# MACHINE LEARNING APPLICATIONS IN DIGITAL PATHOLOGY

Leo Berman
Albert Bulik
Yuan Nghiem

# Table of Contents

# Table of Figures

# 1 Chosen Solution

Training a machine learning model to predict malignancy requires many interlocking components. First, a biopsy slide is digitized by a pathologist at Temple University Health, and the slide image is stored in the TUH Digital Pathology data store (1.1). The slide is then segmented into frames and windows (1.2). An algorithm extracts the features of each window (1.3), and the number of features is reduced using a dimension reduction algorithm (1.4, 1.5). A model is trained on the windows (1.6), and the model output for a test case (a segmented slide image) is postprocessed (2). Finally, the model is tuned and evaluated (Fig. 1).

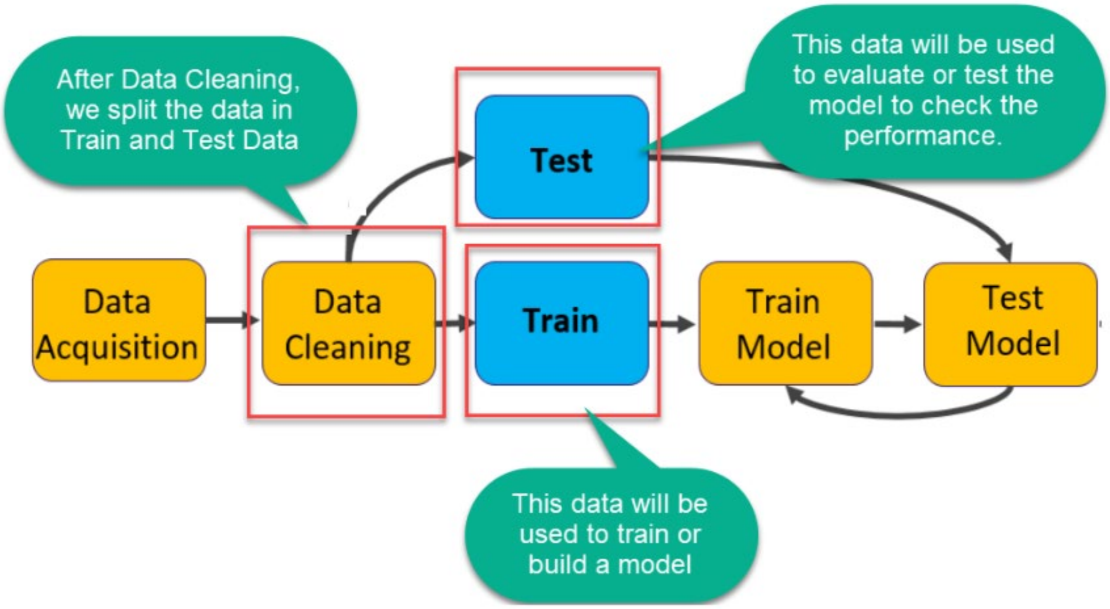

After Data Cleaning, we split the data in Train and Test Data
Test
This data will be used to evaluate or test the model to check the performance.
Data Acquisition
Data Cleaning
Train
Train Model
Test Model
This data will be used to train or build a model

*Fig. 1: Example of the Train & Dev Cycle [1]*

## 1.1 Digitizing and Storing Biopsy Slides

The first job of any data analyst is to collect data. For the purpose of machine learning, data must have clearly defined input and output. For example, if one were predicting height based on age, height would be the output and age would be the input; we would collect and record the height and age of a random sample of the population, and this dataset would form the basis of our predictions. In our case, we are predicting the presence of malignant tissue in a patient biopsy, so we require annotated photos of biopsy slides. The photos themselves (or some aspect of each photo) will be the input for our model, and the annotations will be the model's output. See for a representative example of a biopsy slide.

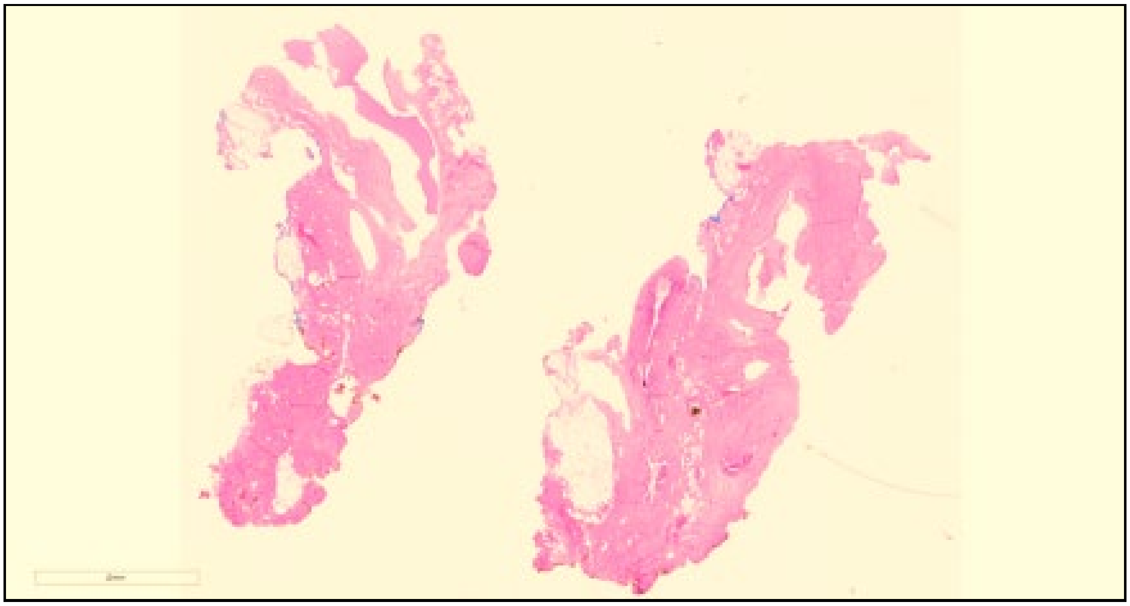


*Fig. 2: Digitized biopsy slide with two tissue samples.*

What do we mean by *annotation*? In Fig. 1, several shapes will have been traced over the image by a pathologist. Each shape has a corresponding label with a specific meaning. Each label and its meaning are listed in Tab. 1 by ascending order of urgency. The higher the urgency, the more likely it is that tissue from the sample will develop into malignant tissue (ductal carcinoma in situ, or DCIS).

*Tab. 1: Slide labels, their meanings, and order of urgency.*

| Label | Description | Urgency |
|---|---|---|
| **Unlab** | Unlabeled tissue area; non-annotated area of the biopsy slide. | 0 |
| **Bckg** | Background stroma tissue surrounding lobules and ducts. | 1 |
| **Norm** | Normal tissue, including lobules or ducts with empty spaces called lumen (areas of the slide that allow light to shine through). | 2 |
| **Null** | Indistinguishable tissue, often caused by poor cuts of the sample. | 3 |
| **Artf** | Artifacts in the slide image, such as pen marks. | 4 |
| **Nneo** | Non-neoplastic tissue, a non-malignant lesion. Includes fibrosis and hyperplasia. May develop into a malignancy. | 5 |
| **Infl** | Inflammation, high concentration of small dots typically found around stroma. | 6 |
| **Susp** | Suspicious tissue, at risk for developing into DCIS. | 7 |
| **Indc** | Invasive ductal carcinoma in situ, cancer tissue, freeform malignant tissue invading stroma. | 8 |
| **Dcis** | Ductal carcinoma in situ, dense and tightly enclosed cancer tissue. | 9 |

*Fig. 3: Leica Biosystems Aperio AT2 biopsy slide scanner.*

Biopsy slides are prepared and scanned by the pathology lab at Temple University Health (TUH) using a Leica Biosystems Aperio AT2 (Fig. 3). All identifying patient information is removed and the slides are stored in the TUH Digital Pathology database in Scanscope Virtual Slide (SVS) format. An SVS file contains the labels for each slide annotation and the slide image at multiple resolutions (1:1, 4:1, 16:1, 32:1). The images are compressed using JPEG 2 (Joint Photographic Experts Group) compression with three color values per pixel (red, green, and blue, or, 'RGB'); the full resolution of a biopsy slide is 50,000 x 50,000 pixels.

## 1.2 Slide Segmentation

Unlike machine learning (ML) models that focus on whole-slide image (WSI) classification, our approach is unique in that we segment slide images into smaller areas called frames. Our ML model is trained on frames, and makes predictions on (i.e., assigns labels from Tab. 1 to) frames. The benefit of this approach is that it allows a reviewing pathologist to quickly localize the affected area of a malignancy. For each slide in the training dataset, the slide is divided into a uniform grid of 200 x 200 pixels per frame (Fig. 4).

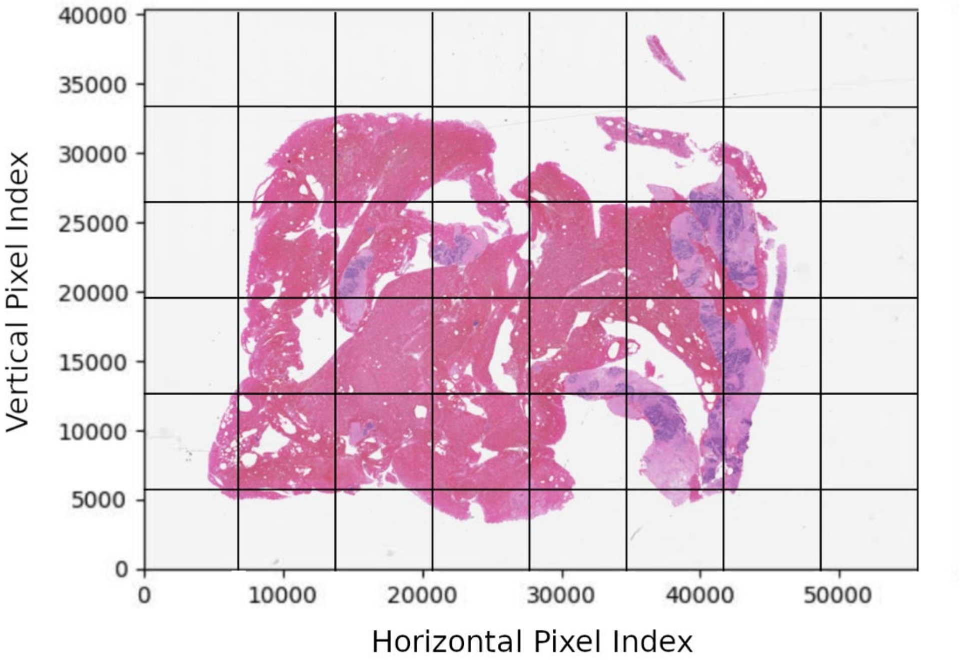


*Fig. 4: Biopsy slide segmented into frames; frames are larger than 200x200 pixels to make them easier to see.*

When we pass frames as input to the training algorithm, we allow some overlap between adjacent frames. Prior to training, frame boundaries are extended by 25 pixels on every side. The extended 250 x 250 area is what we call a *window* (Fig. 5). The amount of overlap is a non-negative number set by the training engineer, and the reason that we desire overlap is that it encodes context into the training data.

