

NORTHWEST NAZARENE UNIVERSITY

Identifying Prostate Cancer in Biopsy Images using a
Support Vector Machine and Decision Tree

THESIS

Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements

for the degree of

BACHELOR OF SCIENCE

Hannah L Moxham

2019

THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE

Hannah L Moxham
2019

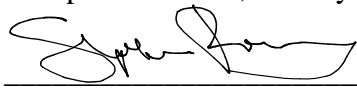
Identifying Prostate Cancer in Biopsy Images using a
Support Vector Machine and Decision Tree

Author: 

Hannah L Moxham

Approved: 

Barry L. Myers, Ph.D., Professor, Department of Mathematics &
Computer Science, Faculty Advisor

Approved: 

Stephen Riley, Ph.D., Professor, School of Theology & Christian
Ministry, Second Reader

Approved: 

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

ABSTRACT

Identifying Prostate Cancer in Biopsy Images using a Support Vector Machine and Decision Tree.

MOXHAM, HANNAH (Department of Mathematics and Computer Science),
MYERS, DR. BARRY (Department of Mathematics and Computer Science).

Prostate cancer is the second most common cancer in men. Its high five-year relative survival rate hinges on identification of the cancer, especially before it spreads. A negative misdiagnosis can be deadly, which creates need for a consistently accurate method of identification. This research sought to develop a computer vision software tool that, given a digital image of a stained prostate biopsy, locates any malignant glands present in the image. A three-step process was devised for this: first, run supervised machine learning classifiers to mark the key cellular structures that point to adenocarcinoma of the prostate—nuclei, nucleoli, and lumina. Second, analyze those structures for key traits such as size and clustering. Third, use these derived traits in a second round of classification to locate cancerous regions. A support vector machine and decision tree were used for step one with reasonable success—nuclei and lumina were found with high accuracy, but nucleoli identification was troublesome. Better accuracy than this is desired. Future work includes determining the value of continuing with this three-step method, and if so refining step one and completing steps two and three. Otherwise, a new classification algorithm such as a convolutional neural network will be investigated.

ACKNOWLEDGEMENTS

I want to thank Dr. Joseph Kronz for taking time from his work to teach me about the visual identification process for prostate cancer and to obtain imagery for testing and training our classifiers. I want to thank Idaho INBRE for the support and funding through an Institutional Development Award (IDeA) from the National Institute of General Medical Sciences of the National Institutes of Health under Grant #P20GM103408. I want to thank Dr. Dale Hamilton and the undergraduate summer research team of 2018 for encouragement, suggestions, and listening ears as I worked on this project. I want to thank Dr. Barry Myers for his mentorship throughout this project from beginning to end. I want to thank Dr. Stephen Riley for his excitement about my work and his own work as my second reader.

TABLE OF CONTENTS

Title Page	i
Signature Page	ii
Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Figures	vi
Thesis Body	1
Background: Prostate Cancer	1
Background: Machine Learning	4
Algorithm Selection	6
Development: Normlization	8
Developmet: Nuclei and Lumina	9
Development: Nucleoli	11
Results: Final Process	14
Results: Accuracy	17
Future Work	18
References	20
Appendices	22
Appendix A: Classified Images	22
Appendix B: Code	28
Batch file - Classification	28
Batch file - Accuracy	29
Normalization program	30
Texture generation program	37
Accuracy program.	67

LIST OF FIGURES

Figure 1 - Image A, a prostate biopsy.....	2
Figure 2 - Important cellular structures labeled.....	3
Figure 3 - Benign versus malignant.....	4
Figure 4 - Binary classification of nuclei.....	5, 10
Figure 5 - Min-max normalization formula.....	9
Figure 6 - Differences in lighting using images A and F.....	9
Figure 7 - A test of average accuracies for nucleus classification.....	10
Figure 8 - Binary classification of lumina.....	11
Figure 9 - FireMAP texture example.....	12
Figure 10 - Contrast-based texture.....	13
Figure 11 - Contrast-based texture in color.....	14
Figure 12 - Original.....	15
Figure 13 - Normalized.....	15
Figure 14 - Nuclei.....	15
Figure 15 - Lumina.....	15
Figure 16 - Nucleoli (zoomed).....	16
Figure 17 - Final (zoomed).....	16
Figure 18 - Final classification image.....	16
Figure 19 - Accuracy results.....	17
Figure 20 - Accuracy averages.....	17

IDENTIFYING PROSTATE CANCER IN BIOPSY IMAGES USING A SUPPORT VECTOR MACHINE AND DECISION TREE

BACKGROUND: PROSTATE CANCER

Cancer is an extremely serious disease of the early twenty-first century. More than one in three individuals living in the United States will be diagnosed with cancer during their lifetime (“Cancer Statistics”). Pathologists like Dr. Joseph Kronz dedicated hours each day to scouring slides of biopsy specimens under a microscope in search of cancerous cells. In his work, Dr. Kronz sees an opportunity for improvement. Even the most expert pathologist will make mistakes and misdiagnose a case, but a software screening tool could help remedy this error. Such a tool could be used for pre-screening or post-screening, to point a pathologist in the direction of the right diagnosis. This is the motivation behind this project—improvement in accuracy of diagnosis—as cancer can only be beaten after it is first found.

This project focused on identifying acinar adenocarcinoma of the prostate, cancer in the epithelial cells of the small glands within the prostate (“Adenocarcinoma”), for three reasons. First, developing a prototype that addressing anything more than a narrow sample of cancers would be too much for a single project. Second, this type of prostate cancer is the specialty of collaborator Dr. Kronz, which allows for quick learning about the problem the tool must address. Third, prostate cancer is the second most common cancer in men (“Key Statistics for Prostate Cancer”), which gives many image samples for testing and many opportunities to improve diagnosis once there is a finished product. Prostate cancer may have a high five-year relative survival rate (“Survival Rates for Prostate Cancer”), but this is contingent on finding the cancer first. Even a small improvement in diagnosis accuracy offered by screening via software could make a huge difference in the life of an individual.

The task Dr. Kronz faces of locating cancer under a microscope is a challenging one. The only way a program could successfully complete this task itself is if the programmer has a decent understanding of the process, thus the need to work closely with Dr. Kronz and learn from him. Figure 1 is an example of what the pathologist or screening tool is first presented with, a cross-section of a needle-core biopsy of a prostate.

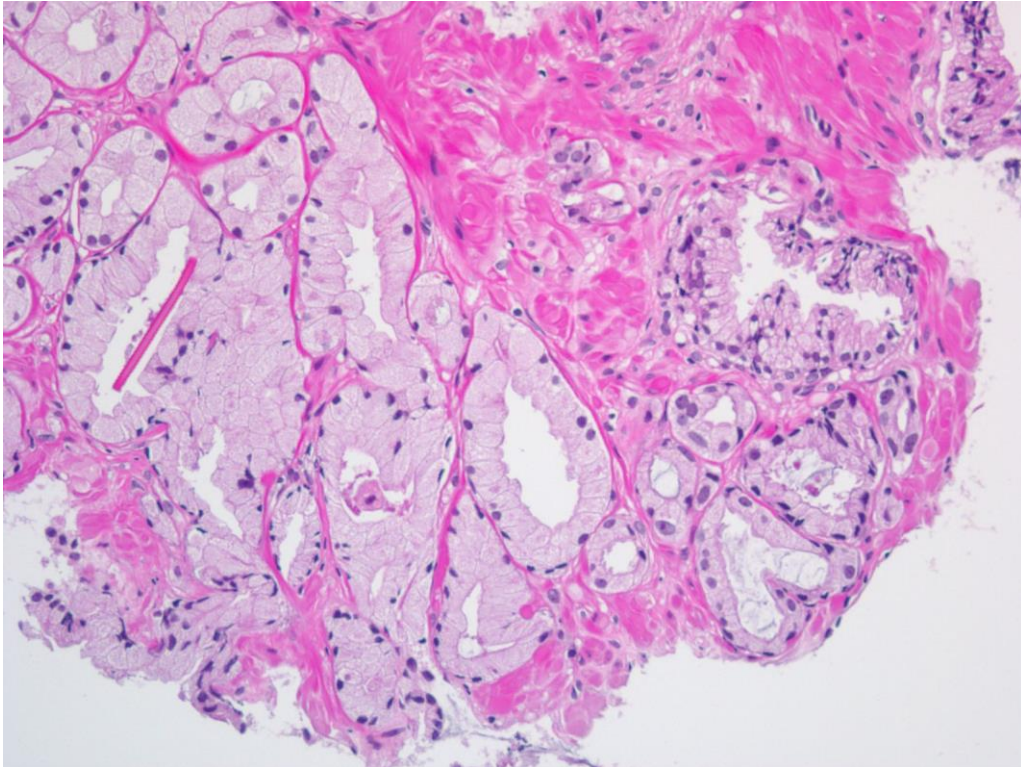


Figure 1 - Image A, a prostate biopsy

Figure 2 gives a labeled subsection of Figure 1 detailing the cellular anatomy. The prostate gland itself contains many small gland openings that secrete prostate fluid. The tube running through one of these glands, seen as a circular or irregularly-shaped opening in the biopsy cross-section, is a lumen (plural lumina). Like any cells, those making up the other epithelial layer of the gland contain cytoplasm and a nucleus (plural nuclei). The nucleolus (plural nucleoli), when visible, can be seen as a darker dot inside the nucleus. Since this project is targeting cancer in the epithelial tissue of the glands, the nuclei, nucleoli, and lumina are the most important to examine.

Stroma, the supporting connective tissue filling the space between glands (“Stroma”), is not of interest for this kind of cancer. H&E (haematoxylin and eosin) staining is used to color the biopsy and bring out the details. The base-like haematoxylin reacts with basophilic structures like nuclei and nucleoli to dye them blue-purple, and the acid-like eosin reacts with acidophilic structures such as the stroma, dyeing them red-pink (Parry).

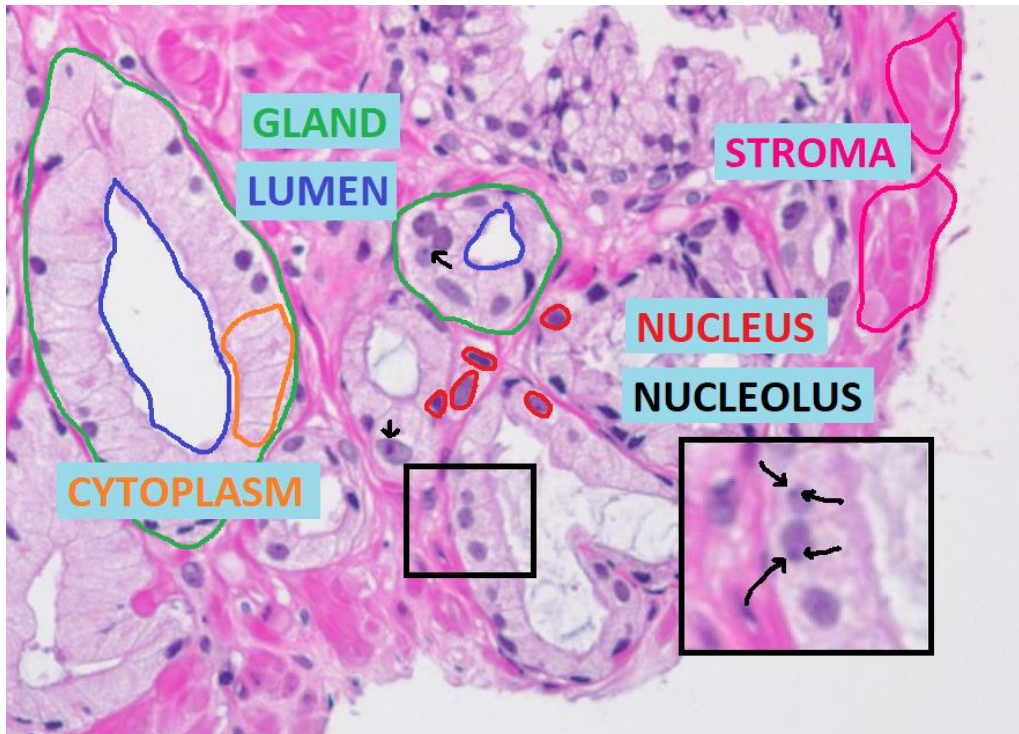


Figure 2 - Important cellular structures labeled

While there are more exceptions than rules when it comes to something as complicated as cancer, there are a few guiding principles that point an examiner towards their diagnosis. A healthy, benign gland will have many small nuclei around a large, irregularly shaped lumen. A cancerous gland will have large, round nuclei with easily visible nucleoli surrounding smaller lumina closer to a circle in shape. The more the cancer progresses, the more the healthy lumen breaks down into small lumina bubbles. In Figure 3, benign glands are outlined in green; the rest of the image is high-grade cancer

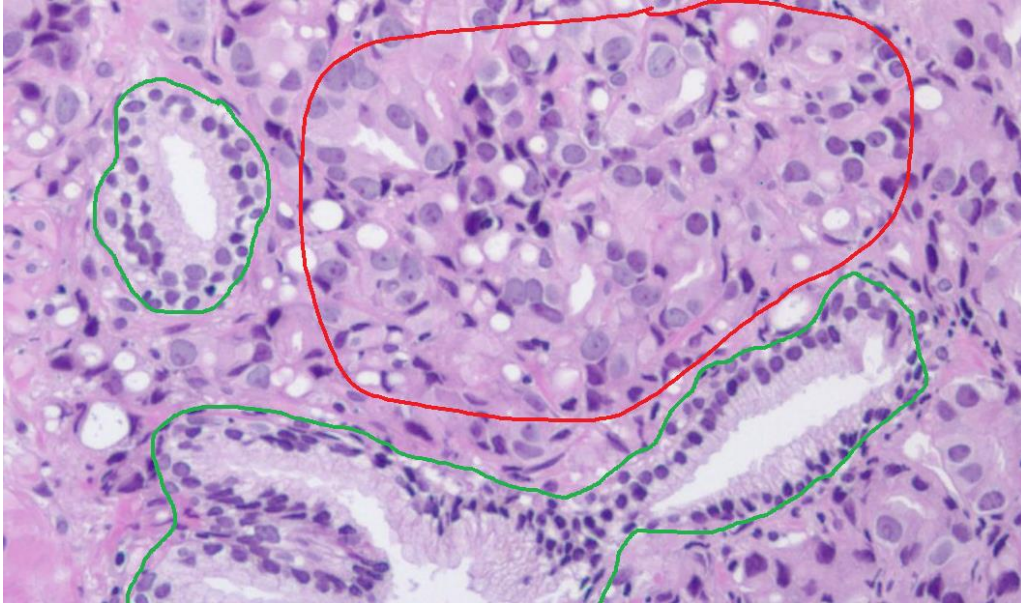


Figure 3 - Benign versus malignant

BACKGROUND: MACHINE LEARNING

Given how complicated the cancer identification process is and how much of it relies on trained human intelligence rather than strict logic, machine learning stood out as the method of choice to underly the screening tool. Machine learning is a subset of artificial intelligence that allows a program to perform its processing through observing and analyzing patterns rather than following explicit instructions (Team). The program “learns” the patterns and acts on them with minimal guidance, which is ideal for modeling a human thought process. Within machine learning, there are two categories: supervised and unsupervised learning. The first is best for cases in which the desired classifications (such as benign and malignant) are known, while the second is best for uncovering new inferences and correlations (Soni). With the clear project goal of identifying cancer, supervised learning was the best choice. Supervised learning works by first training the classifier using training data that contains labeled examples of the classes it must identify. One class may be labeled “benign” and the other “malignant.” The algorithm behind the classifier allows the program to learn the traits that make an object one class or the other and

determine a method of sorting objects into their classes. Classifications can be binary (sorting into two classes) or ternary (sorting into three or more classes).

When classifying images, the input can be taken two ways. Either the whole image is treated as an object to classify, with the pixels making up millions of data points about the object, or each individual pixel is treated as an object to classify, with the red-green-blue (RGB) color intensity values making up the data points about the object. The latter method of one pixel at a time was used for this project. In this case, training data looks like a list of pixels, some labeled Class 1 and some labeled Class 0. Class 1 could be “malignant” while Class 0 is “benign,” or any other desired labels. Any number of math-based algorithms could be used to analyze the training data and create a quick means of sorting objects into classes. The classifier then receives an input image it has never seen before and classifies each pixel based on guidelines extracted from the training data, recoloring those put in Class 1 as white and those put in Class 0 as black. The new black-and-white image becomes the output. An example of this can be seen in Figure 4.

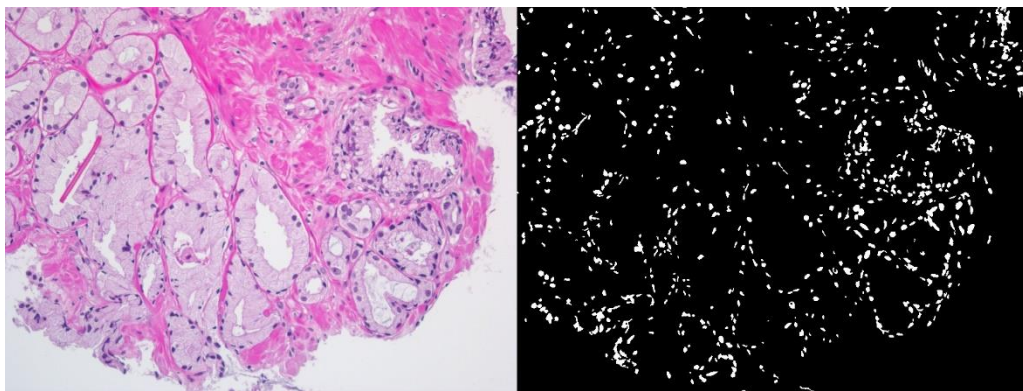


Figure 4 - Binary classification of nuclei (white)

While the output may look pretty, its accuracy is of greater importance. The classification is compared against validation data, created by a human manually labeling a set of objects, in this case pixels, as Class 1 or Class 0. Accuracy is computed using a confusion matrix, which

includes information on how a predicted class was correct or wrong rather than simply whether it was right or wrong. This results in four categories for a binary classification—true positives, Class 1 pixels classified as Class 1; true negatives, Class 0 pixels classified as Class 0; false positives, Class 0 pixels classified as Class 1; and false negatives, Class 1 pixels classified as Class 0. Accuracy is the number correct—true positives and negatives—out of the total pixels listed in the validation data. The extra information of how a classification was wrong or right can be crucial. Sometimes a false negative is much worse than a false positive. In the realm of cancer, a false negative, a misdiagnosis of an individual as cancer free, can be deadly.

