory...

    Llewellyn Johnston - June 2016
    *****/

void rasterImage::setSize(int rows, int columns, int nType)
{
    if (nType == 0) //if greyscale
        raster = Mat(rows, columns, CV_8UC1);
    else //if colourized
        raster = Mat(rows, columns, CV_8UC3);
    type = nType;
}

/*****
    NAME: setPixelClasses
    CLASS: rasterImage

    SETS THE CLASS NUMBER TO THE PIXELVALUE FOR AN (X, Y) COORDINATE IN THE RASTER
    MAT FOR LATER USE
    If the current class number is larger than the recorded number of classes then
    the number of classes is set to the new highest value.

    Llewellyn Johnston - June 2016
    *****/

void rasterImage::setPixelClasses(int x, int y, int cls)
{
    if (cls > numberOfClasses)
    {
        numberOfClasses = cls;
    }

    if (type == 0)
    {
        raster.at<uchar>(x, y) = cls;
    }
    else
    {
        Vec3b bgr;
        bgr[0] = cls, bgr[1] = 0, bgr[2] = 0;
        raster.at<Vec3b>(x, y) = bgr;
    }
}

```

```

}

/*****
    NAME: printFirstElement
    CLASS: rasterImage

    DEBUGGING FUNCTION TO SEE WHAT WAS ACTUALLY BEING STORED IN THE FIRST POSITION
    OF THE COLOURIZED IMAGE

    Llewellyn Johnston - June 2016
    *****/

void rasterImage::printFirstElement()
{
    int q;
    for (int i = 0; i < raster.rows - 3; i++)
    {
        for (int j = 0; j < raster.cols; j++)
        {
            q = (int)raster.at<Vec3b>(i, j)[0];
            cout << q << " ";
        }
    }
}

/*****
    NAME: setSize
    CLASS: saveColourizedImage

    BASED ON THE PREVIOUSLY ESTABLISHED CLASS NUMBER SET THE COLOUR OF THE PIXEL AND
    STORE IT IN RASTER
    After setting all of the pixels then save the image to the provided path

    NOTE: STILL A WIP (Work in Progress), DOESN'T HANDLE LOTS OF CLASSES VERY WELL

    Llewellyn Johnston - June 2016
    *****/

void rasterImage::saveColourizedImage(string path)
{
    int cls;
    Vec3b pixel;
    for (int i = 0; i < raster.rows; i++)
    {
        for (int j = 0; j < raster.cols; j++)
        {
            cls = (int)raster.at<Vec3b>(i, j)[0];

            if (cls == 0)
            {
                pixel[0] = 0;
                pixel[1] = 0;
                pixel[2] = 0;
            }
            else if (cls == 1)
            {
                pixel[0] = 0;
                pixel[1] = 255;
                pixel[2] = 0;
            }
            else if (cls == 2)

```

```

        {
            pixel[0] = 0;
            pixel[1] = 0;
            pixel[2] = 255;
        }
        else if (cls == 3)
        {
            pixel[0] = 255;
            pixel[1] = 255;
            pixel[2] = 0;
        }
        else if (cls == 4)
        {
            pixel[0] = 255;
            pixel[1] = 0;
            pixel[2] = 255;
        }
        else if (cls == 5)
        {
            pixel[0] = 0;
            pixel[1] = 255;
            pixel[2] = 255;
        }
        else if (cls == 6)
        {
            pixel[0] = 255;
            pixel[1] = 0;
            pixel[2] = 0;
        }
        else
        {
            pixel[0] = 255;
            pixel[1] = 255;
            pixel[2] = 255;
        }

        raster.at<Vec3b>(i, j) = pixel;
    }
}

imwrite(path, raster);

return;
}

/*****
NAME: setSize
CLASS: saveGreyscaleImage

BASED ON THE PREVIOUSLY ESTABLISHED CLASS NUMBER SET THE COLOUR OF THE PIXEL AND
STORE IT IN RASTER
After setting all of the pixels then save the image to the provided path

Llewellyn Johnston - June 2016
*****/

void rasterImage::saveGreyscaleImage(string path)
{
    int pixel = 0;
    for (int i = 0; i < raster.rows; i++)
    {

```

```

        for (int j = 0; j < raster.cols; j++)
        {
            pixel = raster.at<uchar>(i, j);
            pixel = ((float)pixel / numberOfClasses) * 255;
            raster.at<uchar>(i, j) = pixel;
        }
    }

    imwrite(path, raster);

    return;
}

/*****
NAME: setSize
CLASS: saveImage

CALLS THE GREYSCALE SAVE OR THE COLOURIZED SAVE BASED ON THE IMAGE TYPE
ESTABLISHED AT CONSTRUCTION
Pretty self explanatory...