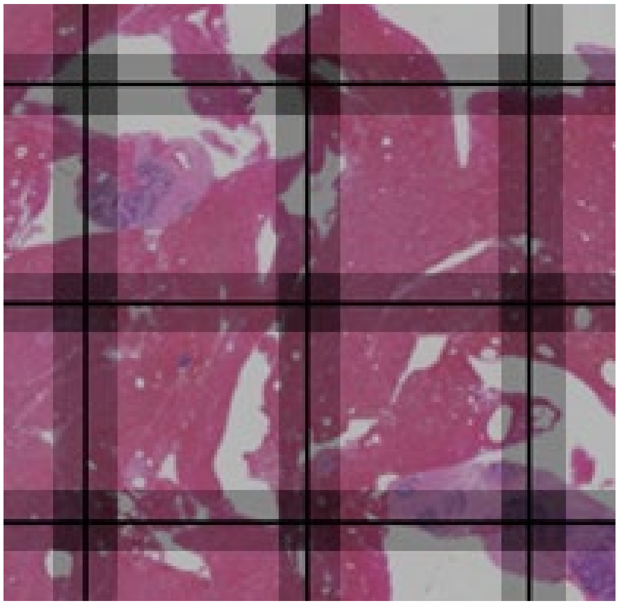


*Fig. 5: Borders of windows overlapping with adjacent windows and frames.*

Context is a broad subject in machine learning but is, in essence, the surrounding information or environment that may affect a prediction. For instance, in language processing certain letters are more likely to appear in certain positions in a word, and words have different meanings depending on their position in a sentence.

In our application, context is correlation between adjacent frames. If one frame is classified DCIS, then incorporating context raises the likelihood that adjacent frames are classified DCIS as well. This allows us to keep predictions relatively consistent.

## 1.3 Feature Extraction

In machine learning lingo, *features* are subsets of an input datum. If we were predicting height based on weight and age, weight and age would be features of the input data. For simple examples like this, features are often readily apparent. For the purpose of predicting label classifiers from bitmap images, the features are not readily apparent. The question of extracting features is analogous to the question of determining what aspects the data yield the most predictive ability and information for our model. Our input datum (the smallest unit of input) to the training algorithm is a bitmap image for a single window, extracted from a biopsy slide. Each bitmap image is separated into 3 layers (one per color) and each layer is a 250 x 250 matrix, encoding color intensity per pixel. How can we organize the color matrices into useful features for our training algorithm?

The most obvious way to extract features for a bitmap image is to consider each pixel value a separate feature. For one window we would have 250x250x3 = 187,500 features. As simple and direct as this method is, it often yields surprising results when training a convolutional neural network (CNN). We discuss CNNs and scoring metrics in later sections, but to put the performance of this method into context, a CNN trained directly on image data yielded 86.64% accuracy and an 89.00% F1 score in a study of 400 slide images; the best results in the same study and dataset were 96.00% accuracy and 96.00% F1 score yielded by a CNN trained on statistics from a contourlet transform and histograms. [2]

We don't want to limit ourselves to one method of feature extraction; the best method varies by application. Let's consider other methods of characterizing input data.

### 1.3.1 Frequency Analysis Overview

Before discussing alternative methods of feature extraction, we'll briefly introduce the math behind frequency transforms. Any signal $f$ can be decomposed into frequencies $F$. The relationship is one-to-one; if we know $F$, then $f$ can be fully reconstructed. This means that every signal is a superposition of periodic functions, and the frequency spectrum encodes magnitude and phase for every frequency necessary to reconstruct $f$. It's worth noting that each window matrix in our training set is a sequence of discrete values (color intensity per pixel). This allows us to leverage aspects of discrete series that we would otherwise not be able to do for continuous signals.

Consider a simple sequence of bits $f = \{1,0,1,1\}$. We note that the sequence contains $N = 4$ bits, so it requires 4 steps to traverse—$N$ is the *fundamental period* of this bit sequence. Therefore, the *fundamental frequency* $\omega_0 = 2\pi/4 = \pi/2$ corresponds to a quarter rotation of a circle. It can be shown that only 4 frequencies are required to reconstruct the signal $\left(0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\right)$ when $f$ is applied to a discrete Fourier transform (DFT). The transform pair of a DFT is provided in Eqn. 1.

$$F[k] = \sum_{n=0}^{N-1} f[n]\, e^{-j\frac{2\pi}{N}kn}, \qquad f[n] = \frac{1}{N}\sum_{k=0}^{N-1} F[k] e^{j\frac{2\pi}{N}kn},$$

*Equation 1: Discrete Fourier transform analysis (left) and synthesis (right) equations.*

where $n$ is the index of $f$, $N$ is the fundamental period of $f$, and $F[k]$ is the $k^{th}$ harmonic in frequency spectrum $F$. If $r$ is an integer, then substituting $k + rN$ for $k$ yields the same analysis expression (Eqn. 2).

$$F[k + rN] = \sum_{n=0}^{N-1} f[n]\, e^{-j\frac{2\pi}{N}(k+rN)n} = \sum_{n=0}^{N-1} f[n]\, e^{-j\frac{2\pi}{N}kn} e^{-j\frac{2\pi}{N}(rN)n} = \sum_{n=0}^{N-1} f[n]\, e^{-j\frac{2\pi}{N}kn}$$

*Equation 2: Substituting a sum of harmonic k and integer multiple of the fundamental rN.*

This implies that harmonics $\geq N$ are copies of harmonics in $[0, N-1]$, because $e^{-j\frac{2\pi}{N}(rN)n}$ can only be a multiple of $e^{-j2\pi} = 1$.

The DFT transforms a series of length $N$ to a frequency spectrum of equal length $N$. The DFT of a 2-dimensional matrix is more involved but the result is the same. If we apply a 2D DFT to a 250x250 window matrix of pixels, then we receive a 250x250 matrix of every combination of horizontal and vertical harmonics. This will become important later when discussing dimension reduction.

### 1.3.2 Feature extraction by frequency analysis

Our choice of frequency transform is the discrete cosine transform II (DCT-II). The DCT-II $F_{c2}$ is related to the DFT of $f$ if $f$ were mirrored on its final value and $2N$-points long (Eqn. 3). The way $f$ is mirrored is illustrated in Fig. 6.

$$F_{c2}[k] = 2\Re e\left\{F_{2N}[k]e^{-\frac{j\pi}{2N}k}\right\} = 2\sum_{n=0}^{N-1} f_{2N}[n]\cos\left(\frac{\pi k(2n+1)}{2N}\right),$$

*Equation 3: Relationship between DCT-II $F_{c2}[k]$ and DFT $F_{2N}[k]$ analysis equation.*

$$f_{2N}[n] = \left(\frac{1}{2}F_{c2}[0] + \sum_{k=1}^{N-1} F_{c2}[k]\cos\left(\frac{\pi k(2n+1)}{2N}\right)\right)\frac{2}{N}$$

*Equation 4: Synthesis equation of DCT-II.*

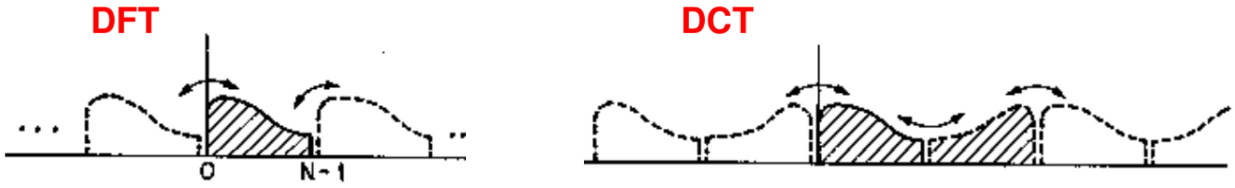


*Fig. 6: Illustration of f_2N [n] (right) and f[n] as it is applied to a DFT (left) (the DFT is identical to the discrete Fourier series of the periodic extension of the signal).*

Mirroring the signal gives DCT-II an advantage over DFT: the DCT-II frequency spectrum is much more compact than DFT. Fig. 6 compares the DFT and DCT-II spectra for the series in Fig. 7. Note that DCT-II coefficients approach zero much quicker than DFT coefficients.

The spectral compaction of DCT-II is desirable because every frequency is a feature of our model. The more features we can remove without losing information about the signal, the better our model's predictive and computational performance will be.