ALGORITHM SELECTION

There are many machine learning algorithms to choose from to make up the backbone of the screening tool. After some exploration, the options were narrowed down to two paths: the SVM or the CNN. The SVM path includes a suite of classifiers from previous years of research on the FireMAP project, such as the k-nearest neighbors (kNN), support vector machine (SVM), and the Iterative Dicotomizer 3 (ID3), also called a decision tree. While FireMAP's goal was to use classifiers to determine the burn severity and extent of wildland fires (Hamilton, D. and A. Van Aardt), machine learning classifiers have wonderful flexibility—all it takes is new training data to start working on a new problem. Classifiers initially intended for wildland fire analysis are full of potential for cancer diagnosis. While these classifiers have the advantage of already being implemented and have proven their worth with successful results for FireMAP, there is doubt about their ability to handle the complexity of cancer. These classifiers all take pixels as input rather than the image. A pixel is essentially a dot of color. Color is enough to distinguish burnt terrain (black) from unburnt terrain (brown or green) easily, but cancer is much more complex than color. There is not enough information contained in an RGB pixel to distinguish between a benign pixel and a malignant pixel. The SVM path would require a more sophisticated

method of identifying cancer than simply training the classifiers on benign versus malignant pixels.

The convolution neural network (CNN) is a promising alternative to the pixel-based classifiers from FireMAP. This algorithm is particularly useful for the problem of computer vision, teaching computers to “see” and identify objects in a picture like a human eye (Saha). Unlike an SVM or decision tree, a CNN take a whole image, twelve million pixels in the case of a biopsy image, as a single input. Through a complex series of matrix operations, key features of the image are preserved while the rest is compacted into a much smaller data set. This smaller set is run through a neural network, another type of classifier like the SVM, to produce the final classification of the image as a whole—cancer or no cancer. Through the convolutional matrix operations that work on many pixels at once, a CNN “looks” at the image on a larger scale than the SVM, allowing it to identify objects, such as a car, cat or malignant nucleus. While highly suited for object identification, CNNs requires a multitude of images to use for training data (Saha). It must be fed thousands of pictures of malignant and benign glands to learn the difference, and the collection of images sourced from Dr. Kronz round out to about two hundred. While the SVM and similar classifiers are pixel-based, which is a large handicap, this can be dealt with using creative workarounds. No amount of creative programming will produce more unique images to train a CNN. This was the biggest factor in the final decision to use the FireMAP suite.

Knowing that pixel-based classification of the FireMAP suite would not be enough by itself to identify cancer, a process of classification and analysis was devised. There are three steps to this process: (1) identify key structures, (2) analyze key structures, and (3) classify on new attributes. First, the classifiers are used to mark out the key structures for identifying cancer:

the nuclei, nucleoli, and lumina. Nuclei are clearly purple, and lumina are clearly white, while nucleoli are clearly darker than the surrounding nucleus. With a little help from texture metrics on the last structure, to be discussed later, all of these are within the realm of classification a pixel at a time. Second, once the key structures are found and labeled correspondingly, they are analyzed for traits important for identifying cancer, such as size and location relative to each other. After this step, a pixel will be tied to information such as type of structure, structure size and relative clustering as well as its RGB color intensities. In the third step, the newly derived attributes are run through a classifier again, this time trained on benign versus malignant, to give the final verdict on whether the biopsy contains cancer or not.

DEVELOPMENT: NORMALIZATION

With the problem defined, the algorithms selected, and a path sketched from the first input to the final output, the development work began. Before jumping into step one of classifying the key cellular structures, there was a small matter to deal with. Not all biopsy images have the same color palette, as seen in Figure 5. This is due to changes in illumination of the microscope and variation due to staining. Too much variation can ruin a classifier's output. The classifiers are trained using one image—this is what it expects to “see” and classify. Anything too far outside the range of the training data is a wild card that the classifier may or may not classify correctly. One way to deal with this is by minimizing the variation between images using normalization. This pre-processes the image before handing it to the classifier, shifting the range of values of the input image to match those of the training image. Min-max normalization was used, with is described by the formula in Figure 6, where value-old is the value to be normalized, min-old is the minimum value of the original range, max-old is the maximum of the original range, with min-new and max-new similar but for the new range. Improvement in accuracies due to normalization will be discussed in later sections.

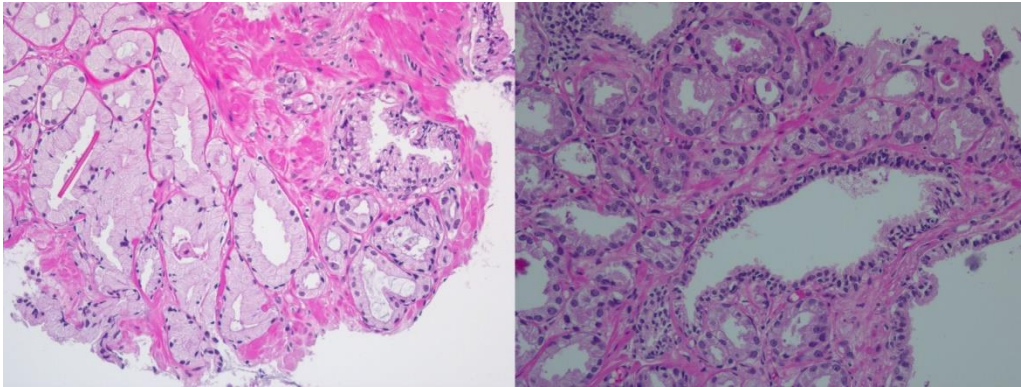


Figure 5 - Min-max normalization formula (Han, Jiawei et al)

$$value_{new} = \frac{value_{old} - min_{old}}{max_{old} - min_{old}} * (max_{new} - min_{new}) + min_{new}$$

Figure 6 - Differences in lighting using images A and F

Accuracy tests for normalization also helped narrow down which of the many classifiers in the FireMAP suite is best suited for cancer diagnosis. There are a number of algorithms available from FireMAP: a decision tree, kNN, and SVM with three different kernels—chi-squared, RBF, and linear. (Kernels are mathematical transformations used on the training data to help improve classification in an SVM (Afonja)). All in all, this results in five classifiers to use. Normalization results eliminated the linear kernel from the list. While the decision tree, SVM-Chi2, and SVM-RBF were consistent in their accuracy, the SVM-Linear was not; while it was generally similar in accuracy to the others, it would too often have inexplicably lower or higher accuracy in some cases. Given this volatility, it was knocked from the running of best FireMAP classifier for prostate cancer.

DEVELOPMENT: NUCLEI AND LUMINA

Since the nuclei in the images are significantly more purple than the rest of the image, the classifiers had little trouble accurately classifying them. A denoise tool from the FireMAP suite

was used on the binary classification images to clean them up. Figure 4 is repeated below as an example of nucleus classification with 93.88% accuracy.

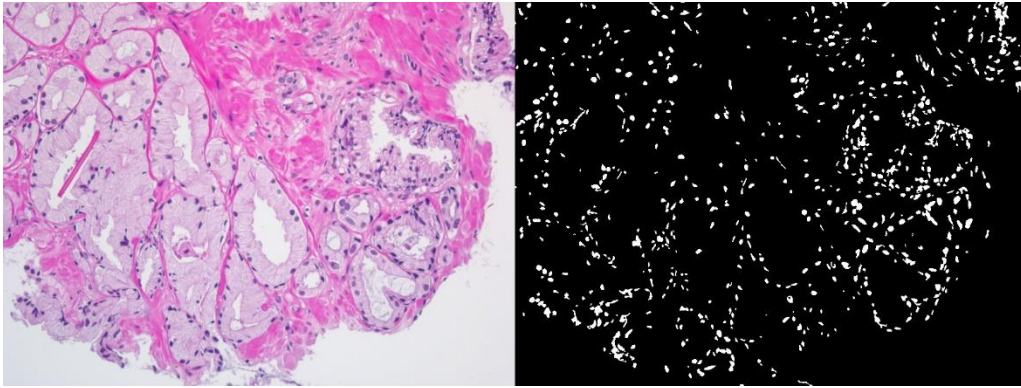


Figure 4 repeated - Binary classification of nuclei

With how stable nucleus accuracy was, it made for a good standard to run another comparison test between the FireMAP classifiers. Figure 7 shows the results of this test, in which accuracy of the five classifiers was averaged across three different sets of training data. At about twenty percentage points lower, it was clear that the kNN would not be useful, so it was dropped from the testing. The remaining three—decision tree, SVM-Chi2, and SVM-RBF, were comparable in accuracy throughout the project, so no more classifiers were dropped.

DecTree	SVM-Chi2	SVM-RBF	kNN
94.887	90.3394	91.6487	70.0455

Figure 7 - A test of average accuracies for nucleus classification

Like the nuclei, lumina have a clear color difference when compared to the rest of the image. They take on the off-white color of the slide background. As a result, the accurate classification of the lumina was easy to achieve with all three of the classifiers. The denoise tool was run again to clean the image. An example with 94.26% accuracy is shown below in Figure 8. With two of three key cell structures located, work began on the final structure—nucleoli.

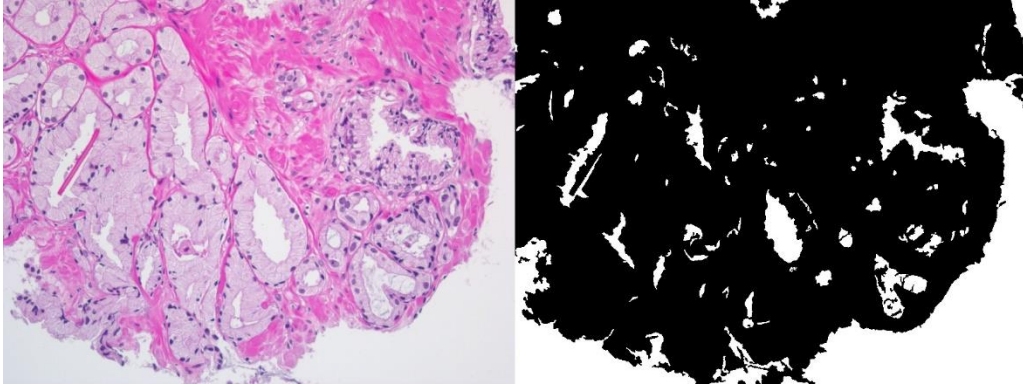


Figure 8 - Binary classification of lumina

DEVELOPMENT: NUCLEOLI

Nucleolus classification proved much more difficult than lumen and nucleus classification. While nuclei and lumina are distinct colors compared to the rest of the image, the nucleoli are not. Like the nucleus, they are purple. A simple classification based on classes of “nucleolus” and “other” ended up being a second means of classifying nuclei. To the human eye, a nucleolus is identified by being a darker spot within a nucleus. As accurate nucleus classifications were already on hand, the search area can be limited to pixels already marked “nucleus.” The problem becomes locating a darker spot, which is a relative term. Knowing whether a pixel is darker requires looking at the pixels around it, which requires more than the basic RGB values of a single pixel. This is where texture comes into play.

Texture is a spatial metric computed for a center pixel using values of pixels around it. A pixel in the middle of a solid color object will have a different texture value than one in the middle of a grassy area. This allows something the human eye would subjectively describe as jagged, soft, coarse, etc to be vaguely represented as numerical value, something more useful to a program. Different kinds of texture metrics exist, using different math behind the calculation of the value (Gogul09). Texture was used in FireMAP to increase accuracy of classification (Hamilton, D. et al), which suggested that texture may be useful for the cancer diagnosis project.

Specifically, it may help with finding nucleoli, since texture is based on looking at the pixels around the one in question, and a nucleolus is visually identified by being darker than the pixels around it.

The texture generation tool for FireMAP was pulled out of the suite and run on the original image to create a second, gray-scale texture image, with the gray color value of the pixel in the texture image representing the texture value of the corresponding pixel in the original image (see Figure 9) Then, both the original image and the texture image were fed into the SVM and decision tree trained on nucleolus versus nucleus. The improvement in accuracy was minimal, however. Even testing six different kinds of texture proved unhelpful. Examination of the texture image revealed that there was no distinct difference where the nucleoli were located in the texture image that the classifier would be able to leverage.

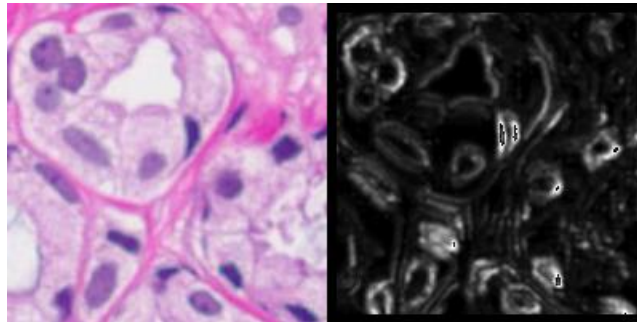


Figure 9 - FireMAP texture example

While the specific implementation of texture generation used in FireMAP did not aid in classification of nucleoli, the idea of texture still had potential. A new texture metric was developed from scratch, based on the idea of contrast. By treating pixels as vectors, the difference between them can be found, also as a vector. The length of that vector can represent the magnitude of difference between the color of the two pixels—a larger number indicates a larger change in color. If the center pixel in question is in a nucleolus, then stepping a few pixels to any direction should be outside the nucleolus, part of the regular nucleus. If the center pixel in

question is in the regular nucleus, then stepping a few pixels to any direction should still be in the nucleus—at most one of those directions will land inside the nucleolus and averaging the differences of all four directions will minimize the difference between the center nucleus pixel and side nucleolus pixel. Using the contrast vector formula described above should result in a larger texture value for the center pixel in the nucleus, which has a change from dark purple to light purple in four directions, than the texture value of a center pixel in the nucleus, which has one change if that from dark purple to light purple. With this idea laying the groundwork, a new texture generation tool was developed. Figure 10 gives an example of the texture image output by this new, contrast-based texture.

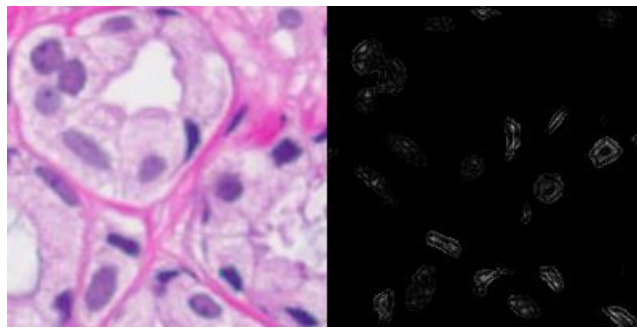


Figure 10 - Contrast-based texture

Here, visibly lighter dots in the texture image correspond the nucleoli in the original image, meaning this texture did find the nucleoli; but, there are also many other light-colored shapes that do not correspond to any nucleoli. The classifier would often find the nucleoli, as well as a lot of other junk. Thankfully, denoising was able to eliminate the junk on the principle that nucleoli are fairly small and take up maybe 15% of a nucleus, while the junk is larger and takes up greater than 15% of the nucleus.

While the classifiers were able to find nucleoli now, whereas using no texture or the FireMAP texture resulted in nothing, the accuracy was not high. It was between 60% and 70% on most cases. Another way to improve upon nucleoli classification was devised. Previously, the

contrast of color in pixels had been looked at in a single dimension—the RGB values were converted to grayscale. Looking at red, green, and blue contrasts separately would provide more information about the contrast, perhaps useful information. One of the areas that the contrast-based texture was catching as junk, that the classifiers were then misclassifying, was the edge of the nucleus, a change from light purple to pink. The only change that the texture needed to catch was dark purple to light purple, the sign of a nucleolus. While these changes looked similar in grayscale, they look different in color. From pink to purple has higher changes in the red and blue values than green, but from purple to dark purple has similar changes in all colors. The texture generating tool was modified to perform its calculations on the red, green, and blue values separately rather than combining them into a single gray value. This resulted in a texture image like that of Figure 11, which does not look like much to the human eye, but close examination reveals faint, dark colors. This color image was then fed into the classifier, without the original image, to produce a nucleoli classification. The improvement in accuracy will be discussed in the upcoming results section.

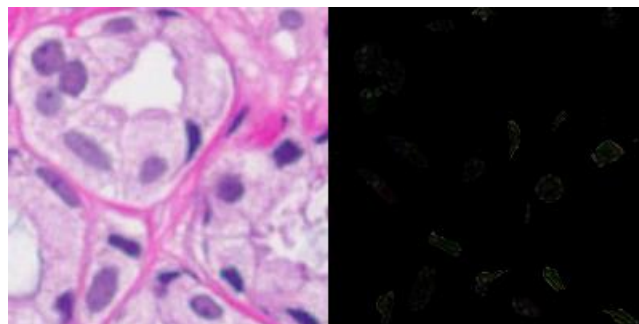


Figure 11 - Contrast-based texture in color

RESULTS: FINAL PROCESS

Many steps made up the process of identifying key structures, completed using a variety of programs and tools. A batch file was used to run each program in series through the command line interface. First, the input image was normalized (Figure 13). Then the nucleus and lumen

classifications were created using the SVM-Chi2 and denoised (Figures 14, 15). A colored texture image was generated and used to classify the nucleoli on top of the nuclei classification, creating an image with white nuclei and red nucleoli on a black background (shown zoomed in in Figure 16). This was also denoised. Finally, the classification images were merged into a single image, with lumina bright green, nuclei as red, and nucleoli as blue, with the rest black (Figures 17, 18). Accuracy calculations for nuclei, lumina, and nucleoli individually were also run as part of the batch file.