Llewellyn Johnston - June 2016
*****/

void rasterImage::saveImage(string path)
{
    if (type == 0)
        saveGreyscaleImage(path);
    else
        saveColourizedImage(path);
}

/*****
NAME: setSize
CLASS: saveImage

CALLS THE GREYSCALE SAVE OR THE COLOURIZED SAVE BASED ON THE IMAGE TYPE
ESTABLISHED AT CONSTRUCTION
Pretty self explanatory... This one has a time appended to end of image
title

Llewellyn Johnston - June 2016
*****/

void rasterImage::saveImage(string path, int time)
{
    int position = path.find('.');
    string beginning = path.substr(0, position);
    path = beginning + '_' + to_string(time) + path.substr(position);
    if (type == 0)
        saveGreyscaleImage(path);
    else
        saveColourizedImage(path);
}

/*****
NAME: setSize
CLASS: saveBlurredImage

USES OPENCV3.1'S MEDIANBLUR FUNCTION USING THE BLURRATIO COMMAND ARGUMENT, SAVES
TO PROVIDED PATH

```

Pretty self explanatory...

Llewellyn Johnston - June 2016

```
*****/
void rasterImage::saveBlurredImage(string path, int blurRatio)
{
    Mat raster2; // = Mat(raster.rows, raster.cols, CV_8UC3);
    medianBlur(raster, raster2, blurRatio);
    imwrite(path, raster2);
}
```

Main Body Source Code

Source.h

```
#include "PixelTree.h"
#include "classList.h"
#include "Raster.h"
#include <ppl.h>
#include <mutex>
#include <fstream>
#include <iostream>

using namespace cv;
using namespace std;
using namespace concurrency; //USED FOR PARALLELIZATION

void readFile(ifstream&, Mat&, vector<PixelNode*>&, int);
int addTrainingData(string cls, Mat imgD, vector<PixelNode*> &trainingData, int
startRow, int endRow, int startCol, int endCol);
void saveFile(ofstream&, PixelTree*);
int voting(BPQHeap*, int);
string blurredPath(string);
string ASCIIPath(string);
void exportASCII(rasterImage&, string);
Mat classifyImage(Mat, rasterImage&, PixelTree*, classList&, int);

//*****
//*****//
//    MAIN

//
//    ARGUMENTS ORDER: TRAINING DATA, BALANCED, INPUT IMAGE, OUTPUT IMAGE, TYPE,
//    NEIGHBOUR COUNT, BLUR RATIO
//

//
//    TRAINING DATA: The input text file for the training data

//
//    BALANCED: 0 or 1, 0 = no balancing, 1 = balanced

//
//    INPUT IMAGE: The image being classified

//
```

```

// OUTPUT IMAGE: The output image for the classified image (if Blurred will be
saved in same location with added "Blur") //
// TYPE: 0 or 1, 0 = greyscale, 1 = colorized

//
// NEIGHBOUR COUNT: The number of desired neighbourCount to classify on //
// BLUR RATIO: An odd positive number for scale of median blur to be applied, if
not given no blur will be applied //
//
//
// Llewellyn Johnston - June 2016 //

//
//*****
*****//

int main(int argc, char** argv)
{
    if (argc != 9 && argc != 10 && argc != 11) //Check if correct command line
arguments number
    {
        cout << endl << "Program usage: Requires appropriate command line
arguments" << endl;
        cout << endl << "EXE *\"Training Data\" *\"Balanced\" *\"Input Image\"
*\"Output Image\" *\"Image Type\" *\"Neighbour Count\" *\"Blur Ratio\" "
<< endl;
        cout << endl << "DIMENSIONCOUNT: The number of dimensions being looked
at (generally 3 or 4)";
        cout << endl << "TRAINING DATA: The input text file for the training
data";
        cout << endl << "TRAINING BASE IMAGE: The input image file for the
training data";
        cout << endl << "TRAINING TEXTURE IMAGE: The texture image file for the
training data";
        cout << endl << "BALANCED: 0 or 1, 0 = no balancing, 1 = balanced";
        cout << endl << "INPUT IMAGE: The image being classified";
        cout << endl << "OUTPUT IMAGE: The output image for the classified
image";
        cout << endl << "TYPE: 0 or 1, 0 = greyscale, 1 = colorized";
        cout << endl << "NEIGHBOUR COUNT: The number of desired neighbourCount
to classify on";
        cout << endl << "(Optional) BLUR RATIO: An odd positive number for scale
of median blur to be applied, if not given no blur will be applied";
        cout << endl;
        return 1;
    }
    else //If correct command line argument number then try
    {
        int dimensionCount = atoi(argv[1]);
        string trainingDataFile = argv[2];
        string inputTrainingImage = argv[3];
        string inputTextureImage, inputImageFile, outputImageFile;
        bool balanced;
        int type, neighbourCount, blurRatio;
        if (dimensionCount == 4)
        {
            inputTextureImage = argv[4];
            balanced = atoi(argv[5]);
            inputImageFile = argv[6];

```