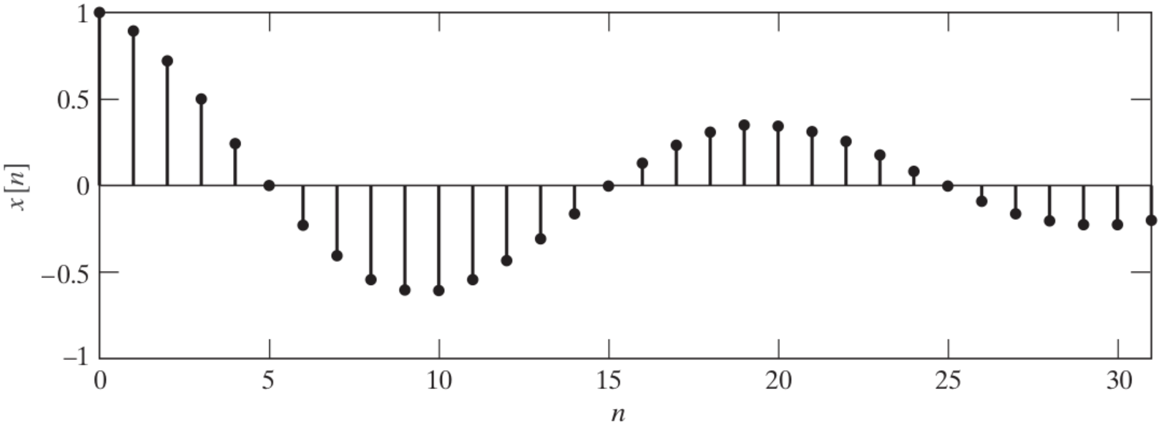
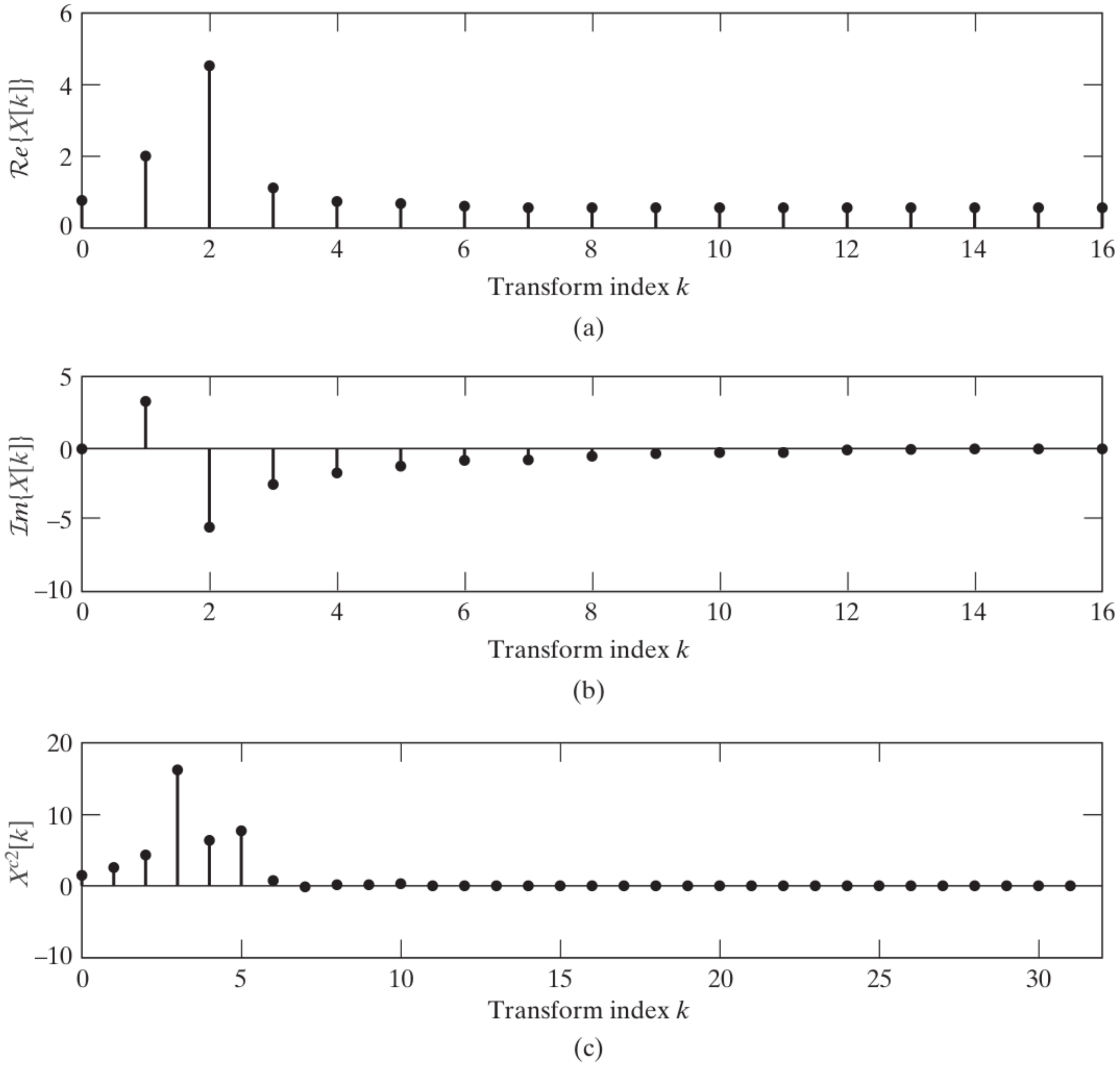


*Fig. 7: Plot of test signal $x[n] = 0.9^n\ cos(0.1\pi n)$, $n \in [0,31]$. [3]*



(a)

(b)

(c)

*Fig. 8: Plot of real (a) and imaginary (b) parts of 32-point DFT of $x[n]$ (pts 16-31 are mirrored). Plot of 32-point DCT (c). [3]*

Worth noting is that the synthesis equation of DCT-II (Equation 3) is not *orthogonal*. Orthogonality is a property of matrices that simplifies calculations. The inverse of an orthogonal matrix $\boldsymbol{C}^{-1}$ is its transpose $\boldsymbol{C}^{T}$. If we substitute $\sqrt{\frac{1}{4N}}f_{2N}[0]$ for $f_{2N}[0]$ and $\sqrt{\frac{1}{2N}}f_{2N}[n]$ for $f_{2N}[n]$, $n \neq 0$, Eqn. 3 simplifies to Eqn. 5.

$$F_{c2}[k] = \sqrt{\frac{1}{N}}f_{2N}[0] + \sum_{n=1}^{N-1} f_{2N}[n]\sqrt{\frac{2}{N}}\cos\left(\frac{\pi k(2n+1)}{2N}\right)$$

*Equation 5: Orthogonal variant of DCT-II analysis equation.*

Equation 5 can be rewritten as the product of an orthogonal matrix $\boldsymbol{C}$ (Eqn. 6) and the signal column vector $f_{2N}[n]$, $\boldsymbol{F} = \boldsymbol{Cf}$. The signal may be reconstructed by the inverse operation, $\boldsymbol{f} = \boldsymbol{C}^{-\boldsymbol{1}}\boldsymbol{F} = \boldsymbol{C}^{\boldsymbol{T}}\boldsymbol{F}$.

$$\boldsymbol{C}_{k,n} = \begin{cases} \sqrt{\frac{1}{N}}, & k = 0 \\ \sqrt{\frac{2}{N}} \cos\left(\frac{\pi k(2n+1)}{2N}\right), & 0 < k < N \end{cases}$$

*Equation 6: Orthogonal matrix representation of DCT-II analysis equation.*

In two dimensions, we simply apply the one-dimensional DCT-II along the columns and rows of $\boldsymbol{C}$; $\boldsymbol{F} = \boldsymbol{C}^{\boldsymbol{T}}\boldsymbol{fC}$. And to reconstruct the signal, we repeat the operation in reverse order, $\boldsymbol{f} = \boldsymbol{CFC}^{\boldsymbol{T}}$. (This may also be done via Kronecker product, $vec(\boldsymbol{f}) = \boldsymbol{C} \otimes \boldsymbol{C}vec(\boldsymbol{F})$.)

We may apply either 2D DCT-II or 1D DCT-II to a window matrix. In the one-dimensional case, all rows of a window matrix would concatenate to form a single vector. Changes in columns of pixels would be encoded as harmonics of the larger vector, so why use 2D DCT-II at all?

The advantage of 2D DCT-II is that it has the greatest spectral compaction of all frequency transforms discussed thus far. It encodes greater information in fewer frequencies than either 1D DCT-II or DFT, and we seek to eliminate redundant features. Compare the reconstructed images for 2D DCT-II in Fig. 9 to those of 1D DCT-II in Fig. 8. For $10 \leq k \leq 20$ frequencies, the 1D DCT-II reconstruction degrades significantly.

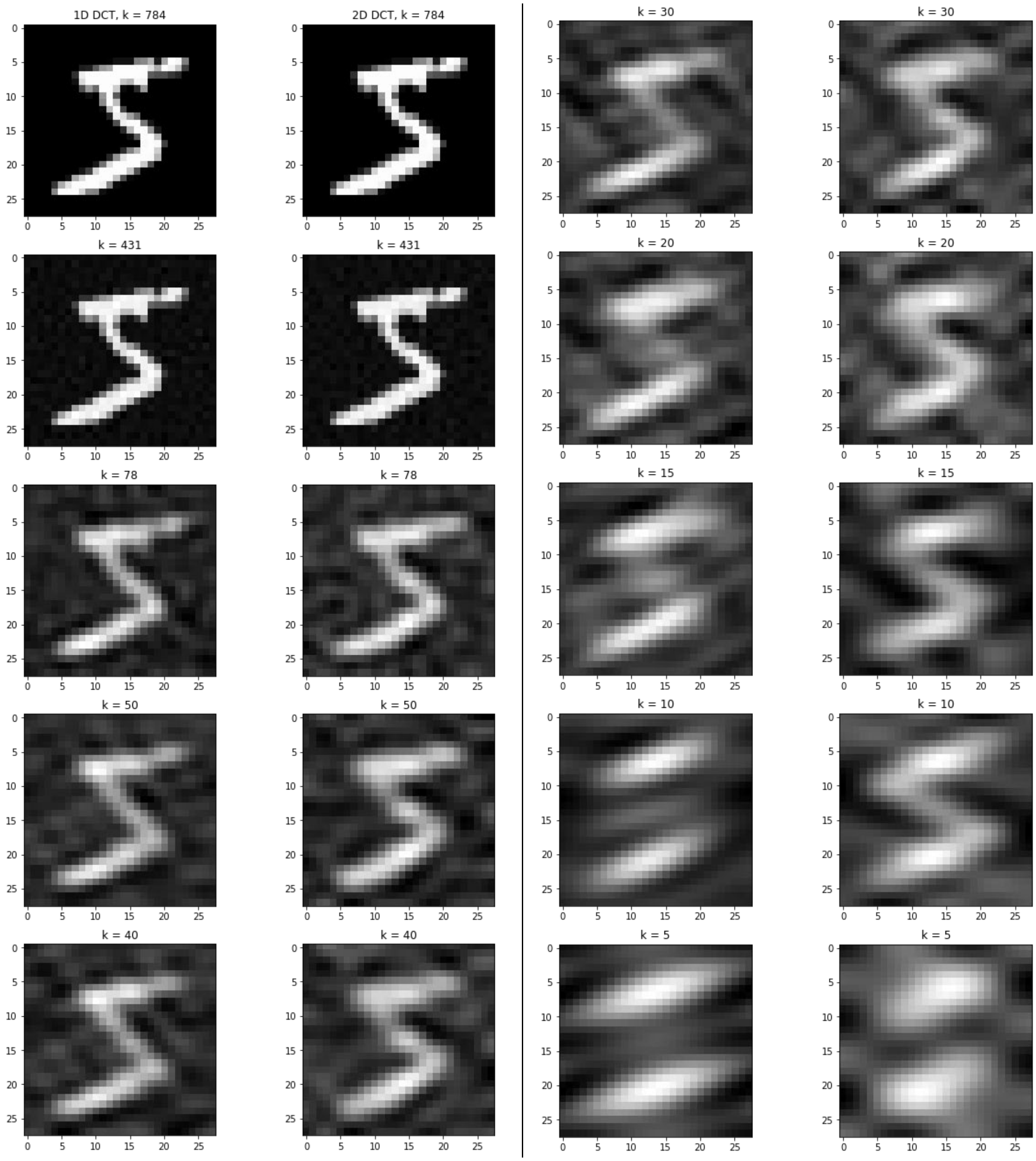


*Fig. 9: Example image reconstructed from k frequencies using 1D DCT-II (left) and 2D DCT-II (right). [4]*

## 1.4 Dimension reduction

After extracting the features of each window, we can begin training. However, if we were to apply our features as-is, we would run into a problem in science and mathematics known as the curse of dimensionality. Models used for classification make predictions by dividing the feature space into label-specific regions. For example, if age and weight were used to predict heart disease, then the coordinate plane of age on one axis, and weight on the other axis, would be divided into separate regions of heart disease and not heart disease (Fig. 10). A trained model would predict whether a person has heart disease based on where their age and weight locate a point on the plane.