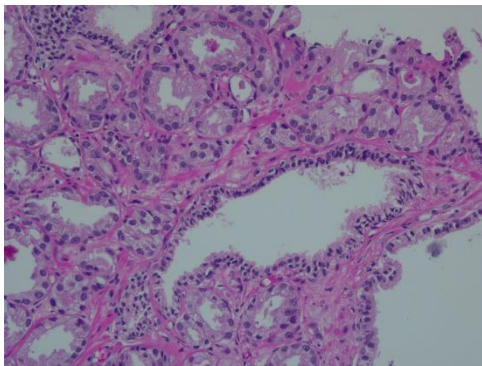


Figure 12 - Original

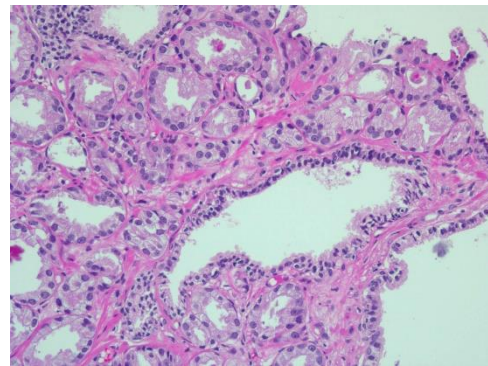


Figure 13 - Normalized

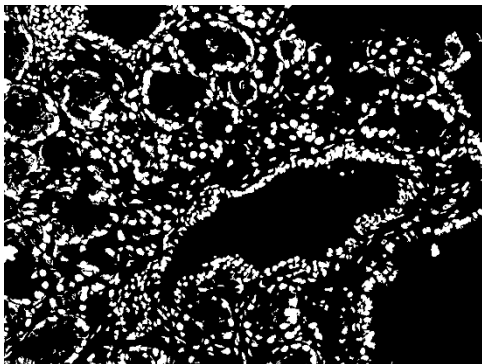


Figure 14 - Nuclei



Figure 15 - Lumina

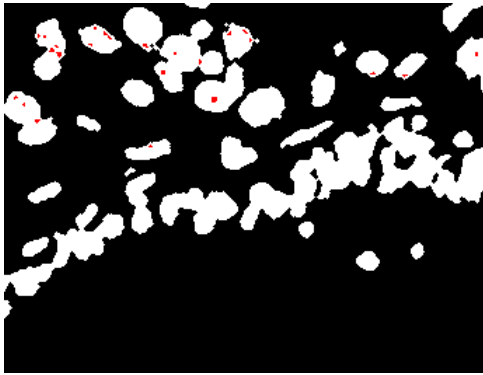


Figure 16 - Nucleoli (zoomed)

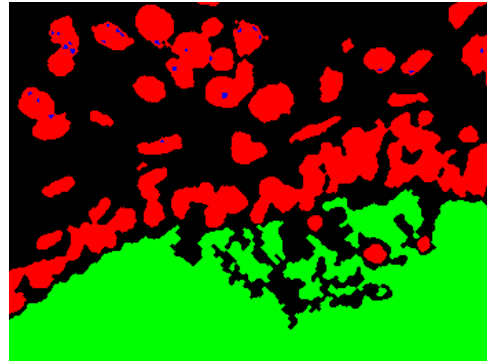


Figure 17 - Final (zoomed)

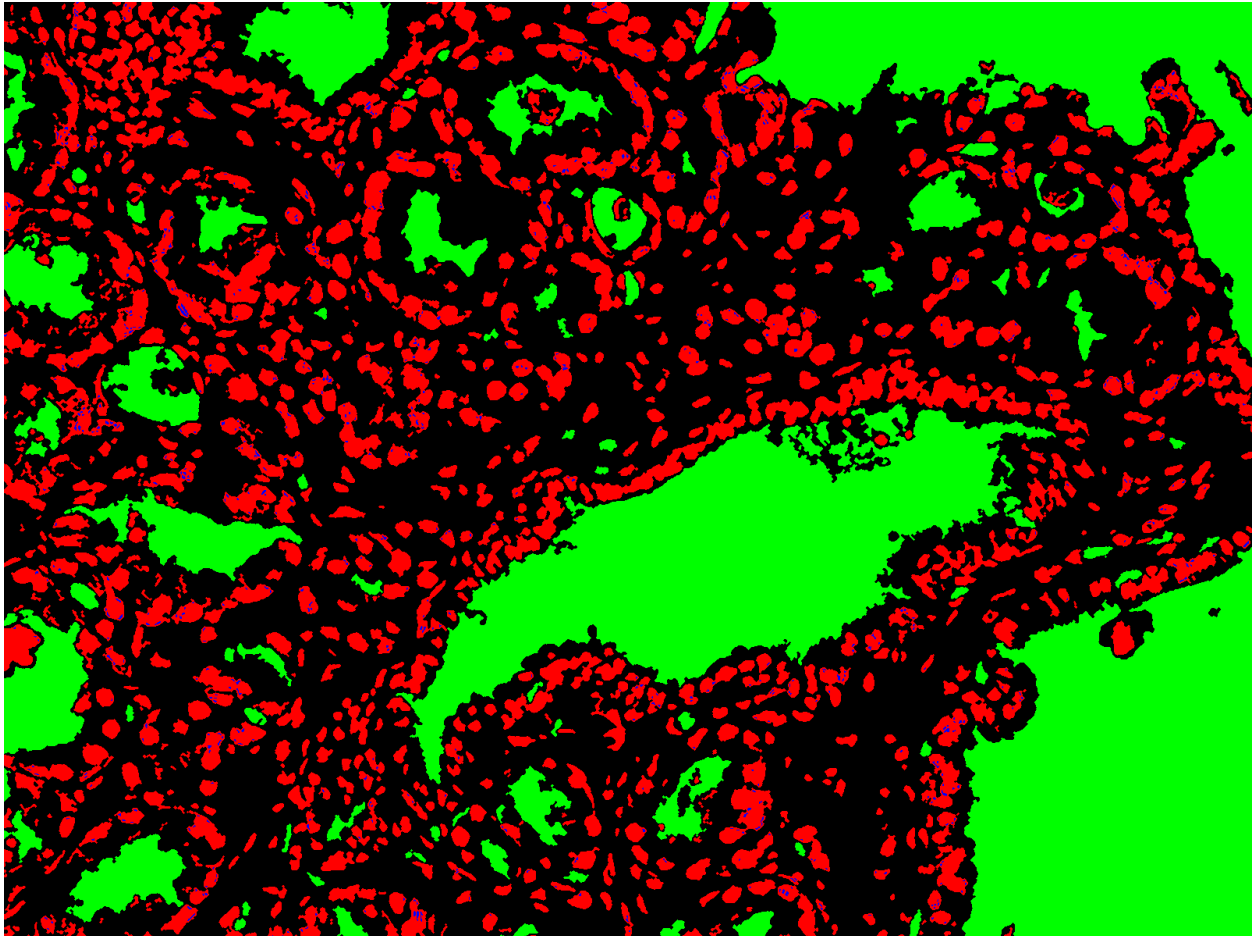


Figure 18 - Final classification image

RESULTS: ACCURACY

While there were many comparisons in accuracy between classifiers and different tweaks in classification methods—texture or no texture for classifying lumina, a new math trick in texture generation—only the most significant of those are presented here. The rest can be found in Appendix C. Figure 19 presents these in a single table, each number representing a single test, while Figure 20 summarizes the results through averaging.

Image	Structure	Normalization		Classifier			Texture	
		Normalized	Regular	SVM-Chi2	SVM-RBF	DecTree	Gray	Color
A	Nuclei			96.0432	95.3237	93.8849		
	Lumina	n/a	n/a	97.0865	97.0009	94.2588		
	Nucleoli			84.6154	80.7692	76.9231	57.6923	84.615
B	Nuclei	92.1405	75.7525	97.8896	91.1371	92.1405		
	Lumina	97.6789	95.9381	90.9091	95.3578	97.6789		
	Nucleoli	74.7664	57.9439	74.7664	73.8318	53.2710	51.4019	74.7664
C	Nuclei	88.0841	86.6822	73.5981	74.5327	88.0841		
	Lumina	99.8022	99.9011	96.8348	96.3403	99.8022		
	Nucleoli	43.5897	46.1538	43.5897	41.0256	43.5897	46.1538	43.5897
D	Nuclei	98.8959	98.7382	64.0379	69.5584	98.8959		
	Lumina	96.1390	96.9112	93.6293	94.4015	96.1390		
	Nucleoli	60.6061	57.5758	60.6061	63.6364	46.9697	53.0303	60.6061
E	Nuclei	97.0740	93.4596	66.7814	74.5267	97.0740		
	Lumina	99.8802	99.8802	97.4850	97.6048	99.8802		
	Nucleoli	78.2609	71.7391	78.2609	80.4348	39.1304	65.2174	78.2609
F	Nuclei	98.3095	66.7100	54.4863	64.8895	98.3095		
	Lumina	98.5959	44.7737	98.5959	98.4399	98.5959		
	Nucleoli	70.3125	50.0000	70.3125	65.6250	70.3125	35.9375	70.3125

Figure 19 - Accuracy results

Structure	Normalization		Classifier			Texture	
	Normalized	Regular	SVM-Chi2	SVM-RBF	DecTree	Gray	Color
Nuclei	94.9008	84.2685	75.4728	78.3280	94.7315		
Lumina	98.4192	87.4809	95.7568	96.5242	97.7258		
Nucleoli	65.5071	56.6825	68.6918	67.5538	55.0327	51.5722	68.69177

Figure 20 - Accuracy averages

The average accuracy across all categories (nuclei, lumina, and nucleoli) without normalization is 76.14%. With normalization, the average is 86.27%, about 10% higher. This is a

significant increase. Looking closely at the numbers, it can be seen in some cases that normalization did not improve accuracy by much, even lowered it in some cases (Images C, D), but in other cases, such as Image F, normalization was incredibly important. Images C and D are very similar in lighting to Image A, the training image, while Image F is significantly darker (see Appendix A). This explains the variance in how normalization affects accuracy.

The average accuracy using gray texture to find nucleoli is 51.57%. Accuracy this low is not much better than flipping a coin and marking a nucleolus present if the coin turns up heads. The average accuracy using color texture is 68.69%, a significant improvement, but still far from the ideal 90-percentile range.

Of the three classifiers primarily used throughout the project, the decision tree, SVM-Chi2, and SVM-RBF, none showed significantly greater aptitude across all areas, but on specific structures, some did better than others. The decision tree was about 20% better than both kernels of the SVM on nucleus classification. All three did well in lumina classification. The SVM kernels were about 13% better than the decision tree at finding nucleoli, but rounding out at 67-68% themselves, the accuracy in finding nucleoli is not very good. This fairly low accuracy is like a weak link in the chain: the other two steps in the cancer identification process are dependent on the first of finding the key structures. Errors as significant as 30% incorrect classification will propagate through the whole process. While it may be decreased as the accurate nucleus and lumen classification is considered, that 30% error could throw off the final output of cancer or no cancer significantly. Unless this nucleolus classification can be improved, the outlook for the three-step method using SVM and decision tree is bleak.

FUTURE WORK

There is a lot of future work left for this project. While a 10%, 13%, 20% difference in accuracy seems clearly significant, it would be best to run formal statistical significance tests to

determine that they truly are or are not. This will strengthen the conclusions of the results concerning normalization, classifiers, and texture. This is only a small issue compared to a much bigger decision that needs to be made—whether or not to stick with the FireMAP suite for the main processing power of this project. The inaccuracy of nucleoli classification will drag the total accuracy of the final diagnosis down. While there may be more methods and innovations that can improve an SVM’s ability to locate nucleoli, there seems to be a much better option that does not require programming acrobatics—the CNN. CNNs are made for object detection and computer vision. While it may be possible to get the accuracy required with an SVM, this is like digging a trench with a spoon instead of a shovel. There are simply better tools. If the FireMAP suite is abandoned in favor of the CNN, future work will involve development, training, and testing of the CNN. If development continues with the FireMAP suite, future work involves improving nucleolus classification, but also moving onto the second and third steps – analyzing key cell structures and classifying on new attributes. While the exact path to a prostate cancer screening tool may be unknown, this project was a successful step in mapping out the wildlands of using computer vision to diagnose cancer.

REFERENCES

- "Cancer Statistics." National Cancer Institute, 27 Apr 2018. <<https://www.cancer.gov/about-cancer/understanding/statistics>>. Accessed 20 Mar 2019.
- "Key Statistics for Prostate Cancer." American Cancer Society, 8 Jan 2019. <<https://www.cancer.org/cancer/prostate-cancer/about/key-statistics.html>>. Accessed 20 Mar 2019.
- "Survival Rates for Prostate Cancer." American Cancer Society, 7 Feb 2019. <<https://www.cancer.org/cancer/prostate-cancer/detection-diagnosis-staging/survival-rates.html>>. Accessed 20 Mar 2019.
- "Adenocarcinoma." Def.1. *Merriam-Webster.com Dictionary*. Merriam-Webster, 2019. <<https://www.merriam-webster.com/dictionary/adenocarcinoma>>. Accessed 20 Mar 2019.
- "Stroma." Def.2a. *Merriam-Webster.com Dictionary*. Merriam-Webster, 2019. <<https://www.merriam-webster.com/dictionary/stroma>>. Accessed 20 Mar 2019.
- Parry, Nicola. "A Beginner's Guide to Haematoxylin and Eosin Staining." BiteSize Bio, 8 Aug 2015. <<https://bitesizebio.com/13400/a-beginners-guide-to-haematoxylin-and-eosin-staining/>>. Accessed 20 Mar 2019.
- Team, Expert System. "What is Machine Learning? A Definition." Expert System, 2017. <<https://www.expertsystem.com/machine-learning-definition/>>. Accessed 20 Mar 2019.
- Soni, Devin. "Supervised vs. Unsupervised Learning." Towards Data Science, Mar 22, 2018. <<https://towardsdatascience.com/supervised-vs-unsupervised-learning-14f68e32ea8d>>. Accessed 20 Mar 2019.

Afonja, Tejumade. "Kernel Function." Towards Data Science, 2 Jan 2017.

<<https://towardsdatascience.com/kernel-function-6f1d2be6091>>. Accessed 20 Mar 2019.

Saha, Sumit. "A Comprehensive Guide to Convolutional Neural Networks - the ELI5 way."

Towards Data Science, 15 Dec 2018. <<https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>>. Accessed 20 Mar 2019.

Han, Jiawei et al. *Data Mining: Concepts and Techniques*. 3rd ed. Waltham MA, Morgan Kaufmann, 2012.

Gogul09. "Texture Recognition using Haralick Texture and Python." *GitHub*, 15 Dec 2016.

<<https://gogul09.github.io/software/texture-recognition>>. Accessed 20 Mar 2019.

Hamilton, D. et al. "Evaluation of Texture as an Input of Spatial Context for Machine Learning Mapping of Wildland Fire Effects." *Signal and Image Processing: An International Journal*, vol. 8, no. 5, 2017.

Hamilton, D. and A. Van Aardt. "Improving Mapping Accuracy of Wildland Fire Effects from Hyperspatial Imagery Using Machine Learning, Ph.D., The University of Idaho, 2018.

APPENDICES

Appendix A: Classified Images

Image A (Nuclei: 93.8849%; Lumina: 94.2588%; Nucleoli: 84.6154%)

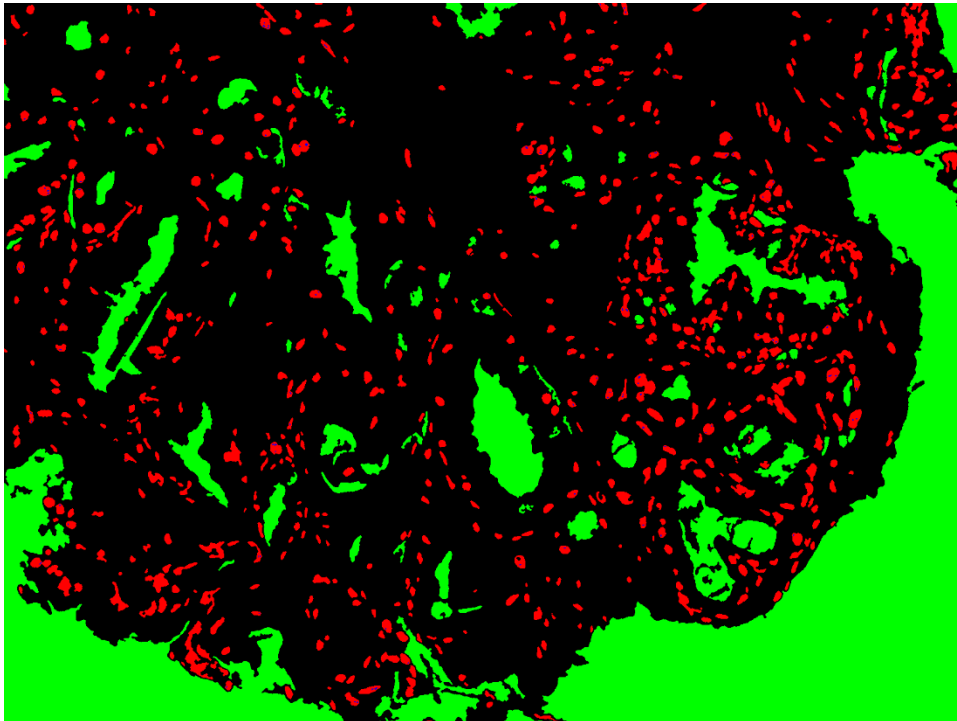
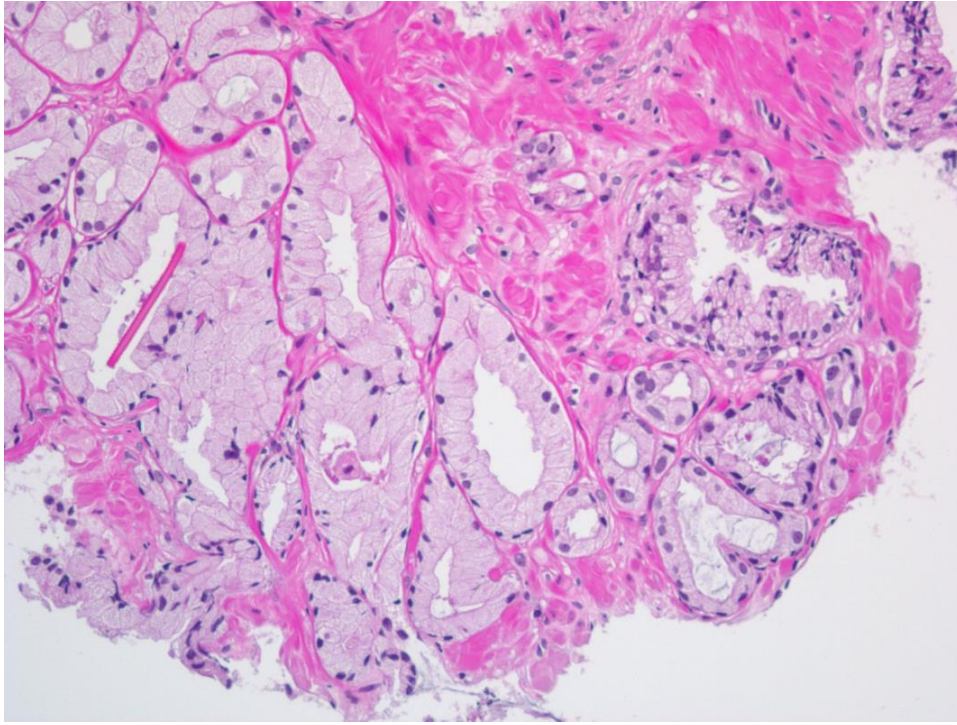


Image B (Nuclei: 92.1405%; Lumina: 97.6789%; Nucleoli: 74.7664%)