```

        outputImageFile = argv[7];
        type = atoi(argv[8]);
        neighbourCount = atoi(argv[9]);
        if (argc == 11) //Check if blur option filled in
            blurRatio = atoi(argv[10]);
    }
    else if (dimensionCount == 3)
    {
        balanced = atoi(argv[4]);
        inputImageFile = argv[5];
        outputImageFile = argv[6];
        type = atoi(argv[7]);
        neighbourCount = atoi(argv[8]);
        if (argc == 10) //Check if blur option filled in
            blurRatio = atoi(argv[9]);
    }

    ifstream inFile; //Input file for the training data
    vector<PixelNode*> newNodes; //PixelNode vector for balancing and
creating the KD-Tree
    Mat inputImage; //Input image file, image being classified
    Mat trainingBaseImage;
    Mat trainingTextureImage;
    rasterImage outputImage; //OutputImage of class rasterImage
(Raster.h)
    classList list; //List of classes for the pixels in the output
image, used in voting
    int pixelVal; //The integer valued for the determined className used for
saving the image

    time_t startT, startC;
    time_t endT, endC;

    time(&startT);
    inFile.open(trainingDataFile);
    inputImage = imread(inputImageFile);
    outputImage.setSize(inputImage.rows, inputImage.cols, type); //Set
the size of the output image and the type (greyscale or colourized)

    trainingBaseImage = imread(inputTrainingImage);
    trainingTextureImage = imread(inputTextureImage);

    readFile(inFile, trainingBaseImage, newNodes, dimensionCount);

    PixelTree* pixelTree = new PixelTree(dimensionCount); //Initialized
the KD-Tree using the dimensionCount

    if (balanced == 0) //If the training data has not been balanced then
prune it and balance it
    {
        cout << "Pruning vector... ";
        newNodes = pixelTree->pruneBalanceVector(newNodes,
dimensionCount); //Prune the vector from the inputFile
        cout << "finished" << endl;
        cout << "Balancing vector... ";
        pixelTree->balanceVector(newNodes);
        cout << "finished" << endl;
    }

    cout << "Constructing tree... ";

```

```

        pixelTree->constructTree(newNodes, 0, 0, newNodes.size() - 1);
//Construct the tree using the pruned and balanced vector
        cout << "finished" << endl;

        time(&startC);
        inputImage = classifyImage(inputImage, outputImage, pixelTree, list,
        neighbourCount);
        time(&endC);

        list.printList();

        time(&endT); //end time for processing, tag onto image name

        double totalTime = difftime(endT, startT);
//exportASCII(outputImage, outputImageFile);
        outputImage.saveImage(outputImageFile, totalTime);
        /*if (blurRatio > 0)
            outputImage.saveBlurredImage(blurredPath(outputImageFile),
blurRatio);*/

        delete pixelTree; //Delete the pixelTree
        cout << "Completed" << endl;
        cout << "Classification time: " << difftime(endC, startC) << " seconds."
        << endl;
        cout << "Total time: " << difftime(endT, startT) << " seconds." << endl
        << endl;

        return 0;
    }
}

/*****
NAME: readFile

READS TRAINING DATA IN FROM AN INPUT FILE THAT HAS THE DIMENSIONS VALUE AT THE
TOP AS "K=#"
Reads the values in from the file and then creates new PixelNodes within the
training data vector

Llewellyn Johnston - June 2016
*****/

void readFile(ifstream &inFile, Mat &inputBaseImage, vector<PixelNode*> &trainingData,
int dimensionCount)
{
    string line;
    getline(inFile, line);

    string type;
    int counter = 0;
    int x;
    int y;
    getline(inFile, line);
    while (!inFile.eof())
    {
        vector<float> values;

        int pos = line.find(',');
        type = line.substr(0, pos);
        line = line.substr(pos + 1);
        pos = line.find(',');

```