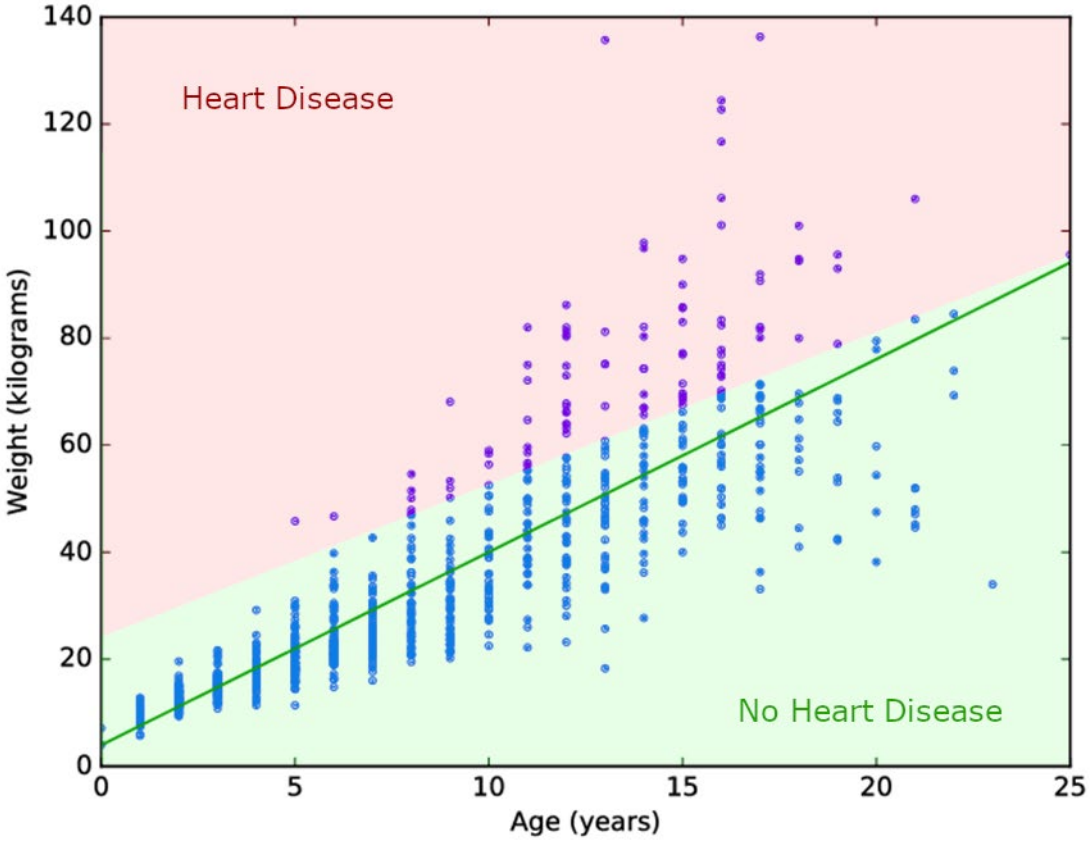


*Fig. 10: Example plot of weight vs. age of a population sample.*

The line (or *decision surface*) that separates the red and green regions in Fig. 10 can be extended to 3 dimensions (for example, if height were a feature), or n dimensions for n features in $R^n$. In 3 dimensions the decision surface would be a plane, and for >3 dimensions, the decision surface would be a hyperplane.

The curse of dimensionality states that in higher dimensions, every point in a dataset is approximately the same distance to every other point. [5] This is troublesome for training because we prefer each class in the training data to be tightly separated in feature space from every other class. The question then becomes, how many dimensions is too many for the training data?

A good rule of thumb is to use fewer features than the square root of training inputs. Let $p$ be the number of features in a window, and $n$ be the number of windows in our training set; we desire $p < \sqrt{n}$. We have 3,505 tissue images, and each image is 50,000x50,000 pixels.

These are segmented into 200x200 frames for classification and extended to windows. Therefore, we have $n = (3{,}505[slides])\left(\frac{50{,}000^2}{200^2}\left[\frac{windows}{slide}\right]\right) = 219{,}062{,}500$ windows in our training set. We must limit ourselves to $p < \sqrt{219{,}062{,}500}$ or $p < 14{,}800$ features.

This poses a problem; we need fewer than 14,800 features, but bitmap images and frequency transformations for each window both yield 250x250x3 = 187,500 features. We need to reduce the number of features from 187,500 to 14,800. We mentioned in passing in section 1.3.2 that we desire frequency-based feature extraction methods with the greatest spectral compaction. We may prune frequencies above a threshold to constrain the number of features, but we may also apply linear or non-linear transforms to reduce that number even further. There is a tradeoff between the number of features pruned and the predictive ability of features lost to pruning. This is a subject of data analysis known as dimension reduction.

## 1.5 Principal Component Analysis (PCA)

The most popular method of dimension reduction is principal component analysis (PCA). The idea behind PCA is relatively simple: we find the line of best fit between every two features, and best-fit lines (*eigenvectors*) become our new axes in the principal component (PC) space. Each datapoint in feature space is projected onto the eigenvector (Fig. 11). Principal components subsequently replace the original features for training.

While PCA produces the same number of principal components as features, and ostensibly does nothing to reduce dimensionality, it allows the analyst to neatly prune PCs. This is because PCA orders PCs by variance, and half of all principal components have greater variance than the features from which they're derived. If two features are highly correlated, then one PC axis will have high variance (datapoints far from the mean), and the other PC will have low variance (datapoints concentrated around the mean).

Variance is our metric of predictive ability with respect to features and principal components. Low-variance principal components can be safely removed without losing useful information for training. In practice, PCA leads to substantial dimension reduction as the variance of principal components decays geometrically. Fig. 12 displays the normalized variance of each PC for a high-resolution test image applied to PCA.

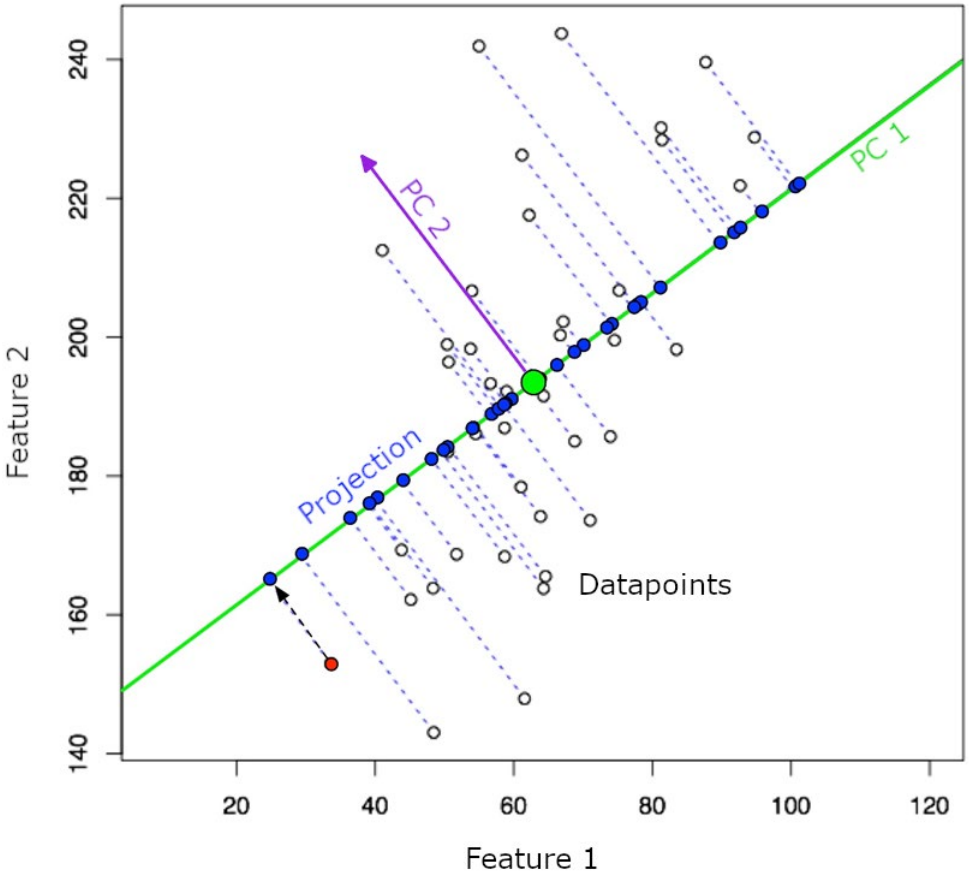


*Fig. 11: PCA demonstration; vertical and horizontal axes represent two features, and each point is a separate input.*