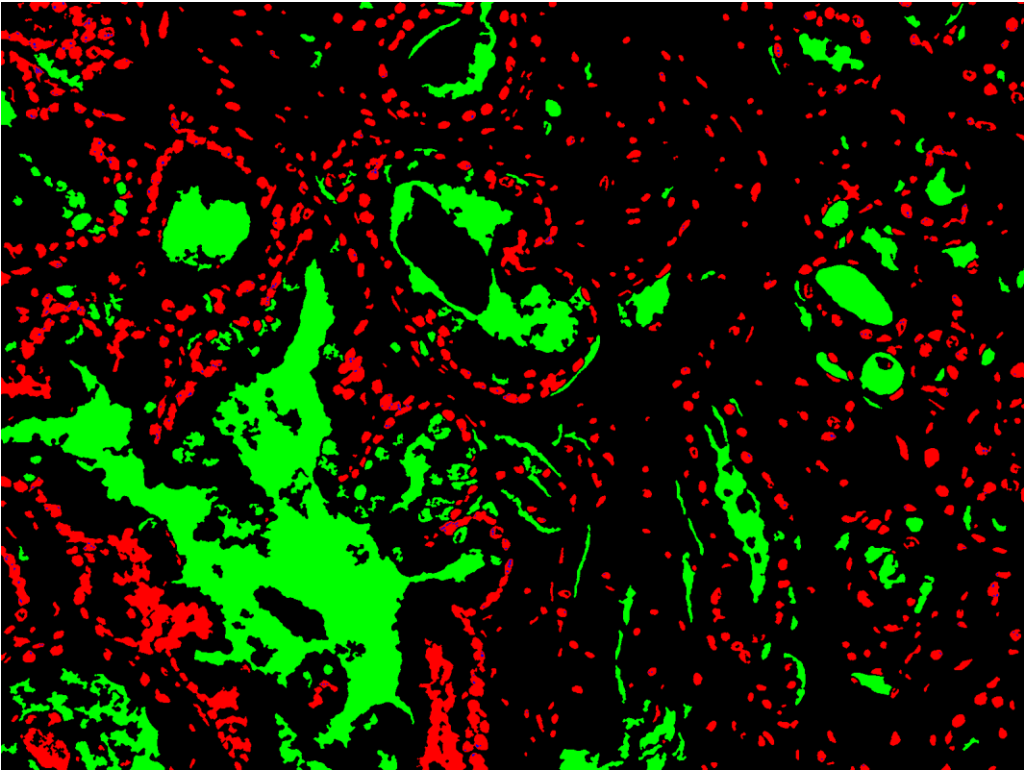
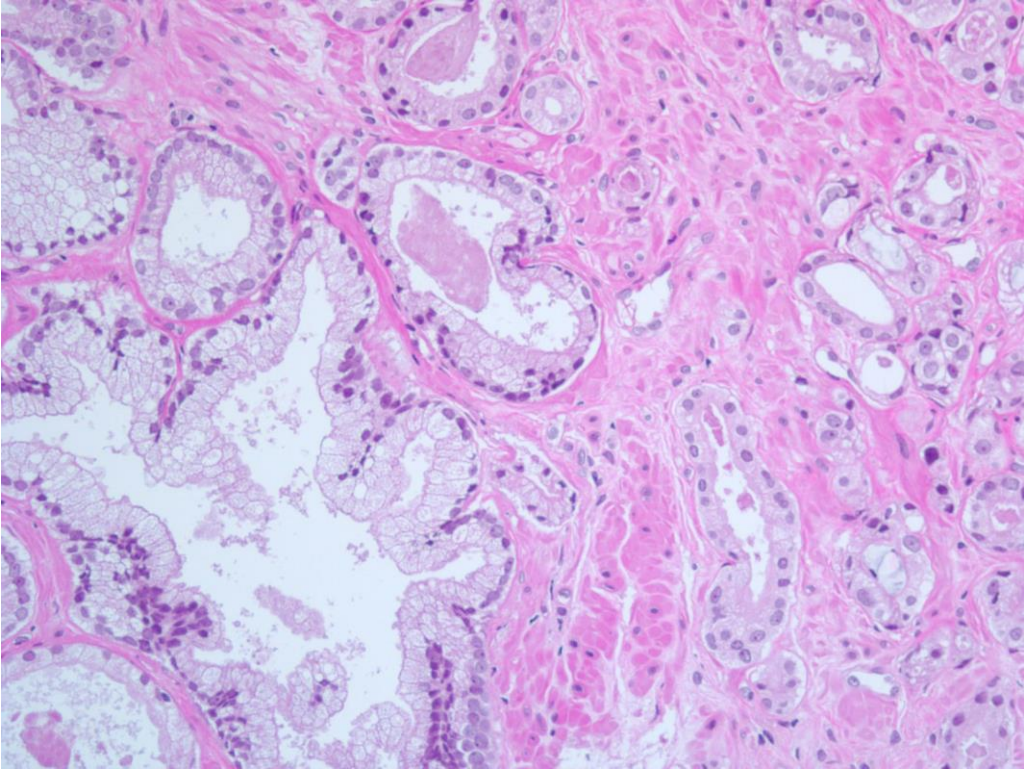


Image C (Nuclei: 88.0841%; Lumina: 99.8022%; Nucleoli: 43.5897%)

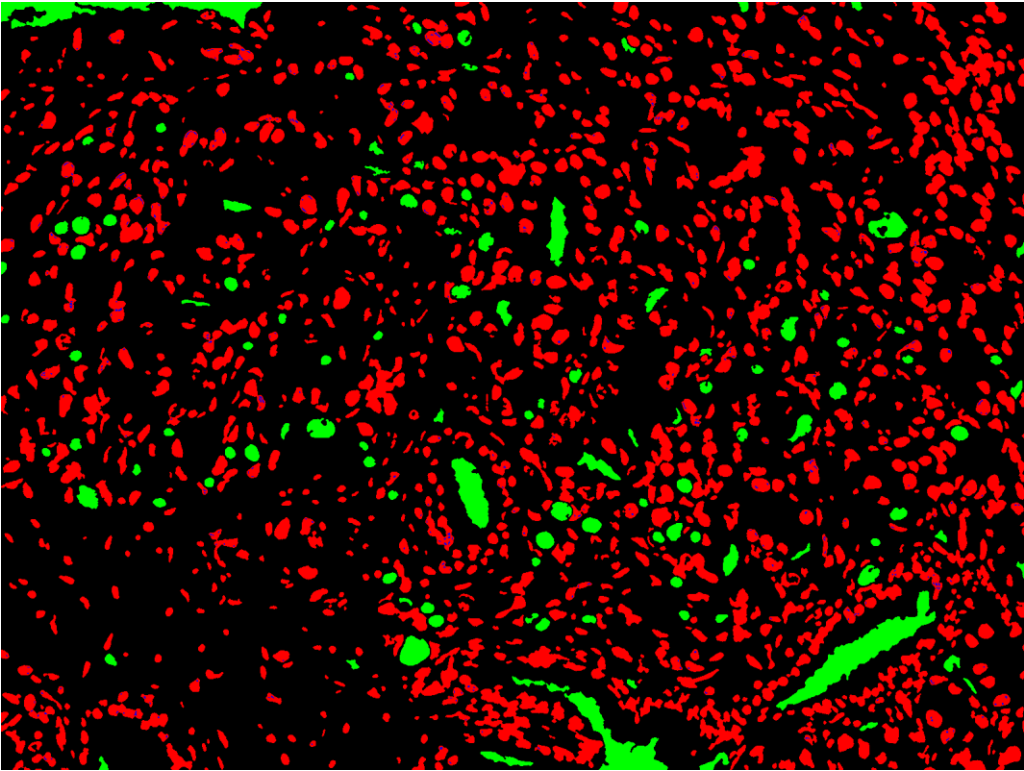
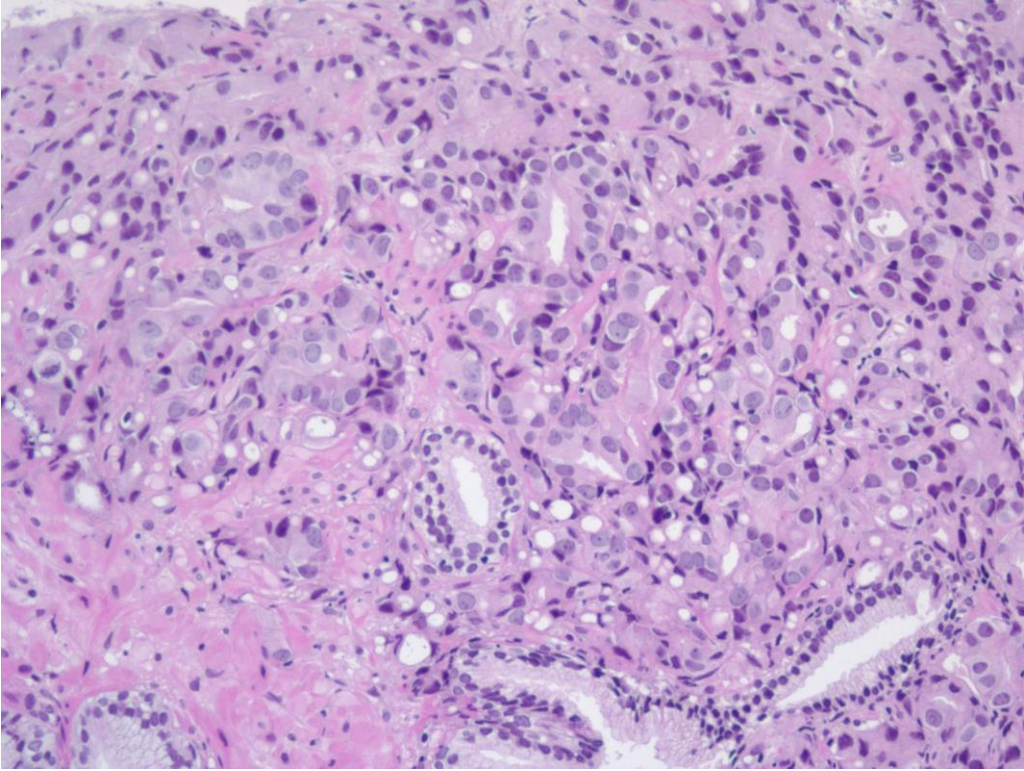


Image D (Nucleoli: 98.8959%; Lumina: 96.1390%; Nucleoli: 60.6061%)

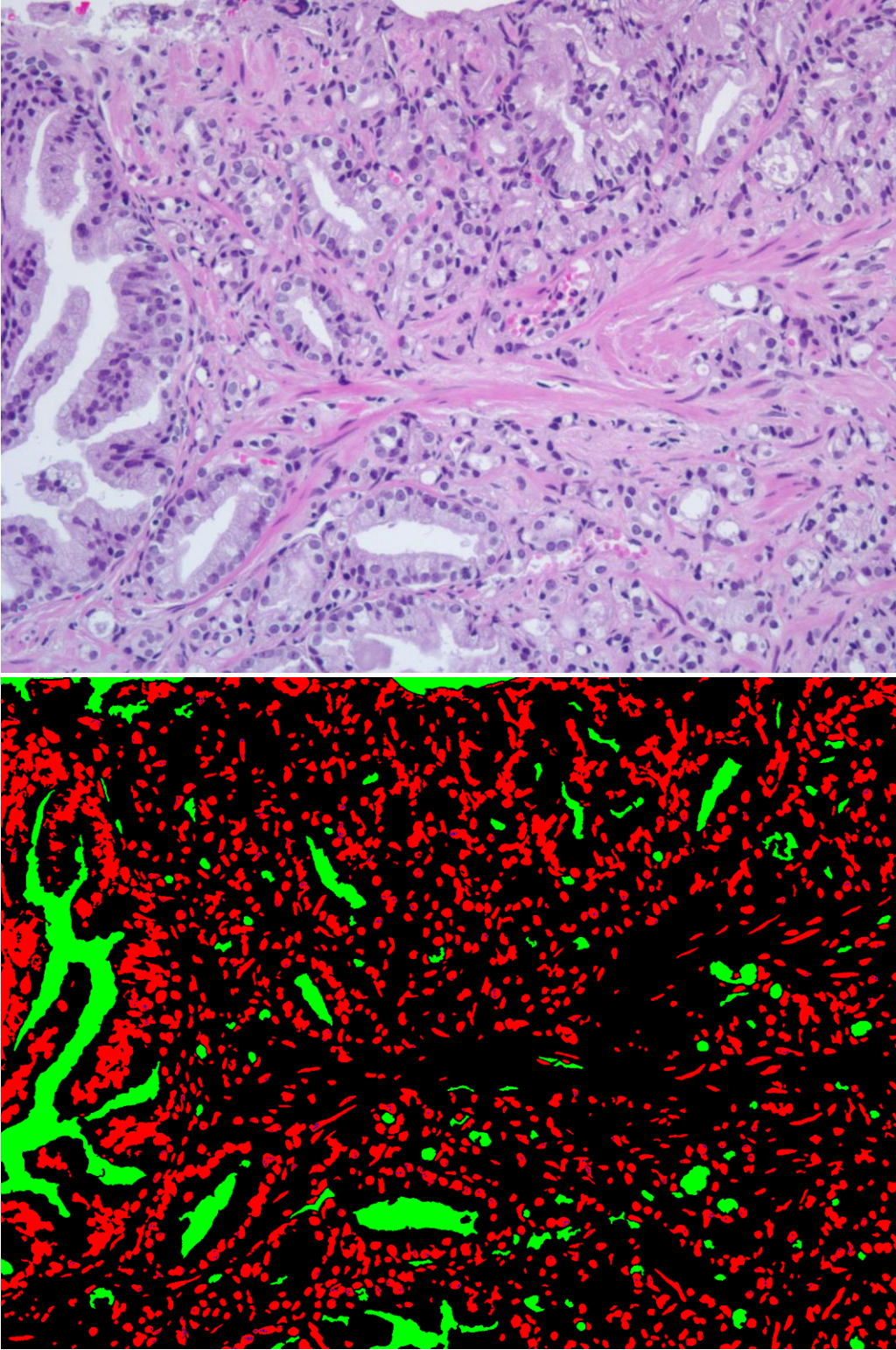


Image E (Nuclei: 97.0740%; Lumina: 99.8802%; Nucleoli: 78.2609%)