```

        x = stoi(line.substr(0, pos));
        y = stoi(line.substr(pos + 1));

        Vec3b& bgr = inputBaseImage.at<Vec3b>(y, x);
        vector<float> newPixel;
        newPixel.push_back(bgr[2]);
        newPixel.push_back(bgr[1]);
        newPixel.push_back(bgr[0]);

        trainingData.push_back(new PixelNode(newPixel, type, 0,
        dimensionCount));
        getline(inFile, line);
    }
}

int readFile(ifstream &inFile, vector<PixelNode*> &newNodes)
{
    string string_kValue;
    int int_kValue;
    getline(inFile, string_kValue);
    //int_kValue = stoi(string_kValue.substr(2, string_kValue.length() - 2));
    int_kValue = 3; //TO DO - POSITION HOLDER

    string line;
    string type;
    int counter = 0;
    int commaPosition1;
    int commaPosition2;
    int commaPosition3;
    while (!inFile.eof())
    {
        vector<float> values;

        getline(inFile, line);
        commaPosition1 = line.find(',');
        commaPosition2 = line.find(',', commaPosition1 + 1);
        commaPosition3 = line.find(',', commaPosition2 + 1);
        type = line.substr(0, commaPosition1);
        for (int i = 0; i < int_kValue; i++)
        {
            values.push_back(stoi(line.substr(commaPosition1 + 1,
            commaPosition2 - commaPosition1 - 1)));
            values.push_back(stoi(line.substr(commaPosition2 + 1,
            commaPosition3 - commaPosition2 - 1)));
            values.push_back(stoi(line.substr(commaPosition3 + 1, line.size()
            - commaPosition3 - 1)));
            //cout << "i";
        }

        newNodes.push_back(0);
        newNodes[counter] = new PixelNode(values, type, 0, int_kValue);
        counter++;
    }

    return int_kValue;
}

/*****
NAME: addTrainingData

```

ADDS THE TRAINING DATA FROM THE INPUT IMAGE GIVEN THE PROVIDED COORDINATES
PARAMETERS
Adds blue, green, and red values of the pixels within the bounds of the
coordinates

Llewellyn Johnston - June 2016

*****/

```
int addTrainingData(string cls, Mat imgD, vector<PixelNode*> &trainingData, int
startRow, int endRow, int startCol, int endCol)
{
    for (int row = startRow; row < endRow; row+=5)
    {
        for (int col = startCol; col < endCol; col+=5)
        {
            Vec3b& bgr = imgD.at<Vec3b>(row, col);
            vector<float> newPixel;
            newPixel.push_back(bgr[2]);
            newPixel.push_back(bgr[1]);
            newPixel.push_back(bgr[0]);

            trainingData.push_back(new PixelNode(newPixel, cls, 0, 3));
        }
    }
}
```

NAME: voting

USES THE CLASSES BPQHEAP AND THE NEIGHBOUR COUNT TO RETURN THE MOST COMMON CLASS
WITHIN THE HEAR
Returns the string name of the class

Llewellyn Johnston - June 2016

*****/

```
int voting(BPQHeap* classes, int kCount)
{
    vector<string> classNames; //Keeps track of all of the different class names
                              found in the BPQHeap
    vector<int> timesFound; //Corresponding vector to classNames to
                              keep track of how many times the corresponding
                              className has been seen

    bool found;
    string type;

    for (int i = 0; i < kCount; i++)
    {
        found = false;
        type = classes->getElement(i)->getType();

        for (int j = 0; j < classNames.size(); j++)
        {
            if (type == classNames[j])
            {
                timesFound[j]++;
                found = true;
                break;
            }
        }
    }
}
```

```

        if (!found)
        {
            classNames.push_back(type);
            timesFound.push_back(1);
        }
    }

    int max = 0;
    int maxPosition;

    for (int i = 0; i < classNames.size(); i++)
    {
        if (timesFound[i] > max)
        {
            maxPosition = i;
            max = timesFound[i];
        }
    }

    type = classNames[maxPosition];
    //VERY SPECIFIC FOR FIREMAP, WILL ALTER FOR OTHER USES
    //if (type == "BURN" || type == "CANCER")
    if (type == "Unburn")
        return 0;
    else if (type == "Burn")
        return 1;
    else if (type == "DIRT")
        return 2;
    else if (type == "VEG")
        return 3;
    else if (type == "CONIFER")
        return 4;
    else if (type == "DECIDUOUS")
        return 5;
    else if (type == "BRUSH")
        return 6;
    else if (type == "HERBACEOUS")
        return 7;
    else
        return 8;
}

/*****
NAME: blurredPath

ADDS THE WORD 'BLUR' TO END OF THE PROVIDED PATH FOR THE OUTPUT IMAGE FOR THE
BLURRED OUTPUT IMAGE
Pretty self explanatory...