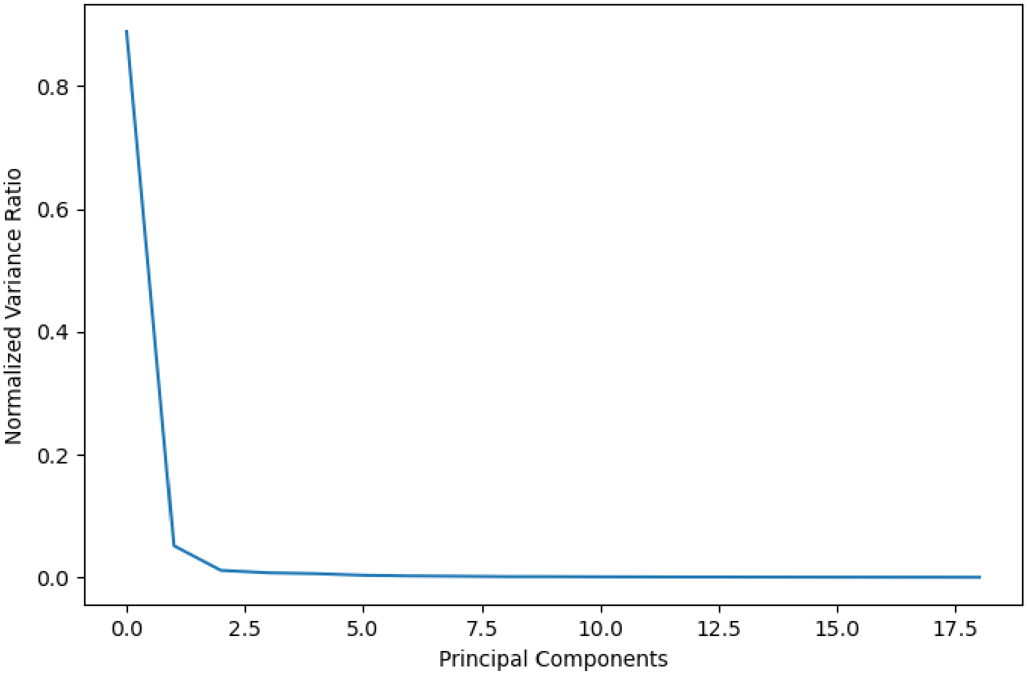


*Fig. 12: Variance (predictive ability of a PC) vs. PC number in a sorted list of PCs.*

PCA operates by computing the eigenvectors of the covariance matrix of training data. If we have a matrix $\boldsymbol{W}$ for which each row $\boldsymbol{w}$ is a window, and each feature is a column of this matrix, then we average each row to form the mean vector $\overline{\boldsymbol{w}}$. We construct the mean matrix $\overline{\boldsymbol{W}} = \begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix}\overline{\boldsymbol{w}}$, subtract means to center each feature, $\boldsymbol{B} = \boldsymbol{W} - \overline{\boldsymbol{W}}$, and compute the covariance matrix $\boldsymbol{C} = \boldsymbol{B}^T\boldsymbol{B}$. Each eigenvector of the covariance matrix $\boldsymbol{v}_i$ is determined by maximizing the eigenvalue respective to $\boldsymbol{v}_i^T\boldsymbol{C}\boldsymbol{v}_i$, and the eigenvectors $\boldsymbol{v}_i \in \boldsymbol{V}$ form the basis of our principal component space. The rows of $\boldsymbol{W}$ are projected onto $\boldsymbol{V}$ to map features to principal components. In practice, more efficient matrix factorization methods are employed.

We also plan to test Uniform Manifold Approximation and Projection (UMAP) as an alternative dimension reduction algorithm. Much as principal components are linear combinations of features, UMAP components are non-linear functions of features. UMAP is shown to provide better class-clustering in feature space and better computational performance on large datasets.

## 1.6 Machine/Deep Learning Models

When we are satisfied with the dimensionality of our features, we begin training on windows of biopsy slides. Machine learning diverges from traditional computer programs in that the computer program is learned from available data. There are three essential components to machine learning: the data, the mathematical model, and the training algorithm. [6] We've already talked about data, so let's clarify what we mean by *model* and *training*.

The model is a mathematical representation of different relationships inherent to the training data. Data is fed to the model and variables of the model are continuously adjusted so its output agrees with its input. In a broad sense, the model may be thought of as a control system. When we train a model, we assess the error between input and output. In machine learning, error is evaluated by means of a *loss function*. A loss function may be as simple as the sum of squared errors, or as involved as cross-entropy.

All machine learning output is, in some sense, an average of training data. Polynomial regression fits a polynomial to data trends, neural networks superimpose activation functions, decision trees divide the feature space into averages of constituent points, and k-nearest-neighbors compute averages directly for points nearest to the input. The model chosen for any particular application should reflect the behavior of training data. For this reason, we survey a number of models and assess their ability to predict trends and converge in reasonable amounts of time.

### 1.6.1 Random Forest (RNF)

One of the most commonly used machine learning models is Random Forest (RNF), also known as Random Decision Trees. This model is a supervised ensemble learning model, made up of many weaker models, or Decision Trees, that collaboratively creates a stronger model and trains on labeled data. The structure of an RNF model can be seen in Fig. 13. Overall, it resembles a forest that has *x* number of trees. Starting from one point, it splits to multiple tree-like diagrams, and each tree will continue to branch out, all reaching a unique endpoint. The first part of an RNF model is the sample, or the *root node*. This root node is the very first node at the top of the forest, which starts with a dataset – the set of features generated and extracted from one of the previously mentioned feature extraction processes.

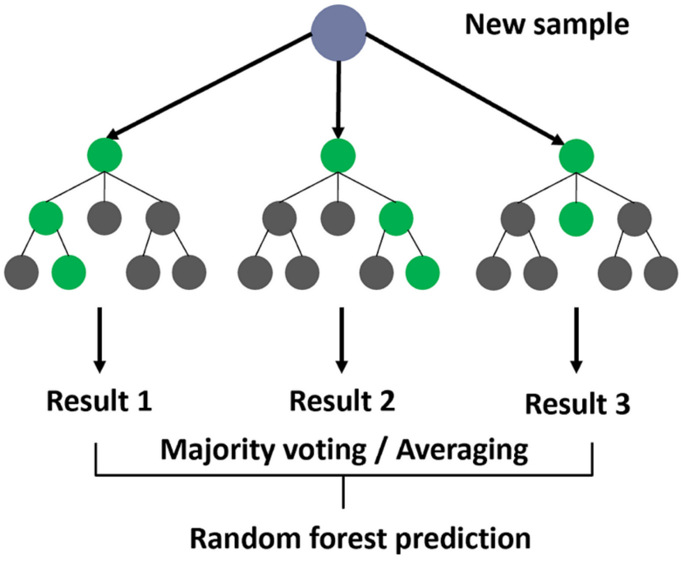


*Fig. 13: Random Forest structure [7].*

#### *Bootstrapping*

Because the sample is limited in quantity, RNF implements a method called *bootstrapping* to quantize the sample. Bootstrapping takes the sample and resamples it over and over with subsets of the sample. The subsets are random parts of the original data with replacement and equal distribution. In other words, some subsets may contain multiple of the same feature, and each of the features have an equal probability of getting selected. [8] Fig. 14 below is an example of bootstrapping, or resampling with replacement. Because the original sample on the left has only a few datapoints, there is a limited number of combinations that can be made when sampling. In this case, there are three blue, two orange, two green, and two yellow data points. However, when resampling with replacement, more combinations can be made since datapoints are reused. In Sample 1, it reuses a yellow datapoint, and in Sample 3, it reuses blue datapoints and does not use the green datapoints. In general, bootstrapping gives a model more samples to train on.

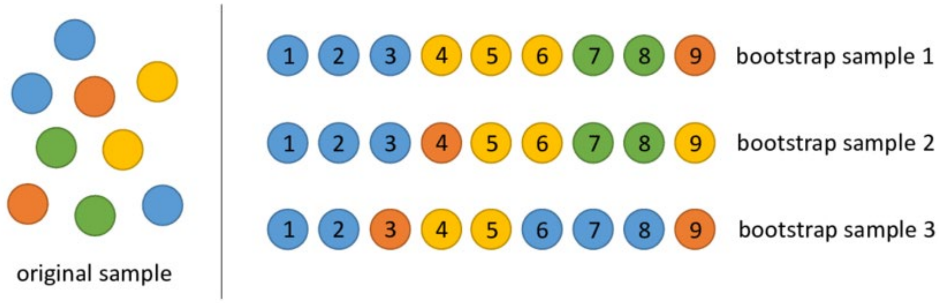

original sample
1 2 3 4 5 6 7 8 9 bootstrap sample 1
1 2 3 4 5 6 7 8 9 bootstrap sample 2
1 2 3 4 5 6 7 8 9 bootstrap sample 3

*Fig. 14: Bootstrapping example [9].*

For each time the sample is split into random subsets, each set marks the beginning of a decision tree stemming from the root node. Based on the features, each decision tree will branch out to some sort of possibility, also called a *decision node*, and it will continue doing so until each tree reaches a point that it cannot make any more decisions. This stage is called a *prediction*. In short, this model builds decisions off decisions recursively until all trees reach a prediction.

### *Bagging (Bootstrap Aggregation)*

Because each tree is handling random subsets of the overall sample data, there will be multiple different predictions. A quarter of the predictions may want to classify a frame as INFL, while the rest of the predictions think it may be DCIS. RNF deals with different weights of prediction using the *bagging* method, also called bagging aggregation. Just as the name implies, after the bootstrapping method, all the predictions are aggregated, and the final prediction is chosen based on a majority vote of all the predictions. This final prediction is the classification of the frame.

An example of bagging is shown below in Fig. 15. If our model is trained to classify different fruits, the sample would be features of a specific fruit. The features would undergo the bootstrapping methods, so the model has many different combinations to train on. Eventually, each tree will produce a prediction: Tree 1 and Tree 2 will predict the fruit is an apple, and Tree 3 will predict it is a banana. The bagging method will combine all the predictions. In this case, two out of three predictions are apples, so by majority vote, the fruit in question is classified as an apple.