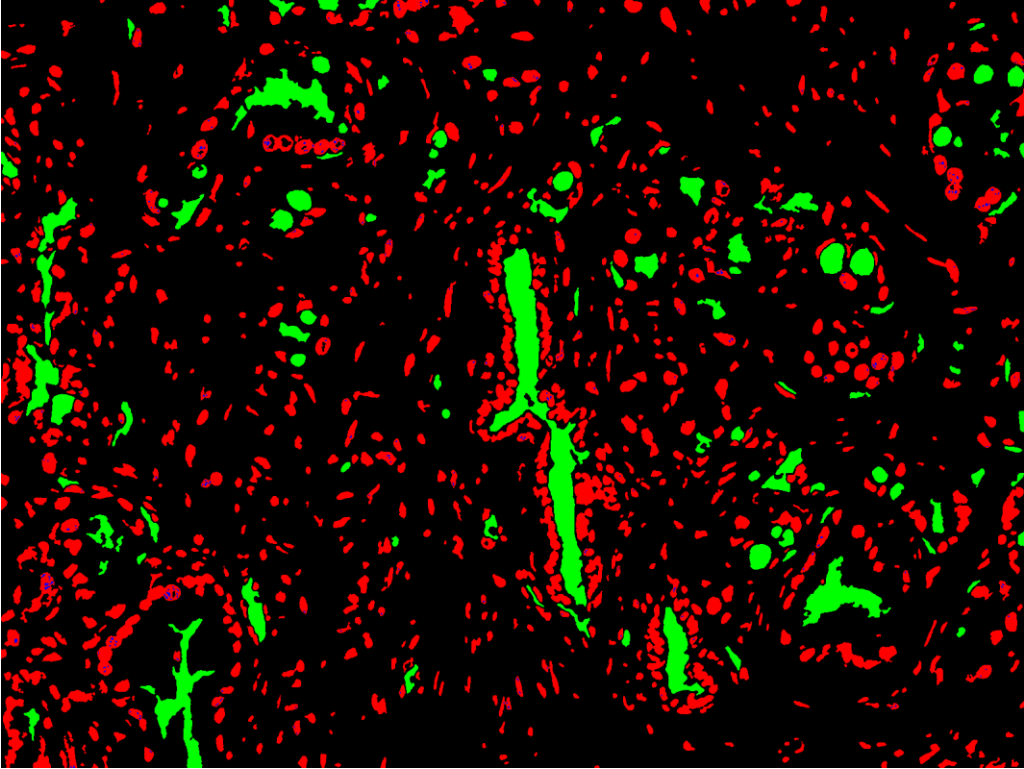
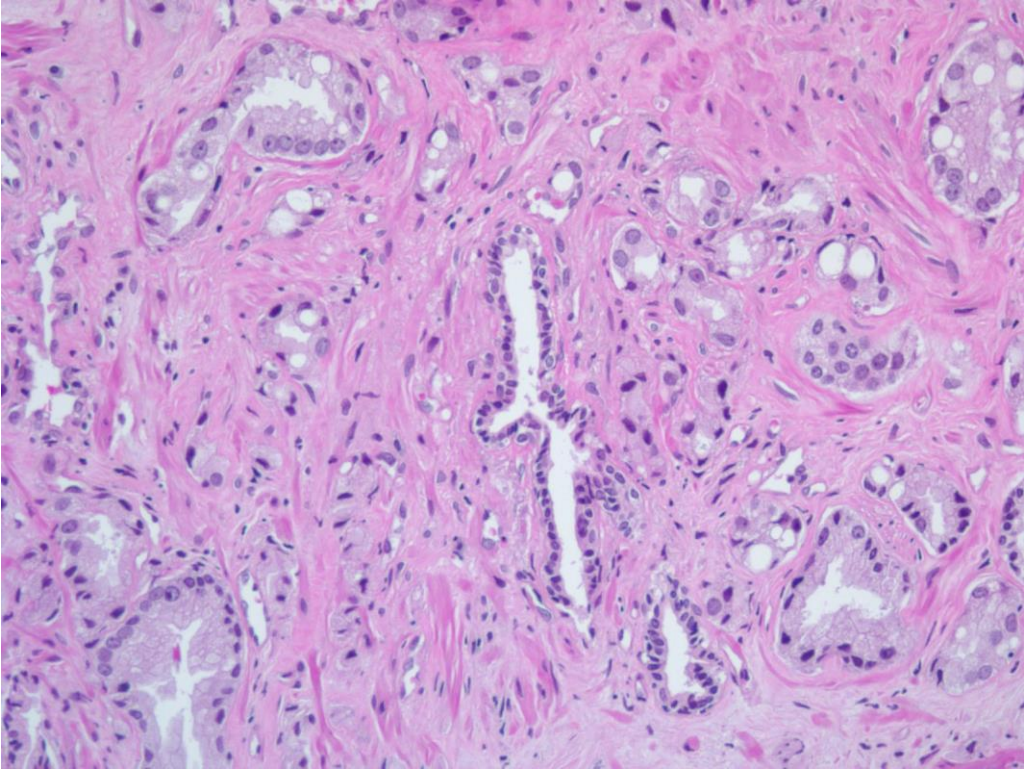
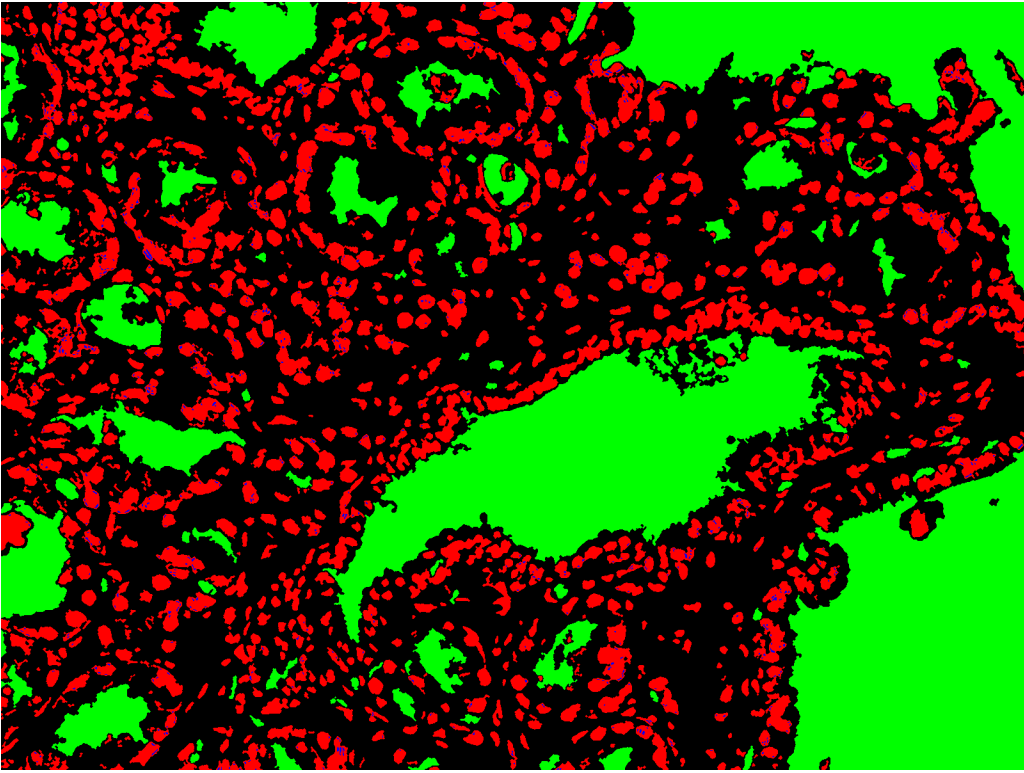
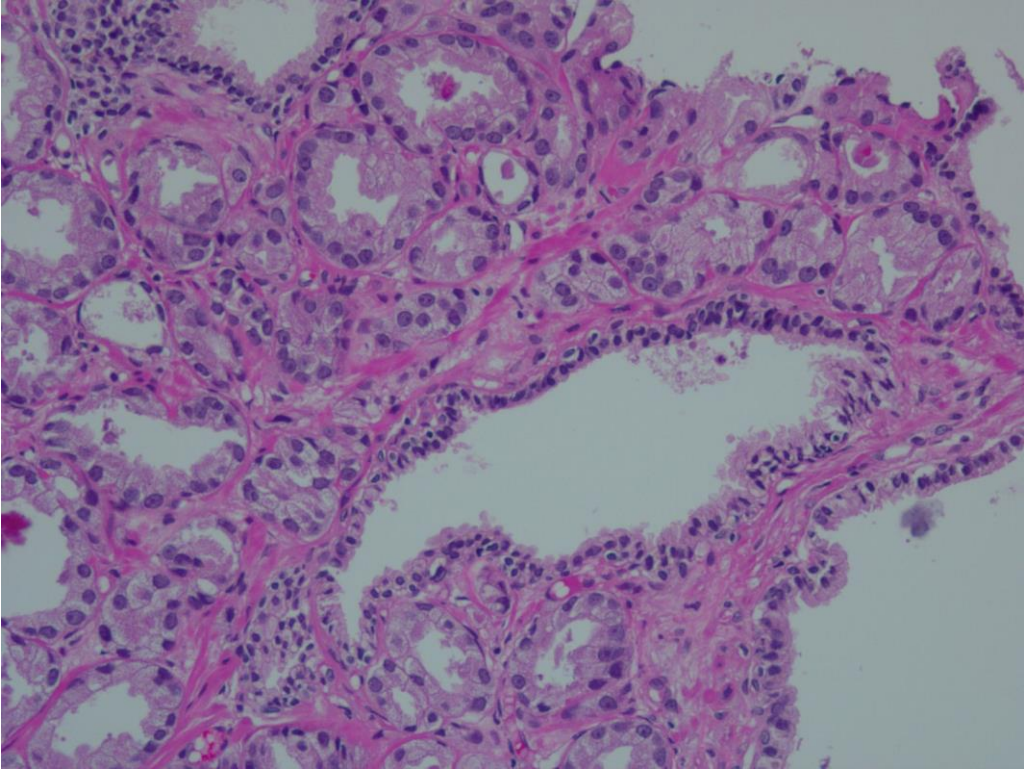


Image F (Nuclei: 98.3095%; Lumina: 98.5959%; Nucleoli: 70.3125%)



Appendix B: Code

Batch file used to run full classification from Windows Command Prompt

```
::STEP 0: DEFINE IMAGE NAMES
SET imNucleusTrain="C:\Classified\Summer\Final\Train\9.jpg"
SET imLumenTrain="C:\Classified\Summer\Final\Train\9.jpg"
SET imNuclOCTrain="C:\Classified\Summer\Final\Train\9-Texture.tif"
SET imNuclODTrain="C:\Classified\Summer\Final\Train\9-NuclOmerge.tif"
SET image=9

SET imOriginal="C:\Classified\Summer\Final\%image%.jpg"
SET imNormed="C:\Classified\Summer\Final\%image%-norm.jpg"
SET imNucleusMess="C:\Classified\Summer\Final\%image%-Nucleus.tif"
SET imNucleusClean="C:\Classified\Summer\Final\%image%-Nucleus-denoise.tif"
SET imLumenMess="C:\Classified\Summer\Final\%image%-Lumen.tif"
SET imLumenClean="C:\Classified\Summer\Final\%image%-Lumen-denoise.tif"
SET imNuclOTex="C:\Classified\Summer\Final\%image%-Texture.tif"
SET imNuclOClass="C:\Classified\Summer\Final\%image%-NuclO.tif"
SET imNuclOMerge="C:\Classified\Summer\Final\%image%-NuclOmerge.tif"
SET imNuclOClean="C:\Classified\Summer\Final\%image%-NuclOdenoise.tif"
SET imMerged="C:\Classified\Summer\Final\%image%-Classified.tif"

::STEP 1: NORMALIZE
SET normExe="C:\Users\hlmo\Documents\NNU\Summer Research\Normalization\x64\Release\
Normalization.exe"
SET inRanges="C:\Classified\Summer\Final\Train\9-ranges.txt"

%normExe% 2 %inRanges% %imOriginal% %imNormed%

::STEP 2: CLASSIFY NUCLEI & LUMINA
SET svmExe="C:\Users\hlmo\Documents\NNU\Summer Research\SvmBurnClassifier\x64\Release\
SvmBurnClassifier.exe"
SET treeExe="C:\Users\hlmo\Documents\NNU\Summer Research\ArchDecTree2\x64\Release\
ArchDecTree2.exe"
SET trainData1="C:\Classified\Summer\Final\Train\trainNucleus.csv"
SET key1="C:\Classified\Summer\Final\Train\keyNucleus.csv"
SET trainData2="C:\Classified\Summer\Final\Train\trainLumen.csv"
SET key2="C:\Classified\Summer\Final\Train\keyLumen.csv"

%treeExe% %key1% %trainData1% %imNucleusTrain% %imNormed% %imNucleusMess%
%treeExe% %key2% %trainData2% %imLumenTrain% %imNormed% %imLumenMess%
```

```
::STEP 3: DENOISE NUCLEI & LUMINA
SET denoiseExe="C:\Users\hlmox\Documents\NNU\Summer Research\Denoise\Denoise_v1.4\x64\
Release\Denoise_v1.4.exe"
```

```
%denoiseExe% %imNucleusMess% -pf 1 20 20
%denoiseExe% %imLumenMess% -pf 1 128 128
```

```
::STEP 4a: NUCLEOLI--TEXTURE
SET nucleoliExe="C:\Users\hlmox\Documents\NNU\Summer Research\ReadPixels\x64\Debug\
ReadPixels.exe"
SET offset=4
SET color=1
```

```
%nucleoliExe% %imOriginal% %imNucleusClean% %imNuclTex% %offset% %color%
```

```
::STEP 4b: NUCLEOLI--CLASSIFY
SET trainData3="C:\Classified\Summer\Final\Train\trainNuclC.csv"
SET key3="C:\Classified\Summer\Final\Train\keyNuclC.csv"
```

```
%svmExe% %imNuclCTrain% %imNuclTex% %imNuclClass% %key3% -p -chi2 -tp %trainData3%
```

```
::STEP 4c: NUCLEOLI--MERGE
SET mergeExe="C:\Users\hlmox\Documents\NNU\Summer Research\Merge\x64\Release\Merge.exe"
```

```
%mergeExe% 2 %imNucleusClean% %imNuclClass% %imNuclMerge%
```

```
::STEP 4d: NUCLEOLI--DENOISE
SET denoiseTrain="C:\Classified\Summer\Final\Train\trainNuclD.csv"
```

```
%nucleoliExe% %imNuclDTrain% %imNuclMerge% %imNuclClean% %denoiseTrain%
```

```
::STEP 5: MERGE ALL
```

```
%mergeExe% 3 %imLumenClean% %imNuclClean% %imMerged%
```

Batch file used to run accuracy calculations from Windows Command Prompt

```
SET num=9
```

```
SET image1="C:\Classified\Summer\Research\SVMvTree2\%num%-chi2.tif"
```

```
SET validNucleus="C:\Classified\Summer\Final\Valid %num%\img%num%\ValidNucleus.csv"
```

```
SET validLumen="C:\Classified\Summer\Final\Valid %num%\img%num%\ValidLumen.csv"
```

```
SET validNuclo="C:\Classified\Summer\Final\Valid %num%\img%num%\ValidNuclo.csv"
SET exe="C:\Users\hlmoj\Documents\NNU\Summer Research\CancerAccuracy2.0\x64\Debug\
CancerAccuracy2.0.exe"
```

```
%exe% %validNucleus% %validLumen% %validNuclo% %image1%
```

Normalization program Norm.h

```
#ifndef NORM_H
#define NORM_H

#include "stdafx.h"
#include "opencv2/highgui.hpp"
#include "opencv2/core.hpp"
#include <fstream>

using namespace cv;
using namespace std;

class Normalization
{
private:
    int normRanges[6];           // Ranges of RGB values to normalize to
    Mat inImage;                // Image normRanges are pulled from
    int oldRanges[6];           // Ranges of RGB values of image to be
normalized
    Mat toNormImage;            // Image to normalize
    Mat outImage;               // Normalized image

    const int RED_HIGH = 0;     // Set indices for values in ranges
arrays
    const int RED_LOW = 1;      // E.g. the lowest red value is in arr[1]
    const int GREEN_HIGH = 2;
    const int GREEN_LOW = 3;
    const int BLUE_HIGH = 4;
    const int BLUE_LOW = 5;

public:
    Normalization();            // Constructor, initializes ranges arrays
    void setInImage(string);    // Reads from given path into member variable
inImage
    void setToNormImage(string); // Reads from given path into member variables
toNormImage, creates matching but empty outImage
    void printRanges(string);   // Output normRanges to given .txt file
    void readRanges(string);    // Read from given .txt file into normRanges
    void getRanges(int[], Mat); // Fill given array with ranges pulled from the
given image
    void getNormRanges();       // Calls other member functions to read
normRanges from an image
    void normalizeImage();      // Gets ranges for toNormImage, steps
through each pixel and calculates normalized values
};
```

```

    Vec3b normalizeVector(Vec3b);           // Normalizes a given pixel
    int normalizeValue(int, int);         // Normalizes a given pixel value
    void saveImage(string);               // Saves outImage to the given string
};

#endif

```

Norm.cpp

```

#include "stdafx.h"
// #include "opencv2/highgui.hpp" in Norm.h
// #include "opencv2/core.hpp" in Norm.h
// #include <fstream> in Norm.h

#include "Norm.h"

Normalization::Normalization()
{
    // Initialize both ranges arrays so they aren't empty
    normRanges[RED_HIGH] = 0;
    normRanges[GREEN_HIGH] = 0;
    normRanges[BLUE_HIGH] = 0;
    normRanges[RED_LOW] = 255;
    normRanges[GREEN_LOW] = 255;
    normRanges[BLUE_LOW] = 255;

    oldRanges[RED_HIGH] = 0;
    oldRanges[GREEN_HIGH] = 0;
    oldRanges[BLUE_HIGH] = 0;
    oldRanges[RED_LOW] = 255;
    oldRanges[GREEN_LOW] = 255;
    oldRanges[BLUE_LOW] = 255;
}

void Normalization::setInImage(string input)
{
    inImage = imread(input);
}

void Normalization::setToNormImage(string input)
{
    toNormImage = imread(input);
    outImage.create(toNormImage.rows, toNormImage.cols, CV_8UC3);
}

void Normalization::printRanges(string output)
{
    ofstream outFile;
    outFile.open(output.c_str());

    outFile << "Red range: ( " << normRanges[RED_LOW] << " " << normRanges[RED_HIGH]
    << " )" << endl;
    outFile << "Green range: ( " << normRanges[GREEN_LOW] << " " <<
    normRanges[GREEN_HIGH] << " )" << endl;
}

```

```

        outFile << "Blue range: ( " << normRanges[BLUE_LOW] << " " <<
normRanges[BLUE_HIGH] << " )" << endl;

        outFile.close();
    }

void Normalization::readRanges(string input)
{
    ifstream inFile;
    string line;
    string low;
    string high;
    inFile.open(input.c_str());

    for (int i = 0; i < 3; i++)
    {
        getline(inFile, line, '(');
        inFile >> low;
        normRanges[1 + 2 * i] = stoi(low);
        inFile >> high;
        normRanges[2 * i] = stoi(high);
        getline(inFile, line);
    }
}

void Normalization::getNormRanges()
{
    getRanges(normRanges, inImage);
}

void Normalization::getRanges(int ranges[], Mat image)
{
    Vec3b pixel;

    for (int i = 0; i < image.rows; i++)
    {
        for (int j = 0; j < image.cols; j++)
        {
            pixel = image.at<Vec3b>(i, j);

            //check for new high
            if (pixel[2] > ranges[RED_HIGH])
            {
                ranges[RED_HIGH] = pixel[2];
            }
            if (pixel[1] > ranges[GREEN_HIGH])
            {
                ranges[GREEN_HIGH] = pixel[1];
            }
            if (pixel[0] > ranges[BLUE_HIGH])
            {
                ranges[BLUE_HIGH] = pixel[0];
            }

            //check for new low
            if (pixel[2] < ranges[RED_LOW])
            {
                ranges[RED_LOW] = pixel[2];
            }
        }
    }
}

```

```

        }
        if (pixel[1] < ranges[GREEN_LOW])
        {
            ranges[GREEN_LOW] = pixel[1];
        }
        if (pixel[0] < ranges[BLUE_LOW])
        {
            ranges[BLUE_LOW] = pixel[0];
        }
    }
}

void Normalization::normalizeImage()
{
    getRanges(oldRanges, toNormImage);

    Vec3b pixel;

    for (int i = 0; i < toNormImage.rows; i++)
    {
        for (int j = 0; j < toNormImage.cols; j++)
        {
            pixel = toNormImage.at<Vec3b>(i, j);
            pixel = normalizeVector(pixel);
            outImage.at<Vec3b>(i, j) = pixel;
        }
    }
}

Vec3b Normalization::normalizeVector(Vec3b oldPixel)
{
    Vec3b newPixel;
    newPixel[2] = normalizeValue(oldPixel[2], 0); //0 for red
    newPixel[1] = normalizeValue(oldPixel[1], 1); //1 for green
    newPixel[0] = normalizeValue(oldPixel[0], 2); //2 for blue
    return newPixel;
}

int Normalization::normalizeValue(int oldVal, int RGB)
{
    // RGB = 0 for red, 1 for green, 2 for blue
    // Using min-max normalization

    double numerator = double(oldVal - oldRanges[2 * RGB + 1]);
    double denominator = double(oldRanges[2 * RGB] - oldRanges[2 * RGB + 1]);
    double newRange = double(normRanges[2 * RGB] - normRanges[2 * RGB + 1]);
    double newOffset = double(normRanges[2 * RGB + 1]);

    double newVal = (numerator / denominator)*newRange + newOffset;
    int roundVal = round(newVal);

    return roundVal;
}

void Normalization::saveImage(string output)
{
    imwrite(output, outImage);
}