Llewellyn Johnston - June 2016
*****/

string blurredPath(string path)
{
    string blurPath;
    int position = path.find('.');
    blurPath = path.substr(0, position) + "Blur" + path.substr(position,
    path.length() - position);
    return blurPath;
}

```

```

/*****
    NAME: ASCIIPath

    CREATES THE FILE PATH FOR THE ASCII EXPORT
    Appends ASCII.txt to the end of the given image output path

    Llewellyn Johnston - August 2016
    *****/

string ASCIIPath(string path)
{
    string asciiPath;
    int position = path.find('.');
    asciiPath = path.substr(0, position) + "ASCII.txt";
    return asciiPath;
}

/*****
    NAME: exportASCII

    THE EXPORT FUNCTION FOR THE ASCII, WRITES THE CLASSIFIED IMAGE INTO A TEXT
    DOCUMENT WITH THE CLASSES LISTED (basically an un-normalize or compressed
    greyscale image)

    Llewellyn Johnston - August 2016
    *****/

void exportASCII(rasterImage& outputImage, string name)
{
    ofstream outFile;
    outFile.open(ASCIIPath(name));
    Mat raster = outputImage.getRaster();

    //ArcGIS header information for ASCII export
    outFile << "NCOLS " << raster.cols << endl << "NROWS " << raster.rows << endl;
    //TO DO-----
    outFile << "XLLCORNER " << "00000" << endl << "YLLCORNER " << "00000" << endl;
    //Filler information for now, will eventually red from file
    outFile << "CELLSIZE 0.2" << endl << "NODATA_VALUE -32768" << endl << endl;

    for (int row = 0; row < raster.rows; row++)
    {
        for (int col = 0; col < raster.cols; col++)
        {
            outFile << (int)raster.at<uchar>(row, col) << " ";
        }
        outFile << endl;
    }
}

/*****
    NAME: classifyImage

    PARALLEL FUNCTION TO CLASSIFY THE INPUT IMAGE USING THE TRAINING DATA
    Traverses through each pixel row by column assigning classifications to each
    based on the kNN algorithm and the given training data.

    Llewellyn Johnston - August 2016
    *****/

```

```

Mat classifyImage(Mat inputImage, rasterImage& outputImage, PixelTree *pixelTree,
classList &list, int neighbourCount)
{
    mutex threadLock;
    int pixelVal;

    parallel_for(0, inputImage.rows, 1, [&](int row)    //For loop cycling through
                                                         each of the rows in the image
    {
        string className;    //Voted className for the pixels in the output
                             image, voted on using the classList
        for (int col = 0; col < inputImage.cols; col++)    //For loop cycling
                                                         through each of the
                                                         columns in the image
        {
            BPQHeap *heap = new BPQHeap(neighbourCount);
            Vec3b& bgr = inputImage.at<Vec3b>(row, col); //Grab the
                                                         pixel at position
                                                         (row, col) and place
                                                         it in bgr

            vector<float> newPixel;

            newPixel.push_back(bgr[2]); //Push the RED value of pixel onto
                                       the newPixel vector
            newPixel.push_back(bgr[1]); //Push the GREEN value of the pixel
                                       onto the newPixel vector
            newPixel.push_back(bgr[0]); //Push the BLUE value of the pixel
                                       onto the newPixel vector

            pixelTree->kNearestNeighbourNodes(heap, newPixel); //Get the k
                                                         nearest
                                                         neighbours and
                                                         store into
                                                         heap

            pixelVal = voting(heap, neighbourCount);

            threadLock.lock();    //threaded mutex variable to lock the
                                 writing sections of the parallel processing
                                 to avoid deadlocks and access violation
                                 errors
            outputImage.setPixelClasses(row, col, pixelVal); //Set the
                                                         pixel's class
                                                         in the output
                                                         image using
                                                         the pixelVal

            threadLock.unlock(); //unlock mutex variable for other threads
                                 to proceed

            delete heap;
        }
    });
    return outputImage.getRaster();
}

```