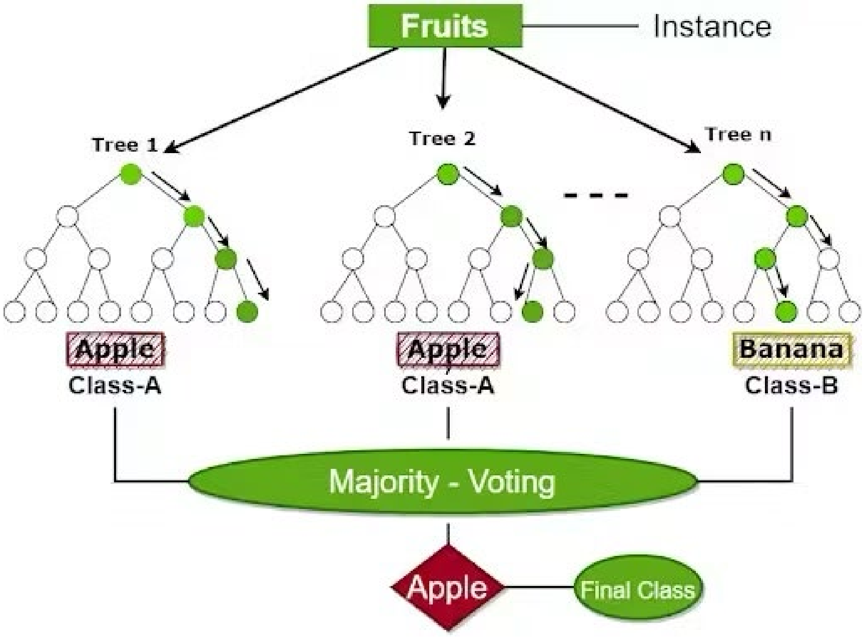


*Fig. 15: Random Forest example [10].*

RNF models have their own set of benefits that set them apart from other Machine Learning models. If the data were trained only on a decision tree model, the tree would keep branching, with each split eventually becoming too specific to the original training data. This is called overfitting. Since RNF is one big model combined with many decision trees, there would be multiple predictions, resulting in a smaller chance of overfitting and lower variance. However, depending on how large the dataset is, this may lead to a more complex algorithm with a longer runtime, as the data for each decision tree is computed. [11]

#### *How was RNF used in our project so far?*

There are three main hyperparameters: the node size, number of trees, and the number of features sampled. Using the Python's SKLearn library, it already has a random forest function built-in. There are many modifiable parameters, however, for our project, we have not set any parameters and hyperparameters. At the moment, we have only input our data into the RNF model to see if it works. By not inserting any parameters, the parameters may have been the default settings. Of course, the results were not what we wanted, but it gave us a baseline to know where we should start and what we should look into tuning.

### 1.6.2 Convolutional Neural Networks

Unlike Random Forest, Convolutional Neural Networks (CNN) is a deep learning model that is an extended version of the machine learning model Artificial Neural Networks (ANN). ANN is a model that holds units called neurons. These neurons receive an input, does some sort of computation, and provide an output, and those outputs will become inputs to the next outputs. Just like a brain, these neurons are all intertwined, forming a complex structure that provides highly accurate classifications. However, as ANN is used for a wide range of tasks, not specifically dedicated in one area, CNN builds off ANN and specializes in image recognition and classification. CNN has three types of layers: input layer, hidden layer, and output layer.

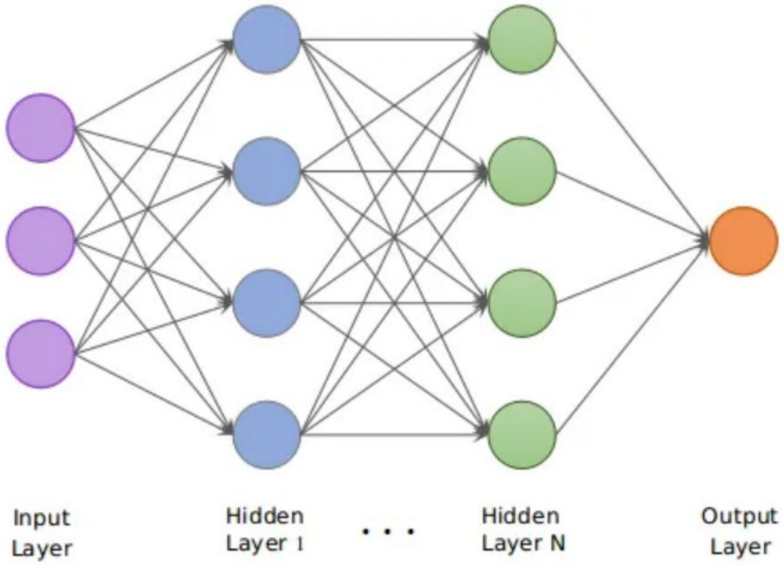


*Fig. 16: CNN structure [12].*

### *Input Layer*

The input layer takes one image at a time. Usually, images have three channels: red, green, and blue, which is why in Fig.15, the input layer has three dimensions. These refer to the channels. In our case, our training data consists of images in RGBA format, which means there will be four dimensions in the input: red, green, blue, and alpha. In addition, the images are 50k×50k pixels. Since they are quite large, rather than the whole image, the input will be provided with windows of the images containing only the extracted features.

### *Hidden Layer*

The hidden layer has three main types of layers: a convolutional layer, pooling layer, and dense layer. At the end of a convolutional layer, a very important step before moving onto the pooling layer is applying the activation function.

### *Convolutional Layer*

The convolutional layer takes the features from the input layer and performs linear transformations, specifically convolution, using a filter – a matrix with weights. Since the image has three dimensions (the last dimension depth of 3 channels: RGB), the filter must also have three dimensions with a depth of 3. A typical filter would usually have a height and width of three 3 pixels, while the depth is the same as the image. It would resemble Fig. 17, where *c* is the depth (i.e., number of channels). The height and width of the filter must be significantly smaller compared to the input as there would be too many weights resulting in too many connections if the filter was the same size.

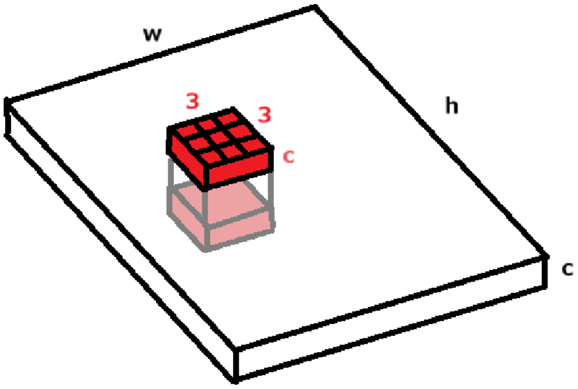
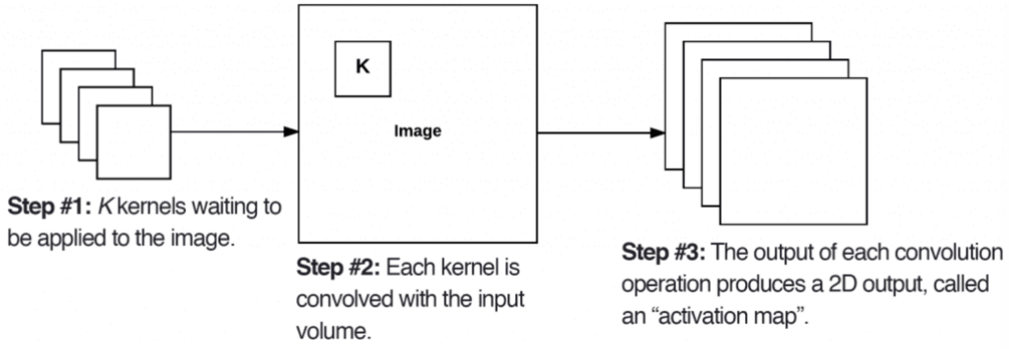


*Fig. 17: Filter over image with depth 'c'.*

The filter will start by positioning itself to the top-left of the image, and the dot product of the filter and the features laying underneath will be calculated. The filter will move in strides (which can be adjusted) and iterates through each row of the image. Once the whole image is swept through, it will output a feature map, a matrix containing all of the scalars from the dot product computations. [13] There can be multiple filters in a convolutional layer, which will result in multiple feature maps. Multiple filters are used to extract different features (e.g., edges, textures, color), and each of those filters will work on the same input image. In Fig. 18, you can see that the number of feature maps outputted is the same as the number of filters that run through the image. It should be noted that *filters* and *kernels* are interchangeable, and *feature maps* and *activation maps* are interchangeable.



**Step #1:** *K* kernels waiting to be applied to the image.

**Step #2:** Each kernel is convolved with the input volume.

**Step #3:** The output of each convolution operation produces a 2D output, called an “activation map”.

*Fig. 18: Usage of multiple filters in the convolutional layer [14].*

The feature maps are all 2-dimensional, but when stacked together in the depth axis, it creates a 3-dimensional output, as shown in Fig. 19.

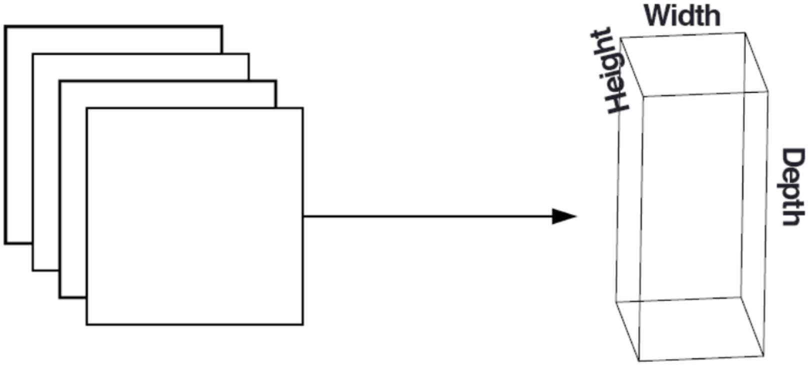


*Fig. 19: Stacking feature maps [14].*

In addition, there can be multiple convolutional layers, where the next layer takes the feature map of the previous layer as the input. Using multiple layers helps improve efficiency and accuracy since the layers build off one after another, extracting more complex features from the previous features.

### *Activation Function*

The activation function is a nonlinear function that is applied to each feature map of every convolutional layer. The activation function is an important step of the CNN model because it produces only the meaningful parts of the data. One of the most common activation functions is the rectified linear unit (ReLU). Fig. 20 below shows a plot of the ReLU function. As you can see, when $x$ is negative, $y$ is equal to 0, and when $x$ is positive, $y$ is equal to $x$. When this function is applied to the input, the negative elements of the matrix are zeroed while the positive elements are kept the same. In other words, It turns off the negative elements, and only keeps the elements that seem important. This allows the model to learn complex relationships with the input and output. Afterwards, the resulting matrix is sent as the input to the next convolutional layer or the pooling layer if there are no more convolutional steps.