```

```
}
```

Normalization.cpp

```
// Normalization.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "opencv2/highgui.hpp"
#include "opencv2/core.hpp"
#include <iostream>
#include <fstream>
#include <string>

#include "Norm.h"

using namespace cv;
using namespace std;

// Prototypes
bool checkFile(string);

// Driver for Norm.cpp
int main(int argc, char* argv[])
{
    if (argc == 1)
    {
        cout << "First argument must be option number (1, 2, or 3)" << endl <<
endl;

        cout << "Option 1: given an image, output to a text file the range of
colors in that image" << endl;
        cout << "Cmdline params: 1 input_image (output_file_path)" << endl << endl;

        cout << "Option 2: given an image and color range, output image normalized
to that range" << endl;
        cout << "Cmdline params: 2 input_color_range to_norm_image
(output_image_path)" << endl << endl;

        cout << "Option 3: given two images, output the second normalized to the
first" << endl;
        cout << "Cmdline params: 3 input_image to_norm_image (output_image_path)"
<< endl << endl;
    }
    else // (argc > 1)
    {
        int option = 0;

        try
        {
            option = stoi(argv[1]);
        }
        catch (exception e)
        {
            cout << "First argument must be option number (1, 2, or 3)" << endl;
        }
    }
}
```



```

    if (option == 1 && argc >= 3)
    {
        // Set up and check args
        string inputImagePath = argv[2];
        string outputFilePath;
        if (argc == 3)
        {
            outputFilePath = inputImagePath.substr(0,
inputImagePath.find_last_of('.') + "-ranges.txt");
        }
        else //argc == 4
        {
            outputFilePath = argv[3];
        }

        if (checkFile(inputImagePath))
        {
            // Run Option 1: get range for normalization from
an image
            cout << "Running Option 1... ";
            Normalization norm;
            //initialize norm function object
            norm.setInImage(inputImagePath); //initialize
input image to get ranges from
            norm.getNormRanges();
            //find range values
            norm.printRanges(outputFilePath); //save ranges
to filepath
            cout << "done" << endl;
        }
    }
    else if (option == 2 && argc >= 4)
    {
        // Set up and check args
        string inputRangesPath = argv[2];
        string toNormImagePath = argv[3];
        string outputImagePath;
        if (argc == 4)
        {
            outputImagePath = toNormImagePath.substr(0,
toNormImagePath.find_last_of('.') + "-normalized.tif");
        }
        else //argc == 5
        {
            outputImagePath = argv[4];
        }

        if (checkFile(inputRangesPath) && checkFile(toNormImagePath))
        {
            // Run Option 2: normalize an image to the given
range
            cout << "Running Option 2... ";
            Normalization norm;
            //initialize norm function object
            norm.setToNormImage(toNormImagePath);
            //initialize image to normalize

```

```

                                norm.readRanges(inputRangesPath);           //read
in new ranges from file
                                norm.normalizeImage();
                                //normalize image to new ranges
                                norm.saveImage(outputImagePath);          //save
normalized image
                                cout << "done" << endl;
                                }
                                }
else if (option == 3 && argc >= 4)
{
    // Set up and check args
    string inputImagePath = argv[2];
    string toNormImagePath = argv[3];
    string outputImagePath;
    if (argc == 4)
        outputImagePath = toNormImagePath.substr(0,
toNormImagePath.find_last_of('.') + "-normalized.tif");
    else //argc == 5
        outputImagePath = argv[4];

    if (checkFile(inputImagePath) && checkFile(toNormImagePath))
    {
        // Run option 3: normalize one image to another
        cout << "Running Option 3... ";
        Normalization norm;

        //initialize norm function object
        norm.setInputImage(inputImagePath);
        //initialize input image to get ranges from
        norm.setToNormImage(toNormImagePath);
        //initizlie image to normalize
        norm.getNormRanges();
        //get new ranges to normalize to
        norm.normalizeImage();
        //normalize image to new ranges
        norm.saveImage(outputImagePath);          //save
normalized image
        cout << "done" << endl;
    }
}
else //incorrect arguments given
{
    cout << "First argument must be option number (1, 2, or 3)" << endl
<< endl;

    cout << "Option 1: given an image, output to a text file the range
of colors in that image" << endl;
    cout << "Cmdline params: 1 input_image (output_file_path)" << endl
<< endl;

    cout << "Option 2: given an image and color range, output image
normalized to that range" << endl;
    cout << "Cmdline params: 2 input_color_range to_norm_image
(output_image_path)" << endl << endl;

    cout << "Option 3: given two images, output the second normalized to
the first" << endl;

```

```

        cout << "Cmdline params: 3 input_image to_norm_image
(output_image_path)" << endl << endl;
    }
}

return 0;
}

bool checkFile(string fileName)
{
    bool good = false;

    //considered good if file opens
    fstream infile;
    infile.open(fileName.c_str());
    if (infile)
        good = true;
    else
        cout << "File <" << fileName << "> cannot be opened" << endl;
    infile.close();

    return good;
}

```

Texture generation program GradientImage.h

```

#ifndef GRIMG_H
#define GRIMG_H

#include "stdafx.h"
#include "opencv2/highgui.hpp"
#include "opencv2/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <string>
#include <fstream>
#include <iostream>

using namespace std;
using namespace cv;

class PixelGrad {
public:
    int grads[4][4];
    float gradsAvg[4];

    /* 2D GRADIENT ARRAY LAYOUT */
    //          up down left right
    // blue
    // green
    // red
    // total
};

/*

```

```

Adjustments I'd like to make:
- (X) store imOut as a member variable, so it persists the functions that create
it
- store dimensions of the image as member variables, so I'm not always accessing
an image just to get that value
*/

class Image {
private:
    const int SIDE_FLAG = -1234;

    double threshHigh;
    double threshLow;
    int ROWS;
    int COLS;

    Mat imRGB; //original raw, RGB image
    Mat imClass; //nucleus positive, other negative
    Mat imClassShave; //edges removed
    Mat imOutClass; //double positive and other
    Mat imOutTex; //avg gradient output
    Mat imGray; //human-eye weighted grayscale
    Mat imGrayBinned; //binned grayscale :D
    PixelGrad** gradient; //2D array matching imRGB's dimensions

    int calcTotalGradient(Vec3b, Vec3b);
    void calcColorGradient(PixelGrad&, Vec3b, Vec3b, int);
    void setTotalGradient(int[], int, int);
    void setColorGradient(PixelGrad, int, int);
    void getAvgGrads(int, int);
    int gradGreaterThen(int, int, int);
    int closeToEdge(int, int, int);
    int getSize(int, int, Mat&);
    void deleteChunk(int, int);
    void getCounts(int, int, int&, int&, Mat&);
    void deleteRed(int, int, Mat&);

public:
    // Basic member functions
    Image();
    void setImage(string, string);
    Image(string, string);
    ~Image();
    double getHigh();
    double getLow();

    // Output Functions
    void printGradient(); // simple outputs as tuples
    void printPrettyGradient(string); // outputs into csv for Excel
    void saveGradTotalTexture(string); // currently normalizes, might be useless
    void saveGradColorTexture(string); // color version of gradients :D
    void saveClassified(string); // self-explanatory
    void saveConTexture(string); // save outImTex, basically

    // Calculate Gradients
    void getTotalGrads(int); // Get general gradients based on
given offset

```

```

    void getColorGrads(int); // Get general gradients in red,
green, and blue
    void getTotalAvgGrads(int); // Compute avg gradients, assuming
general
    void getColorAvgGrads(int); // compute avg grad for RGB
channels

    // Training Data
    void getSplits(string); // gets splits for
classification
    float getPercentage(string, string); // gets percent for denoise2
    float infoGain(int[], float[], int); // finds greatest entropy
    float getEntropy(int[][2]); // computes entropy
    float neg_xlogx(float); // -x*log(x), 0 at 0;

    // Classification
    void classify(int, int); // Look at general gradients, 3/4
over threshold
    void classifyv2(int, int, int); // 3/4 general grads over
threshold and not close to edge
    //classify3 // nucleusThresh,
avg gradient thresh (like 1)
    void classifyv4(double, double); // gradAvg between two values

    // Pre- and Post-Processing
    void removeEdges(int); // shaves off positive
shapes by given offset
    void removeRedEdges(int); // shaves off double positive
shapes by given offset
    void denoise1(int, int); // because i'm stubborn (Classic
denoise)
    void denoise2(double); // also because I'm
stubborn (percentage denoise)
    void accuracyEval(string); // does stuff to compute accuracy
and stuff.

    // Contrast
    void contrast(int, int, int); // main for getting GLCM-based
contrast
    void bin(int); // binning!
    void getGLCM(Mat, int, int, int, int); // computes GLCM
};

#endif

#include "stdafx.h"

#include "GradientImage.h"

/* BASIC MEMBER FUNCTIONS */

Image::Image()
{
    threshHigh = 500;
    threshLow = 0;
}

```

```

void Image::setImage(string imageOriginalPath, string imageClassifiedPath)
{
    imRGB = imread(imageOriginalPath, 1);
    imClass = imread(imageClassifiedPath, 0);
    //both input images should have the same dimensions
    ROWS = imRGB.rows;
    COLS = imRGB.cols;

    gradient = new PixelGrad*[ROWS];
    for (int i = 0; i < ROWS; i++)
    {
        gradient[i] = new PixelGrad[COLS];
    }
}

Image::Image(string imageOriginalPath, string imageClassifiedPath)
{
    threshHigh = 500;
    threshLow = 0;

    imRGB = imread(imageOriginalPath, 1);
    imClass = imread(imageClassifiedPath, 0);
    //both input images should have the same dimensions
    ROWS = imRGB.rows;
    COLS = imRGB.cols;

    gradient = new PixelGrad*[ROWS];
    for (int i = 0; i < ROWS; i++)
    {
        gradient[i] = new PixelGrad[COLS];
    }
}

Image::~Image()
{
    for (int i = 0; i < ROWS; i++)
    {
        delete[] gradient[i];
    }
    delete[] gradient;
}

double Image::getHigh()
{
    return threshHigh;
}

double Image::getLow()
{
    return threshLow;
}

/* OUTPUT MEMBER FUNCTIONS */

void Image::printGradient()
{
    //updated but haven't debugged
}

```

```

PixelGrad* ptr;
for (int i = 0; i < ROWS; i++)
{
    for (int j = 0; j < COLS; j++)
    {
        ptr = &gradient[i][j];
        cout << "(" << ptr->grads[3][0] << " " << ptr->grads[3][1] << " " <<
ptr->grads[3][2]
                << " " << ptr->grads[3][3] << " " << ptr->gradsAvg[3] << ") ";
    }
    cout << endl;
}

void Image::printPrettyGradient(string outFilePath)
{
    //Fairly meaningless output if offset != 1
    //Upgraded but haven't debugged

    PixelGrad* ptr;
    ofstream outFile;
    outFile.open(outFilePath.c_str());

    //    For each row of pixels
    for (int i = 0; i < ROWS; i++)
    {
        //    Output top row
        outFile << ",";
        for (int j = 0; j < COLS; j++)
        {
            ptr = &gradient[i][j];
            outFile << ptr->grads[3][0] << ",";
        }
        //    Output side (right) row and avg
        outFile << endl << "-1234,";
        for (int j = 0; j < COLS; j++)
        {
            ptr = &gradient[i][j];
            outFile << ptr->gradsAvg[3] << "," << ptr->grads[3][2] << ",";
        }
        outFile << endl;
    }
    // Output bottom row
    for (int j = 0; j < COLS; j++)
    {
        outFile << ",-1234,";
    }
}

void Image::saveGradTotalTexture(string outImagePath)
{
    int classPixel;
    imOutTex.create(ROWS, COLS, CV_8UC1);

    float highest = 0;
    float lowest = 255;
    float avg;
}

```

```

for (int i = 0; i < ROWS; i++)
{
    for (int j = 0; j < COLS; j++)
    {
        classPixel = imClassShave.at<uchar>(i, j) % 2;
        if (classPixel)
        {
            avg = gradient[i][j].gradsAvg[3];
            if (avg > highest)
                highest = avg;
            if (avg < lowest)
                lowest = avg;
        }
    }
}

for (int i = 0; i < ROWS; i++)
{
    for (int j = 0; j < COLS; j++)
    {
        classPixel = imClassShave.at<uchar>(i, j) % 2;
        avg = (gradient[i][j].gradsAvg[3] - lowest) / (highest - lowest) *
255;
        if (classPixel)
        {
            imOutTex.at<uchar>(i, j) = int(round(avg));
        }
        else
        {
            imOutTex.at<uchar>(i, j) = 0;
        }
    }
}

imwrite(outImagePath, imOutTex);
}

void Image::saveGradColorTexture(string outImagePath)
{
    int classPixel;
    Vec3b texPixel;
    Vec3b black = { 0, 0, 0 };
    imOutTex.create(ROWS, COLS, CV_8UC3);

    //no normalization, not sure it's worth it, we'll see how it looks

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            classPixel = imClassShave.at<uchar>(i, j) % 2;
            //avg = (gradient[i][j].gradsAvg[3] - lowest) / (highest - lowest) *
255;
            if (classPixel)
            {
                for (int k = 0; k < 3; k++)
                {

```



```

        texPixel[k] = round(gradient[i][j].gradsAvg[k]);
    }
    imOutTex.at<Vec3b>(i, j) = texPixel;
}
else
{
    imOutTex.at<Vec3b>(i, j) = black;
}
}
}

imwrite(outImagePath, imOutTex);
}

void Image::saveClassified(string outImagePath)
{
    imwrite(outImagePath, imOutClass);
}

void Image::saveConTexture(string outImagePath)
{
    imwrite(outImagePath, imOutTex);
}

/* GRADIENT COMPUTATION MEMBER FUNCTIONS */

void Image::getTotalGrads(int offset)
{
    Vec3b centerPixel;
    Vec3b sidePixel;
    int gradArr[4];

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            centerPixel = imRGB.at<Vec3b>(i, j);
            for (int k = 0; k < 4; k++)
            {
                gradArr[k] = SIDE_FLAG;
            }
            if (i > offset - 1) //room above
            {
                //get upper gradient
                sidePixel = imRGB.at<Vec3b>(i - offset, j);
                gradArr[0] = calcTotalGradient(centerPixel, sidePixel);
            }
            if (i < (imRGB.rows - offset)) //room below
            {
                //get lower gradient
                sidePixel = imRGB.at<Vec3b>(i + offset, j);
                gradArr[1] = calcTotalGradient(centerPixel, sidePixel);
            }
            if (j > offset - 1) //room on left
            {
                //get left gradient
                sidePixel = imRGB.at<Vec3b>(i, j - offset);
                gradArr[2] = calcTotalGradient(centerPixel, sidePixel);
            }
        }
    }
}

```

```

    }
    if (j < (imRGB.cols - offset)) //room on right
    {
        //get right gradient
        sidePixel = imRGB.at<Vec3b>(i, j + offset);
        gradArr[3] = calcTotalGradient(centerPixel, sidePixel);
    }
    setTotalGradient(gradArr, i, j);
}
}

void Image::getColorGrads(int offset)
{
    Vec3b centerPixel;
    Vec3b sidePixel;

    PixelGrad rainbow;

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            centerPixel = imRGB.at<Vec3b>(i, j);
            for (int k = 0; k < 3; k++)
            {
                for (int l = 0; l < 4; l++)
                {
                    rainbow.grads[k][l] = SIDE_FLAG;
                }
            }

            if (i > offset - 1) //room above
            {
                //get upper gradient
                sidePixel = imRGB.at<Vec3b>(i - offset, j);
                calcColorGradient(rainbow, centerPixel, sidePixel, 0);
            }
            if (i < (imRGB.rows - offset)) //room below
            {
                //get lower gradient
                sidePixel = imRGB.at<Vec3b>(i + offset, j);
                calcColorGradient(rainbow, centerPixel, sidePixel, 1);
            }
            if (j > offset - 1) //room on left
            {
                //get left gradient
                sidePixel = imRGB.at<Vec3b>(i, j - offset);
                calcColorGradient(rainbow, centerPixel, sidePixel, 2);
            }
            if (j < (imRGB.cols - offset)) //room on right
            {
                //get right gradient
                sidePixel = imRGB.at<Vec3b>(i, j + offset);
                calcColorGradient(rainbow, centerPixel, sidePixel, 3);
            }
            setColorGradient(rainbow, i, j);
        }
    }
}

```

```

    }
}

void Image::getTotalAvgGrads(int offset)
{
    getAvgGrads(offset, 3);
}

void Image::getColorAvgGrads(int offset)
{
    getAvgGrads(offset, 0);
    getAvgGrads(offset, 1);
    getAvgGrads(offset, 2);
}

/* TRAININD DATA MEMBER FUNCTIONS */

void Image::getSplits(string filePath)
{
    string line;
    int x;
    int y;
    int texVal;
    int texture[1000]; // don't feel like counting and doing dynamic array :P
    int arrLen = 0;
    int outliers = 0;
    double mean = 0;
    double sd = 0;

    ifstream inFile;
    inFile.open(filePath.c_str());

    getline(inFile, line); //junk header
    while (!inFile.eof())
    {
        // Get values
        getline(inFile, line, ',');
        getline(inFile, line, ',');
        x = stoi(line);
        getline(inFile, line, '\n');
        y = stoi(line);
        texVal = round(gradient[y][x].gradsAvg[3]);
        texture[arrLen++] = texVal;
    }
    inFile.close();

    // Get mean
    for (int i = 0; i < arrLen; i++)
    {
        mean += texture[i];
    }
    mean = mean / arrLen;

    // Get standard deviation
    for (int i = 0; i < arrLen; i++)
    {
        sd += pow(texture[i] - mean, 2);
    }
}

```

```

    }
    sd = sd / arrLen;
    sd = sqrt(sd);

    // Throw out "outliers"
    for (int i = 0; i < arrLen; i++)
    {
        if (texture[i] < (mean - sd) || texture[i] > (mean + sd))
        {
            texture[i] = 0;
            outliers++;
        }
    }

    // Recalculate mean
    mean = sd = 0;
    for (int i = 0; i < arrLen; i++)
    {
        mean += texture[i];
    }
    mean = mean / (arrLen - outliers);

    // Recalculate standard deviation
    for (int i = 0; i < arrLen; i++)
    {
        if (texture[i] != 0) //could technically have a zero value? outlier though?
        {
            sd += pow(texture[i] - mean, 2);
        }
    }
    sd = sd / (arrLen - outliers);
    sd = sqrt(sd);

    // Set thresholds
    threshLow = mean - sd;
    threshHigh = 2 * mean;
}

float Image::getPercentage(string filePath, string imagePath)
{
    // for just denoise
    if (imOutClass.dims == 0)
        imOutClass = imRGB.clone();

    //swap in training image
    Mat imHold = imOutClass.clone();
    imOutClass = imread(imagePath, 1);

    //lots of variables
    string line;
    int x;
    int y;
    float percents[1000]; //parallel array for nuclei
    int classes[1000]; //other parallel array for nuclei
    int arrLen = 0;
    Mat checked;
    checked.create(ROWS, COLS, CV_8UC1);
    int redCount;

```

```

int whiteCount;

ifstream inFile;
inFile.open(filePath.c_str());

getline(inFile, line);    //junk header
while (!inFile.eof())
{
    // Get values: fill class and percent arrays
    getline(inFile, line, ',');
    if (line[0] == 'N') //shortcut check
        classes[arrLen] = 1;
    else
        classes[arrLen] = 0;
    getline(inFile, line, ',');
    x = stoi(line);
    getline(inFile, line, '\n');
    y = stoi(line);
    redCount = whiteCount = 0;
    getCounts(y, x, whiteCount, redCount, checked); //uses imOutClass
    if (redCount != 0)
        percents[arrLen++] = float(redCount) / float(whiteCount);
    else
        percents[arrLen++] = percents[arrLen - 2]; //error with counts, set
to previous
}
inFile.close();

//re-swap imOutClass
imOutClass = imHold.clone();

return infoGain(classes, percents, arrLen);
}

float Image::infoGain(int classes[], float percents[], int arrLen)
{
    float entropy[100];

    //Initialize counter matrix
    int counters[2][2];
    counters[0][0] = 0;
    counters[0][1] = 0;
    counters[1][0] = 0;           // FN TN
    counters[1][1] = 0;         // FP TP

    float high;
    int index;
    int classTemp;
    float percentTemp;
    // for each value
    for (int i = 0; i < arrLen; i++)
    {
        high = 0;
        classTemp = classes[arrLen - i - 1];
        percentTemp = percents[arrLen - i - 1];

        //find next highest

```

```

    for (int j = 0; j < arrLen - i; j++)
    {
        if (percents[j] > high)
        {
            high = percents[j];
            index = j;
        }
    }
    //swap
    classes[arrLen - i - 1] = classes[index];
    percents[arrLen - i - 1] = percents[index];
    classes[index] = classTemp;
    percents[index] = percentTemp;
}

// Count
float midpoint;
entropy[arrLen - 1] = 10; //fill in missing midpoint
for (int i = 0; i < arrLen - 1; i++)
{
    //get midpoint
    midpoint = (percents[i] + percents[i + 1]) / 2;

    //reset counters to zero
    counters[0][0] = 0;
    counters[0][1] = 0;
    counters[1][0] = 0; // FN TN
    counters[1][1] = 0; // FP TP

    //for each nucleus, increment counters
    for (int j = 0; j < arrLen; j++)
    {
        //greater than midpoint, predicted class 0
        if (percents[j] > midpoint)
            counters[classes[j]][0]++;
        else
            counters[classes[j]][1]++;
    }

    //compute entropy for counters
    entropy[i] = getEntropy(counters);
}

//cout << "InfoGain Results:" << endl;
//for (int i = 0; i < arrLen; i++)
//    cout << "Class: " << classes[i] << "\tPercent: " << percents[i] <<
"\tEntropy: " << entropy[i] << endl;

//find highest entropy, go with that split point
float low = 10;
for (int i = 0; i < arrLen; i++)
{
    if (entropy[i] < low)
    {
        low = entropy[i];
        index = i;
    }
}

```

```

        midpoint = float((percents[index] + percents[index + 1]) / 2);
        return midpoint;
    }

float Image::getEntropy(int counters[][2])
{
    double returnVal = 0;
    double placeholder;
    int x, y;
    double sumTerms[3];
    for (y = 0; y < 3; y++)
    {
        sumTerms[y] = 0;
    }
    for (y = 0; y < 2; y++)
    {
        for (x = 0; x < 2; x++)
        {
            sumTerms[y] += counters[x][y];
        }
        sumTerms[2] += sumTerms[y];
    }

    for (y = 0; y < 2; y++)
    {
        placeholder = 0.0;
        for (x = 0; x < 2; x++)
        {
            if (sumTerms[y] != 0)
                placeholder += neg_xlogx(counters[x][y] / sumTerms[y]);
        }
        sumTerms[y] = sumTerms[y] / sumTerms[2] * placeholder;
        returnVal += sumTerms[y];
    }

    return float(returnVal);
}

float Image::neg_xlogx(float x)
{
    if (x == 0)
        return 0;
    else
        return float((-1)*(x)*(log(x) / log(2)));
}

/* CLASSIFICATION MEMBER FUNCTIONS */

void Image::classify(int nucleusThresh, int gradThresh)
{
    Vec3b nucleoli = { 255, 255, 255 };
    Vec3b nucleus = { 0, 0, 0 };
    Vec3b other = { 0, 0, 0 };
    int classPixel;

    imOutClass.create(ROWS, COLS, CV_8UC1);
}

```

```

for (int i = 0; i < ROWS; i++)
{
    for (int j = 0; j < COLS; j++)
    {
        classPixel = imClassShave.at<uchar>(i, j) % 2;
        if (classPixel) //if in a nucleus
        {
            if (gradGreaterThresh(i, j, gradThresh) >= nucleusThresh)
//3 or more changes greater than 20
            {
                //set to nucleoli
                imOutClass.at<Vec3b>(i, j) = nucleoli;
            }
            else //no
significant changes between pixel vals
            {
                //set to nucleus
                imOutClass.at<Vec3b>(i, j) = nucleus;
            }
        }
        else //not in nucleus
        {
            // set to other
            imOutClass.at<Vec3b>(i, j) = other;
        }
    }
}

void Image::classifyv2(int nucleusThresh, int gradThresh, int edgeThresh)
{
    //default arguments: 3, 24, 2
    Vec3b nucleoli = { 255, 255, 255 };
    Vec3b nucleus = { 0, 0, 0 };
    Vec3b other = { 0, 0, 0 };
    int classPixel;

    imOutClass.create(ROWS, COLS, CV_8UC1);

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            classPixel = imClassShave.at<uchar>(i, j) % 2;
            if (classPixel) //if in a nucleus
            {
                if (closeToEdge(i, j, edgeThresh) == 0) //not within
2 pixels of "other"
                {
                    if (gradGreaterThresh(i, j, gradThresh) >= nucleusThresh)
//3 or more changes greater than 20
                    {
                        //set to nucleoli
                        imOutClass.at<Vec3b>(i, j) = nucleoli;
                    }
                    else
//no significant changes between pixel vals
                    {

```



```

        //set to nucleus
        imOutClass.at<Vec3b>(i, j) = nucleus;
    }
}
else //close to
{
    //set to nucleus
    imOutClass.at<Vec3b>(i, j) = nucleus;
}
else //not in nucleus
{
    // set to other
    imOutClass.at<Vec3b>(i, j) = other;
}
}
}
}

void Image::classifyv4(double less, double high)
{
    Vec3b nucleoli = {0, 0, 255 };
    Vec3b nucleus = { 255, 255, 255 };
    Vec3b other = { 0, 0, 0 };
    int classPixel;
    float texPixel;

    imOutClass.create(ROWS, COLS, CV_8UC3);

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            classPixel = imClass.at<uchar>(i, j) % 2;
            if (classPixel)
            {
                texPixel = gradient[i][j].gradsAvg[3];
                if (texPixel >= less && texPixel <= high)
                {
                    imOutClass.at<Vec3b>(i, j) = nucleoli;
                }
                else
                {
                    imOutClass.at<Vec3b>(i, j) = nucleus;
                }
            }
            else
            {
                imOutClass.at<Vec3b>(i, j) = other;
            }
        }
    }
}

/* PRE/POSTPROCESSING MEMBER FUNCTIONS */

```

```

void Image::removeEdges(int offset)
{
    imClassShave.create(ROWS, COLS, 0);

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            if (imClass.at<uchar>(i, j) % 2 == 1)    //in a nucleus
            {
                if (closeToEdge(i, j, offset) != 0)    // && is close to
                edge
                {
                    imClassShave.at<uchar>(i, j) = 0;    //set to
                black
                }
                else
                {
                    imClassShave.at<uchar>(i, j) = 255;    //else set to
                white
                }
            }
            else
            {
                imClassShave.at<uchar>(i, j) = 0;
            }
        }
    }
}

void Image::removeRedEdges(int offset)
{
    //hypothetical denoise3, rmoves any red that touches black, replaces with white
    Mat newImClass;
    newImClass.create(ROWS, COLS, CV_8UC3);
}

void Image::denoise1(int min, int max)
{
    // for just denoise
    if (imOutClass.dims == 0)
        imOutClass = imRGB.clone();

    Mat checked;
    checked.create(ROWS, COLS, CV_8UC1);
    int classPixel;
    int checkedAlready;
    int size;

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            classPixel = imOutClass.at<Vec3b>(i, j)[2] % 2; //1 if full red or
            white
            checkedAlready = checked.at<uchar>(i, j);
            if (classPixel == 1 && checkedAlready != 255)    //if unchecked and a
            nucleus
        }
    }
}

```

```

        {
            size = getSize(i, j, checked); //count red pixels
            if (size < min || size > max)
            {
                deleteChunk(i, j); //delete red pixels
            }
        }
    }
}

void Image::denoise2(double percentage)
{
    // for just denoise
    if (imOutClass.dims == 0)
        imOutClass = imRGB.clone();

    Mat checked;
    Mat checked2;
    checked.create(ROWS, COLS, CV_8UC1);
    checked2.create(ROWS, COLS, CV_8UC1);
    int classPixel;
    int checkedAlready;
    int whiteCount = 0;
    int redCount = 0;

    for (int i = 0; i < ROWS; i++)
    {
        for (int j = 0; j < COLS; j++)
        {
            classPixel = imOutClass.at<Vec3b>(i, j)[2] % 2;
            checkedAlready = checked.at<uchar>(i, j);
            if (classPixel == 1 && checkedAlready != 255)
            {
                redCount = 0;
                whiteCount = 0;
                getCounts(i, j, whiteCount, redCount, checked);
                if ((float)redCount / whiteCount > percentage)
                {
                    deleteRed(i, j, checked2); //might be better to have a
new checked Mat for time complexity
                }
            }
        }
    }
}

void Image::accuracyEval(string filePath)
{
    if (imOutClass.dims == 0)
        imOutClass = imRGB.clone();

    string line;
    int x;
    int y;
    string actualClassS;
    int actualClass;
    int redCount;

```

```

int whiteCount;
Mat checked;
checked.create(imOutClass.rows, imOutClass.cols, CV_8UC1);
int predictedClass;

int conMatr[2][2];
conMatr[0][0] = 0;           // TN  FN
conMatr[0][1] = 0;           // FP  TP
conMatr[1][0] = 0;
conMatr[1][1] = 0;
double accuracy;
int count = 0;

ifstream inFile;
inFile.open(filePath.c_str());
getline(inFile, line); //junk header
while (!inFile.eof())
{
    // Get actual class
    getline(inFile, actualClassS, ',');
    if (actualClassS[0] == 'Y') //shortcut check
        actualClass = 1;
    else
        actualClass = 0;

    // Get predicted class
    getline(inFile, line, ',');
    x = stoi(line);
    getline(inFile, line, '\n');
    y = stoi(line);
    count++;
    redCount = whiteCount = 0;
    getCounts(y, x, whiteCount, redCount, checked);
    if (redCount == 0)
        predictedClass = 0;
    else
        predictedClass = 1;

    // Record in confusion matrix
    conMatr[actualClass][predictedClass]++;
}

accuracy = (double)(conMatr[1][1] + conMatr[0][0]) / (double)count;
cout << "Accuracy: " << accuracy << endl;
cout << '\t' << conMatr[0][0] << '\t' << conMatr[0][1] << endl;
cout << '\t' << conMatr[1][0] << '\t' << conMatr[1][1] << endl;
}

/* CONTRAST-BASED MEMBER FUNCTIONS*/

void Image::contrast(int scale, int neighborhood, int offset)
{
    // Convert to grayscale
    cvtColor(imRGB, imGray, CV_BGR2GRAY);
    // Bin
    bin(scale);

    // Set up Mats

```

```

Mat GLCM;
GLCM.create(pow(2, scale), pow(2, scale), CV_64FC1);
Mat contrast;
contrast.create(ROWS, COLS, CV_64FC1);
contrast = Scalar(0);
imOutTex.create(ROWS, COLS, CV_8UC1);

// For each pixel
for (int i = ((neighborhood / 2) + offset); i < ((ROWS - (neighborhood / 2) -
offset)); i++)
{
    for (int j = ((neighborhood / 2) + offset); j < (COLS - (neighborhood / 2)
- offset); j++)
    {
        if (imClass.at<uchar>(i, j) % 2)
        {
            // Calculate Gray-level co-ocurrence matrix
            getGLCM(GLCM, neighborhood, i, j, offset);

            // Add up and calculate contrast
            contrast.at<double>(i, j) = 0;
            for (int k = 0; k < pow(2, scale); k++)
            {
                for (int l = 0; l < pow(2, scale); l++)
                {
                    contrast.at<double>(i, j) += GLCM.at<double>(k,
1) * ((k - 1)*(k - 1));
                }
            }
        }
        else
        {
            contrast.at<double>(i, j) = 0;
        }
    }
}

// Normalize and save in texture Mat
for (int i = 0; i < ROWS; i++)
{
    for (int j = 0; j < COLS; j++)
    {
        imOutTex.at<uchar>(i, j) = round(contrast.at<double>(i, j)*pow(2, 8
- scale));
    }
}

void Image::bin(int scale)
{
    imGrayBinned.create(ROWS, COLS, CV_8UC1);
    int binNum = pow(2, scale);
    int binSize = 256 / binNum;
    int pixel;

    // for each pixel
    for (int i = 0; i < ROWS; i++)
    {

```

```

for (int j = 0; j < COLS; j++)
{
    pixel = imGray.at<uchar>(i, j);
    // check each bin
    for (int k = 0; k < binNum; k++)
    {
        if ((k*binSize <= pixel) && (pixel < (k + 1)*binSize))
        {
            // put in correct bin
            imGrayBinned.at<uchar>(i, j) = k;
            break;
        }
    }
}
}

void Image::getGLCM(Mat GLCM, int neighborhood, int row, int col, int offset)
{
    int pixel, neighbor;
    GLCM = Scalar(0);

    for (int i = (row - (neighborhood - 1) / 2); i <= (row + ((neighborhood - 1) /
2)); i++)
    {
        for (int j = (col - (neighborhood - 1) / 2); j <= (col + ((neighborhood -
1) / 2)); j++)
        {
            pixel = imGrayBinned.at<uchar>(i, j);
            // Up and down
            if (imClass.at<uchar>(i, j) % 2)
            {
                neighbor = imGrayBinned.at<uchar>(i + offset, j);
                GLCM.at<double>(pixel, neighbor) += 1;
                GLCM.at<double>(neighbor, pixel) += 1;
            }
            // Right diagonal
            if (imClass.at<uchar>(i, j) % 2)
            {
                neighbor = imGrayBinned.at<uchar>(i + offset, j + offset);
                GLCM.at<double>(pixel, neighbor) += 1;
                GLCM.at<double>(neighbor, pixel) += 1;
            }
            // Left and right
            if (imClass.at<uchar>(i, j) % 2)
            {
                neighbor = imGrayBinned.at<uchar>(i, j + offset);
                GLCM.at<double>(pixel, neighbor) += 1;
                GLCM.at<double>(neighbor, pixel) += 1;
            }
            // Left diagonal
            if (imClass.at<uchar>(i, j) % 2)
            {
                neighbor = imGrayBinned.at<uchar>(i - offset, j + offset);
                GLCM.at<double>(pixel, neighbor) += 1;
                GLCM.at<double>(neighbor, pixel) += 1;
            }
        }
    }
}

```

```

    }

    for (int i = 0; i < GLCM.rows; i++)
    {
        for (int j = 0; j < GLCM.cols; j++)
        {
            GLCM.at<double>(i, j) = GLCM.at<double>(i, j) / (8 *
pow(neighborhood, 2));
        }
    }
}

/* PRIVATE MEMBER FUNCTIONS */

int Image::calcTotalGradient(Vec3b center, Vec3b side)
{
    //Currently magnitude and direction
    int difference = 0;
    for (int i = 0; i < 3; i++)
    {
        difference += (side[i] - center[i]);
    }
    return difference;
}

void Image::calcColorGradient(PixelGrad& rainbow, Vec3b center, Vec3b side, int index)
{
    for (int k = 0; k < 3; k++)
    {
        rainbow.grads[k][index] = (side[k] - center[k]);
    }
}

void Image::setTotalGradient(int gradArr[], int row, int col)
{
    gradient[row][col].grads[3][0] = gradArr[0];
    gradient[row][col].grads[3][1] = gradArr[1];
    gradient[row][col].grads[3][2] = gradArr[2];
    gradient[row][col].grads[3][3] = gradArr[3];
}

void Image::setColorGradient(PixelGrad rainbow, int row, int col)
{
    for (int k = 0; k < 3; k++)
    {
        for (int l = 0; l < 4; l++)
        {
            gradient[row][col].grads[k][l] = rainbow.grads[k][l];
        }
    }
}

void Image::getAvgGrads(int offset, int index)
{
    Vec3b colorPixel;
    int classPixel;
    int count;
}

```

```

float avg;

//if no shaved version to use, move original to the shaved image
if (imClassShave.dims == 0)
    imClass.copyTo(imClassShave);

for (int i = 0; i < ROWS; i++)
{
    for (int j = 0; j < COLS; j++)
    {
        if (imClassShave.at<uchar>(i, j) % 2)
        {
            avg = 0;
            count = 0;
            if (i > offset - 1) //up
            {
                classPixel = imClassShave.at<uchar>(i - offset, j) % 2;
                if (classPixel)
                {
                    avg += (float)gradient[i][j].grads[index][0];
                    count++;
                }
            }
            if (i < (ROWS - offset)) //down
            {
                classPixel = imClassShave.at<uchar>(i + offset, j) % 2;
                if (classPixel)
                {
                    avg += (float)gradient[i][j].grads[index][1];
                    count++;
                }
            }
            if (j > offset - 1) //left
            {
                classPixel = imClassShave.at<uchar>(i, j - offset) % 2;
                if (classPixel)
                {
                    avg += (float)gradient[i][j].grads[index][2];
                    count++;
                }
            }
            if (j < (COLS - offset)) //right
            {
                classPixel = imClassShave.at<uchar>(i, j + offset) % 2;
                if (classPixel)
                {
                    avg += (float)gradient[i][j].grads[index][3];
                    count++;
                }
            }

            if (count != 0)
            {
                avg = avg / count;
                gradient[i][j].gradsAvg[index] = abs(avg);
            }
            else
            {

```



```

        gradient[i][j].gradsAvg[index] = 0;
    }
    }
else
{
    gradient[i][j].gradsAvg[index] = 0;
}
}
}
}

```

```

int Image::gradGreaterthan(int row, int col, int gradThresh)
{
    int count = 0;
    PixelGrad* ptr = &gradient[row][col];
    if (ptr->grads[3][0] >= gradThresh)
        count++;
    if (ptr->grads[3][1] >= gradThresh)
        count++;
    if (ptr->grads[3][2] >= gradThresh)
        count++;
    if (ptr->grads[3][3] >= gradThresh)
        count++;

    return count;
}

```

```

int Image::closeToEdge(int row, int col, int offset)
{
    int count = 0;

    for (int k = 1; k <= offset; k++)
    {
        //room above
        if (row > k - 1)
            if (imClass.at<uchar>(row - k, col) == 0)
                count++;

        //room below
        if (row < (ROWS - k))
            if (imClass.at<uchar>(row + k, col) == 0)
                count++;

        //room on left
        if (col > k - 1)
            if (imClass.at<uchar>(row, col - k) == 0)
                count++;

        //room on right
        if (col < (COLS - k))
            if (imClass.at<uchar>(row, col + k) == 0)
                count++;
    }

    return count;
}