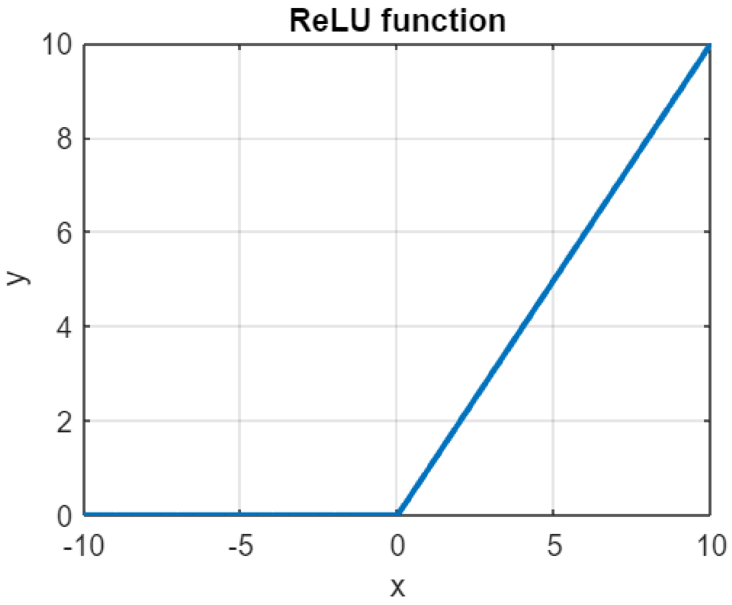


*Fig. 20: ReLU activation function.*

### *1.6.2.1.1.1 Pooling Layer*

After the last convolutional layer, the pooling layer takes the feature maps and downsamples, or compresses, them using a filter and send it to the output array. When compressing the feature map, it summarizes the features by combining multiple features throughout the image, which removes a lot of the fluctuation between them. This helps the model to not “overreact” and misclassify the frame. The filter must be smaller than the feature maps. If the feature map is 250 × 250 pixels and the filter is 2×2 pixels, for every 2×2 position of the feature map, it would compress those values into a single value using max pooling or average pooling. [15] The size of the pooled feature map depends on the stride. In Fig. 21, it shows an example of max pooling the input matrix with a stride of 1 and 2. Higher strides lead to smaller matrices as it sweeps through the input faster, as well as downsampling more, while lower strides lead to bigger matrices, retaining a bit more information. This is also the same for the filter size too. When picking the filter size and stride, there will be chances of oversampling or undersampling, so a middle ground has to be considered.

# CONVOLUTIONAL NEURAL NETWORKS (CNNS) AND LAYER TYPES

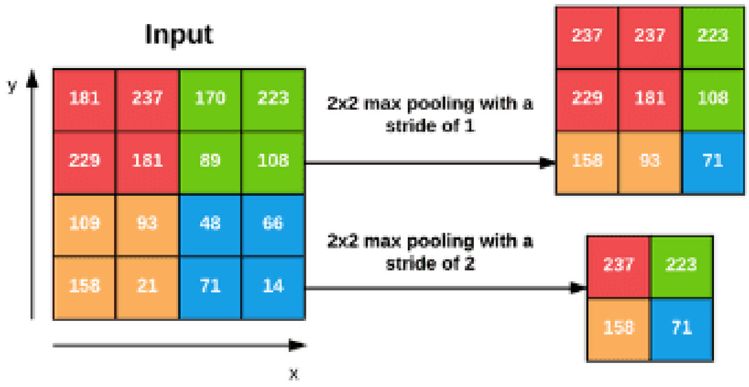


Fig. 21: CNN max pooling example [14].

### *1.6.2.1.1.2 Dense (Fully-connected) Layer*

Before reaching the dense layer, the feature maps received from the pooling layers for each hidden layer must be condensed into one-dimensional vectors and then fed into the dense layer, specifically, each neuron of the dense layer. Each neuron will do computations to the input vector, which helps the neurons connect to each element of all of the feature maps. These computations consist of a weighted sum plus a bias term as Eqn. 7 below.

$$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

Equation 7: Weighted sum plus bias.

This layer also makes use of activation functions as it would be applied to the output of the above equation. Each neuron in the layer would produce a scalar. The scalars would be placed into a vector with size *m*, where *m* is the number of neurons in the layer. An activation function would be applied to this vector (Eqn. 8).

$$a = [a_1, a_2, a_3, \dots, a_m]$$

Equation 8: Vector of computed scalars from each dense layer neuron.

As there can be multiple convolutional, there can also be multiple dense layers. The output vector of the first dense layer would be fed into the second dense layer. Lastly, the outputs of the last dense layer would be sent to the outer layer.

### *Outer Layer*

Just like the hidden layer, the outer layer may also contain a dense layer. However, this dense layer would finalize everything and output a score. Using the same process as above (computing the weighted sum plus a bias), we may use a different type of activation function. There are multiple options but the most common activation functions that produce a score is the sigmoid function and softmax function. If we plan to classify frames by checking for each classification type individually (e.g., "Is this NULL type?", "No." "Is this DCIS type?", "Yes."), then it's better to use sigmoid as it classifies only with two symbols: 0 (no) and 1 (yes). The second option is classifying frames by checking all of the classification types in once process. In this case, it would be better to use the softmax function since it can use multiple symbols: 0 to 9 (e.g., 0: Unlab, 1: BCKG, 2: NULL, etc.) If there is a higher probability of a certain class, then the output would produce the score of the probability and the classification.

Using this model over Random Forest would provide many advantages. Since this model specializes in image recognition and classification, this would be more ideal to use as it provides more accurate classifications and is more efficient for what our project is intended to do. It also handles noise and variation much better since the input would go through many pooling layers and activation functions for combining and downsampling. However, CNN models are more difficult to train as they are very complex and require a lot of computational resources. In addition, the outputs of the overall model or individual layers may be difficult to interpret, that it would take some time figuring out how to finetune the model. [16]

# 2 Engineering Design Plan

## 2.1 Application of the Flood-Fill Algorithm and WSI Classification

Consider Fig. 22 and suppose the frames highlighted in green were predicted by the model to contain ductal carcinoma in situ (DCIS).

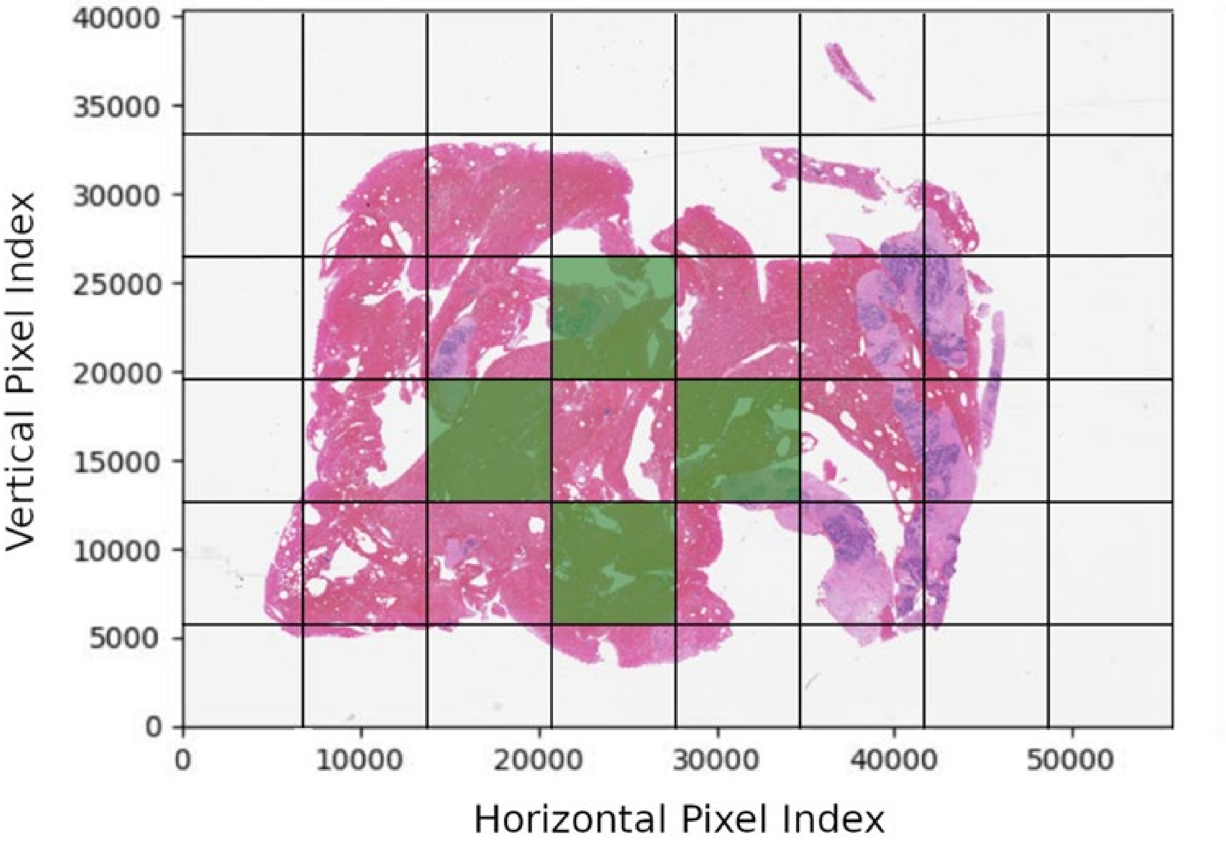


*Fig. 22: Sample model output of Fig. 3*

If a tumor has extended to the region in green, then naturally the interior must be malignant as well. We identify and reclassify the interior frame with the appropriate DCIS class by means of a *flood-fill* algorithm.

Flood-fill algorithms are commonly used in graphics editors and more colloquially known as 'paint-bucket' tools. The purpose of a flood-fill algorithm is to fill an area of an image or array with some value (be it a color or label) while respecting the boundaries of existing values. Flood fill may be used to locate interior regions of DCIS predictions in the following manner. First, we apply flood fill to the entire biopsy slide starting at a boundary frame (e.g., the top-left frame in Fig. 23).