```

```

int Image::getSize(int row, int col, Mat& checked)

```

```

{
    Vec3b red = { 0, 0, 255 };
    int count = 0;

    if (checked.at<uchar>(row, col) != 255
        && imOutClass.at<Vec3b>(row, col) == red)
    {
        count++;
        checked.at<uchar>(row, col) = 255;
        if (row > 0)
            count += getSize(row - 1, col, checked);
        if (row < (imOutClass.rows - 1))
            count += getSize(row + 1, col, checked);
        if (col > 0)
            count += getSize(row, col - 1, checked);
        if (col < (imOutClass.cols - 1))
            count += getSize(row, col + 1, checked);
    }
    return count;
}

void Image::deleteChunk(int row, int col)
{
    Vec3b red = { 0, 0, 255 };
    Vec3b white = { 255, 255, 255 };

    if (imOutClass.at<Vec3b>(row, col) == red)
    {
        imOutClass.at<Vec3b>(row, col) = white;
        if (row > 0)
            deleteChunk(row - 1, col);
        if (row < (ROWS - 1))
            deleteChunk(row + 1, col);
        if (col > 0)
            deleteChunk(row, col - 1);
        if (col < (COLS - 1))
            deleteChunk(row, col + 1);
    }
}

void Image::getCounts(int row, int col, int& whiteCount, int& redCount, Mat& checked)
{
    Vec3b red = { 0, 0, 255 };
    Vec3b pixel;

    if (checked.at<uchar>(row, col) != 255
        && imOutClass.at<Vec3b>(row, col)[2] % 2 == 1) //if unchecked and positive
    {
        whiteCount++;
        pixel = imOutClass.at<Vec3b>(row, col);
        if (pixel == red)
            redCount++;
        checked.at<uchar>(row, col) = 255;

        if (row > 0)
            getCounts(row - 1, col, whiteCount, redCount, checked);
        if (row < (ROWS - 1))
            getCounts(row + 1, col, whiteCount, redCount, checked);
    }
}

```

```

        if (col > 0)
            getCounts(row, col - 1, whiteCount, redCount, checked);
        if (col < (COLS - 1))
            getCounts(row, col + 1, whiteCount, redCount, checked);
    }
}

void Image::deleteRed(int row, int col, Mat& checked)
{
    Vec3b red = { 0, 0, 255 };
    Vec3b white = { 255, 255, 255 };

    if (imOutClass.at<Vec3b>(row, col)[2] % 2
        && checked.at<uchar>(row, col) != 255)
    {
        checked.at<uchar>(row, col) = 255;
        if (imOutClass.at<Vec3b>(row, col) == red)
        {
            imOutClass.at<Vec3b>(row, col) = white;
        }
        if (row > 0)
            deleteRed(row - 1, col, checked);
        if (row < (ROWS - 1))
            deleteRed(row + 1, col, checked);
        if (col > 0)
            deleteRed(row, col - 1, checked);
        if (col < (COLS - 1))
            deleteRed(row, col + 1, checked);
    }
}

```

ReadPixels.cpp

// ReadPixels.cpp : Defines the entry point for the console application.

```

#include "stdafx.h"
#include "opencv2/highgui.hpp"
#include "opencv2/core.hpp"
#include <string>
#include <fstream>
#include <iostream>

#include "GradientImage.h"

using namespace std;
using namespace cv;

void printAll(string, string);
void printRow(int row, Mat image, ofstream& outFile);
bool checkFile(string);

int main(int argc, char* argv[])
{
    // Declare variables in scope for everything
    string imageOriginalPath;
    string imageClassifiedPath;

```

```

string imageTrainPath;
string outImagePath;
string outImageClass;
string outImageTex;
string trainData;
int offset;
int color;
int high;
int low;
double percent;
string validData;
Image BigBoi;
bool goRun = false;

// ARG SET 1: texture only
// - color image
// - b/w image
// - output
// - offset
// - 0 for gray, 1 for color
// ARG SET 2: denoise
// - denoise image
// - output location
// - train data

if (argc == 5) //4 arguments, run denoise
{
    imageTrainPath = argv[1];
    imageOriginalPath = argv[2];
    outImagePath = argv[3];
    trainData = argv[4];
    if (checkFile(imageOriginalPath) && checkFile(trainData))
    {
        cout << "Beginning Denoise... ";
        BigBoi.setImage(imageOriginalPath, imageClassifiedPath);
        BigBoi.denoise2(BigBoi.getPercentage(trainData, imageTrainPath));
        BigBoi.denoise1(4, 255);
        BigBoi.saveClassified(outImagePath);
        cout << "done" << endl;
    }
}
else if (argc == 6) //4 arguments, run texture
{
    imageOriginalPath = argv[1];
    imageClassifiedPath = argv[2];
    outImagePath = argv[3];
    offset = stoi(argv[4]);
    color = stoi(argv[5]);
    if (checkFile(imageOriginalPath) && checkFile(imageClassifiedPath))
    {
        cout << "Getting Texture...";
        BigBoi.setImage(imageOriginalPath, imageClassifiedPath);
        BigBoi.removeEdges(2);
        if (color)
        {
            cout << "color version...";
            BigBoi.getColorGrads(offset);
            BigBoi.getColorAvgGrads(offset);
        }
    }
}

```

```

        BigBoi.saveGradColorTexture(outImagePath);
    }
    else
    {
        cout << "gray version...";
        BigBoi.getTotalGrads(offset);
        BigBoi.getTotalAvgGrads(offset);
        BigBoi.saveGradTotalTexture(outImagePath);
    }
    cout << "done" << endl;
}
}
else
{
    cout << "Invalid arguments" << endl;
}
return 0;
}

void printAll(string imagePath, string outFilePath)
{
    Vec3b pixel;

    Mat inImage = imread(imagePath, 1);
    ofstream outFile;
    outFile.open(outFilePath.c_str());

    for (int i = 0; i < inImage.rows; i++)
    {
        for (int j = 0; j < inImage.cols; j++)
        {
            outFile << i << "," << j << "," << (int)pixel[2] << "," <<
(int)pixel[1]
                << "," << (int)pixel[0] << "\n";
        }
    }
}

void printRow(int row, Mat inImage, ofstream& outFile)
{
    Vec3b pixel;

    int* redArr = new int[inImage.cols];
    int* greenArr = new int[inImage.cols];
    int* blueArr = new int[inImage.cols];

    for (int j = 0; j < inImage.cols; j++)
    {
        pixel = inImage.at<Vec3b>(row, j);
        redArr[0] = pixel[2];
        greenArr[1] = pixel[1];
        blueArr[2] = pixel[0];
    }

    for (int j = 0; j < inImage.cols; j++)
    {
        outFile << redArr[j] << " ";
    }
}

```

```

    }
    outFile << endl;
    for (int j = 0; j < inImage.cols; j++)
    {
        outFile << greenArr[j] << " ";
    }
    outFile << endl;
    for (int j = 0; j < inImage.cols; j++)
    {
        outFile << blueArr[j] << " ";
    }
}

bool checkFile(string fileName)
{
    bool good = false;

    //considered good if file opens
    ifstream infile;
    infile.open(fileName.c_str());
    if (infile)
        good = true;
    else
        cout << "Filepath <" << fileName << "> is incorrect" << endl;
    infile.close();

    return good;
}

//          EXTRA DRIVERS
/*          EXTRA DRIVERS

        //CONTRAST TEXTURE
//Image BigBoi(imageOriginalPath, imageClassifiedPath, offset);
//BigBoi.removeEdges(2);
//BigBoi.contrast(3, 1, 3);
//BigBoi.saveConTexture(outImageTex);

        //CLASSIFICATION
//Image BigBoi(imageOriginalPath, imageClassifiedPath);
//BigBoi.removeEdges(2); //shave off pink-to-purple
//BigBoi.readImage(offset); //read gradients
//BigBoi.getAvgGrads(offset); //get avg
//BigBoi.saveGradTexture(outImageTex); //save texture image for reference
//BigBoi.classifyv4(42, 112); //classify "between highest and
lowest nucleoli values"
//BigBoi.saveClassified(outImageClass); //save classified image!

        //TRAINING DATA
//Image BigBoi(imageOriginalPath, imageClassifiedPath, offset);
//BigBoi.removeEdges(2);
//BigBoi.readImage(offset);
//BigBoi.getAvgGrads(offset);
//BigBoi.readTrain(trainData);

        //DENOISE ONLY
//imageOriginalPath = argv[1]; //denoise image is in color
//imageClassifiedPath = argv[1]; //this one doesn't matter in this case

```

```

//outImagePath = argv[2];
//trainData = argv[3];
//if (checkFile(imageOriginalPath) && checkFile(imageClassifiedPath))
//{
//    cout << "Beginning Denoise... ";
//    BigBoi.setImage(imageOriginalPath, imageClassifiedPath);
//    BigBoi.denoise2(BigBoi.getPercentage(trainData));
//    BigBoi.denoise1(3, 255);
//    BigBoi.saveClassified(outImagePath);
//    cout << "done" << endl;
//}
//
//    //ACCURACY ADD-ON
//    validData = argv[4];
//    if (checkFile(validData))
//        BigBoi.accuracyEval(validData);
//}

//    //FULL DRIVER
//    /*

// Arg set 0: Grad Texture
//          original classified output offset
// Arg set 1: Classify
//          (...) trainData
// Arg set 2: Denoise
//          (...) (...) low high percent
// Arg set 3: Validation
//          (...) (...) (...) validData

if (argc >= 5)    // Arg set 0, minimum required arguments
{
    //    Get arguments
    imageOriginalPath = argv[1];
    imageClassifiedPath = argv[2];
    outImagePath = argv[3];
    outImageTex = outImagePath.substr(0, outImagePath.find_last_of(".")) + "-
tex.tif";
    outImageClass = outImagePath.substr(0, outImagePath.find_last_of(".")) + "-
class";
    offset = stoi(argv[4]);

    //    Do processing for this level: get texture metric
    goRun = (checkFile(imageOriginalPath) && checkFile(imageClassifiedPath));
    if (goRun)
    {
        cout << "Getting texture... ";
        BigBoi.setImage(imageOriginalPath, imageClassifiedPath);
        BigBoi.removeEdges(2);
        //BigBoi.getColorGrads(offset);
        //BigBoi.getColorAvgGrads(offset);
        //BigBoi.saveGradColorTexture(outImageTex);
        BigBoi.getTotalGrads(offset);
        BigBoi.getTotalAvgGrads(offset);
        //BigBoi.saveGradTotalTexture(outImageTex);
        cout << "done" << endl;
    }

    if (argc >= 6)    // Arg set 1

```

```

{
    // Get arguments
    trainData = argv[5];

    // Do processing for this level: Classify
    goRun = (checkFile(trainData) && goRun);
    if (goRun)
    {
        cout << "Classifying... ";
        BigBoi.getSplits(trainData);
        BigBoi.classifyv4(BigBoi.getLow(), BigBoi.getHigh());
        BigBoi.saveClassified(outImageClass + "-1.tif");
        cout << "done" << endl;
    }
}
if (argc >= 9) // Arg set 2
{
    // Get arguments
    low = stoi(argv[6]);
    high = stoi(argv[7]);
    percent = stof(argv[8]);

    // Do processing for this level: Denoise
    if (goRun)
    {
        cout << "Cleaning up... ";
        BigBoi.denoise2(percent);
        BigBoi.saveClassified(outImageClass + "-2.tif");
        BigBoi.denoise1(low, high);
        BigBoi.saveClassified(outImageClass + "-3.tif");
        cout << "done" << endl;

        cout << "Parameters: " << endl;
        cout << "  Offset: " << offset << endl;
        cout << "  Classification Range: " << BigBoi.getLow() << ", "
<< BigBoi.getHigh() << endl;
        cout << "  Denoise Island Range: " << low << ", " << high <<
endl;
        cout << "  Denoise Percent Threshold: " << percent << endl;
    }
}
if (argc >= 10) // Arg set 3
{
    // Get arguments
    validData = argv[9];

    // Do processing for this level: Accuracy
    goRun = (checkFile(validData) && goRun);
    if (goRun)
    {
        BigBoi.accuracyEval(validData);
    }
}
cout << "Processing finished" << endl;
}
else // argc < 5
{
    std::cout << "Not enough arguments" << endl;
}

```



```

        std::cout << "Arg set 0: original_image classified_image out_image offset"
<< endl;
        std::cout << "Arg set 1: (arg0) train_data" << endl;
        std::cout << "Arg set 2: (arg1) low high percent" << endl;
        std::cout << "Arg set 3: (arg2) valid_data" << endl;
    }
*/

```

Accuracy computation – the standard confusion matrix was used, where validation data listed pixels of Class 1 and Class 0. For nucleoli however, there is a slight variation where instead of receiving in validation data pixels of each class, the program receives a list of pixel coordinates for nuclei and whether they contain a visible nucleolus.

ConMatr.h

```

#ifndef CONMATR_H
#define CONMATR_H

#include "stdafx.h"
#include <iostream>
#include <string>
#include <fstream>
#include "opencv2/highgui.hpp"
#include "opencv2/core.hpp"

using namespace std;
using namespace cv;

class ConMatr {
private:
    string valid;
    Mat image;
    int conMatr[2][2];
    int arrLen;

    Vec3b nucleus = { 0, 0, 255 };
    Vec3b nucleoli = { 255, 0, 0 };
    Vec3b lumen = { 0, 255, 0 };
    Vec3b other = { 0, 0, 0 };

public:
    ConMatr(string, string);
    double getAccuracy(int); //1 for nuclei, 2 for lumina, 3 for nucleoli
    int isNucleus(int, int);
    int isLumen(int, int);
    int isNucleolus(int, int);
    void getCounts(int, int, int&, int&, Mat&);
    void print();
};

#endif

```

ConMatr.cpp

```
#include "stdafx.h"
#include "ConMatr.h"

ConMatr::ConMatr(string validFile, string imageFile)
{
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            conMatr[i][j] = 0;

    valid = validFile;
    image = imread(imageFile, 1);
}

double ConMatr::getAccuracy(int option)
{
    string line;
    int x, y;
    string actualClassS;
    int actualClass;
    int predictedClass;
    int count = 0;

    ifstream inFile;
    inFile.open(valid.c_str());
    getline(inFile, line); //junk header
    while (!inFile.eof())
    {
        // Get actual class
        getline(inFile, actualClassS, ',');
        if (actualClassS[0] == '0') //shortcut check
            actualClass = 0;
        else
            actualClass = 1;

        getline(inFile, line, ',');
        x = stoi(line);
        getline(inFile, line, '\n');
        y = stoi(line);

        if (option == 1)
            predictedClass = isNucleus(y, x);
        else if (option == 2)
            predictedClass = isLumen(y, x);
        else if (option == 3)
            predictedClass = isNucleolus(y, x);

        conMatr[actualClass][predictedClass]++;
        count++;
    }

    double accuracy = (double)(conMatr[1][1] + conMatr[0][0]) / (double)count;
    return accuracy;
}
```

```

int ConMatr::isNucleus(int row, int col)
{
    Vec3b pixel = image.at<Vec3b>(row, col);
    if (pixel == nucleus || pixel == nucleoli)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int ConMatr::isLumen(int row, int col)
{
    Vec3b pixel = image.at<Vec3b>(row, col);
    if (pixel == lumen)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int ConMatr::isNucleolus(int row, int col)
{
    int nucleusCount, nuclCount;
    nucleusCount = nuclCount = 0;
    Mat checked;
    checked.create(image.rows, image.cols, CV_8UC1);

    getCounts(row, col, nucleusCount, nuclCount, checked);
    if (nuclCount == 0)
        return 0;
    else
        return 1;
}

void ConMatr::getCounts(int row, int col, int& nucleusCount, int& nuclCount, Mat&
checked)
{
    Vec3b pixel = image.at<Vec3b>(row, col);

    if (checked.at<uchar>(row, col) != 255
        && (pixel == nucleus || pixel == nucleoli))    //if unchecked and in
nucleus
    {
        nucleusCount++;
        if (pixel == nucleoli)
            nuclCount++;
        checked.at<uchar>(row, col) = 255;

        if (row > 0)
            getCounts(row - 1, col, nucleusCount, nuclCount, checked);
        if (row < (image.rows - 1))

```

```

        getCounts(row + 1, col, nucleusCount, nuclCount, checked);
    if (col > 0)
        getCounts(row, col - 1, nucleusCount, nuclCount, checked);
    if (col < (image.cols - 1))
        getCounts(row, col + 1, nucleusCount, nuclCount, checked);
    }
}

void ConMatr::print()
{
    cout << "\t" << conMatr[0][0] << "\t" << conMatr[0][1] << endl;
    cout << "\t" << conMatr[1][0] << "\t" << conMatr[1][1] << endl;
}

```

CancerAccuracy2.0.cpp

```
// CancerAccuracy2.0.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#include <string>
#include <fstream>
#include "opencv2/highgui.hpp"
#include "opencv2/core.hpp"

#include "ConMatr.h"

using namespace std;
using namespace cv;

bool checkFile(string);

int main(int argc, char* argv[])
{
    string validNucleus = argv[1];
    string validLumen = argv[2];
    string validNuclo = argv[3];
    string imagePath = argv[4];

    if (checkFile(validNucleus) && checkFile(validLumen)
        && checkFile(validNuclo) && checkFile(imagePath))
    {

        ConMatr nucleusMat(validNucleus, imagePath);
        ConMatr lumenMat(validLumen, imagePath);
        ConMatr nucloMat(validNuclo, imagePath);

        double accuracyNucleus, accuracyLumen, accuracyNuclo;
        accuracyNucleus = nucleusMat.getAccuracy(1);
        accuracyLumen = lumenMat.getAccuracy(2);
        accuracyNuclo = nucloMat.getAccuracy(3);

        string fileName = imagePath.substr(imagePath.find_last_of('\\') + 1,
imagePath.find('.'));
        cout << "Accuracy for " << fileName << ": " << endl;
        cout << "Nucleus Classification: " << accuracyNucleus * 100 << "%" << endl;
        nucleusMat.print();
        cout << "Lumen Classification: " << accuracyLumen * 100 << "%" << endl;
        lumenMat.print();
        cout << "Nucleolus Classification: " << accuracyNuclo * 100 << "%" << endl;
        nucloMat.print();
    }

    return 0;
}

bool checkFile(string fileName)
{
    bool good = false;
```

```
//considered good if file opens
fstream infile;
infile.open(fileName.c_str());
if (infile)
    good = true;
else
    cout << "Filepath <" << fileName << "> is incorrect" << endl;
infile.close();

return good;
}
```