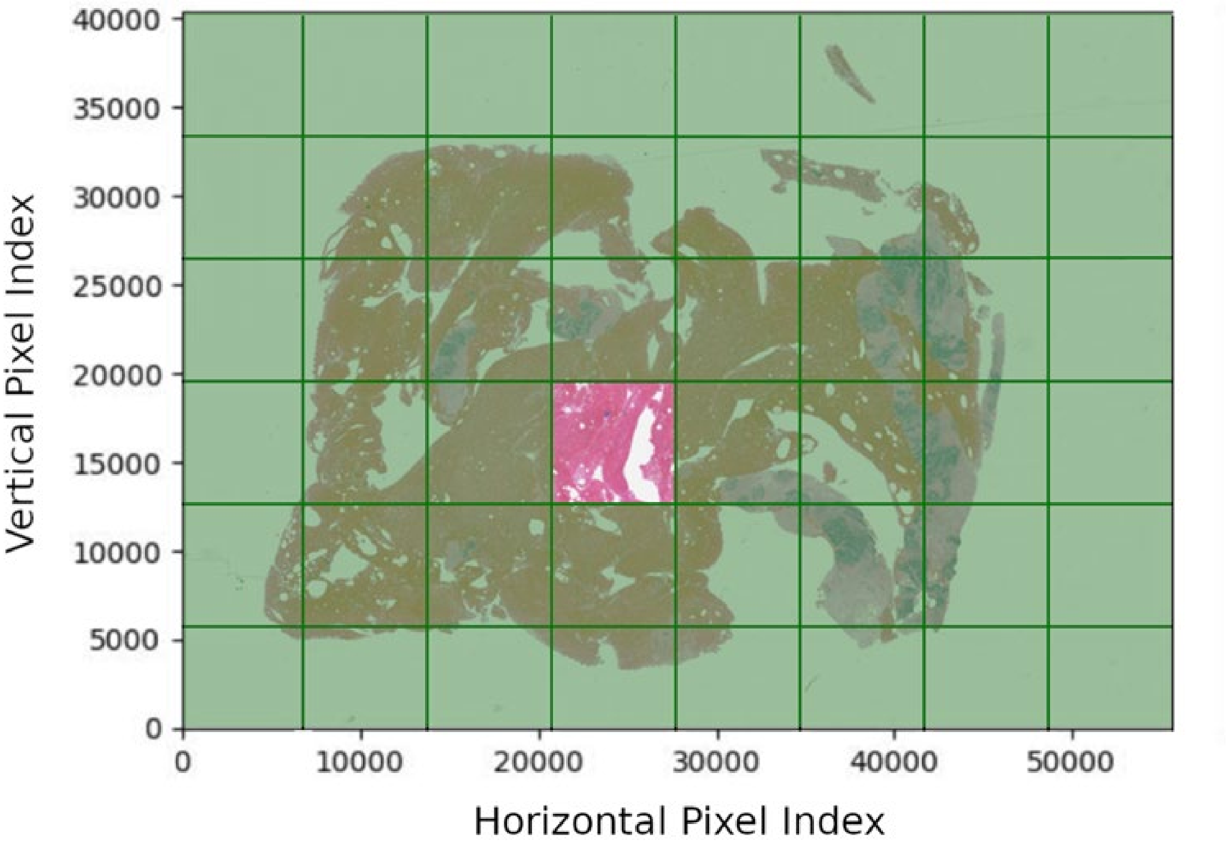


*Fig. 23: Flood fill applied to Fig. 22: Sample model output of*

We invert DCIS predictions such that non-DCIS frames are classified DCIS and vice versa (Fig. 24). Finally, we apply the inverted array in Fig. 24 as a mask to the generating array in Fig. 21. The result is an array of predictions with interior regions of DCIS classifications consistent with adjacent frames (Fig. 25).

To classify whole-slide images, we apply the label with the highest urgency according to Tab. 1. If a slide image contains DCIS, the entire image is classified as DCIS. However, if only non-neoplastic, inflamed, and normal tissue were identified in a biopsy slides' frame classifications, then the slide image would be classified with the highest urgency label (INFL).

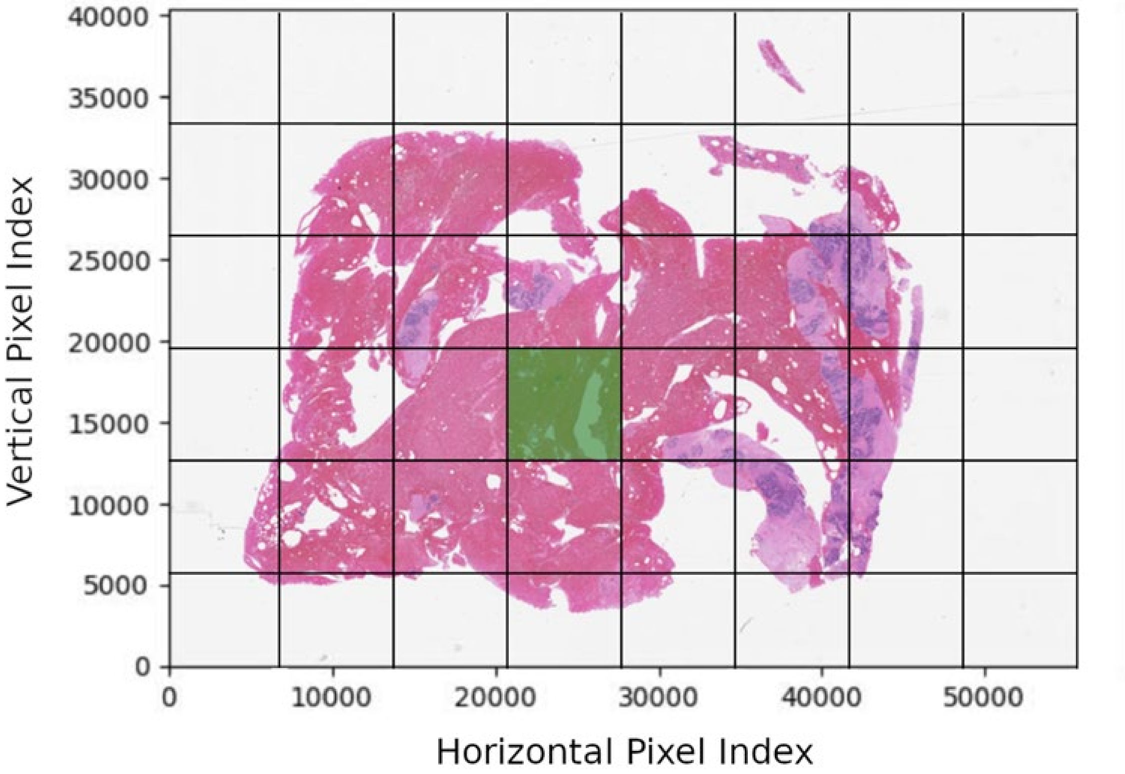


*Fig. 24: Inverted flood-fill of Fig. 23.*

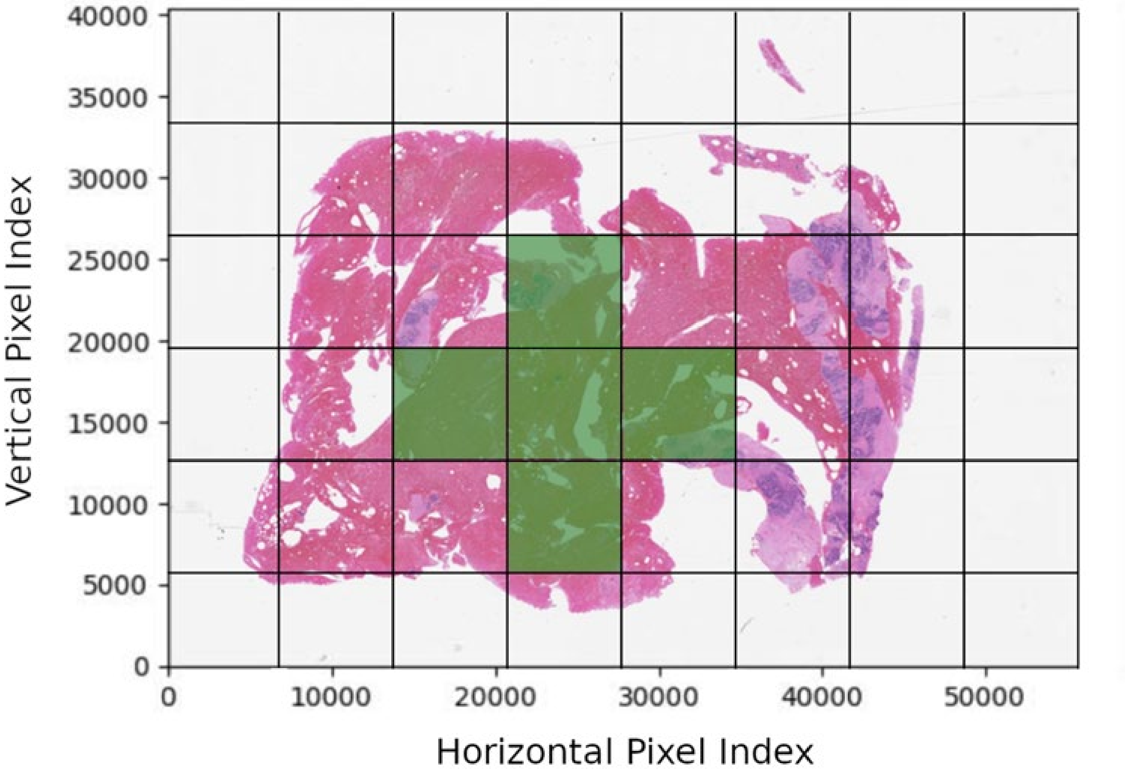


*Fig. 25: Original DCIS predictions with mask (Fig. 24) applied.*

## 2.2 Train, Dev, and Eval

To train a model, we must first initially train the model on a subset of the data. Afterwards, we refine parameters by utilizing a separate subset of the data. Finally, we take this trained and refined model and evaluate it on a third separate subset of the data. The reason we do this is to reduce overfitting. Overfitting happens when we put too much weight on attributes of the data that couldn't be generalized from the size of the available data. For example, if our first subset of the data has a particular feature that is very pronounced in the artifact label, the model is going to attribute that feature to recognizing the artifact labels. However, if this feature doesn't normally exist within the artifact label, we hope that when we compare with a second subset of the data, the model will pick up that this isn't a feature that is normally present in the artifact label. Finally, we utilize a third subset of the data to validate that our model hadn't overfitted to either of the previous two subsets of the data. A graphical representation of this can be shown below in Fig. 26 . We call the first, second, and third subsets of the data Train, Dev, and Eval respectively. [17]

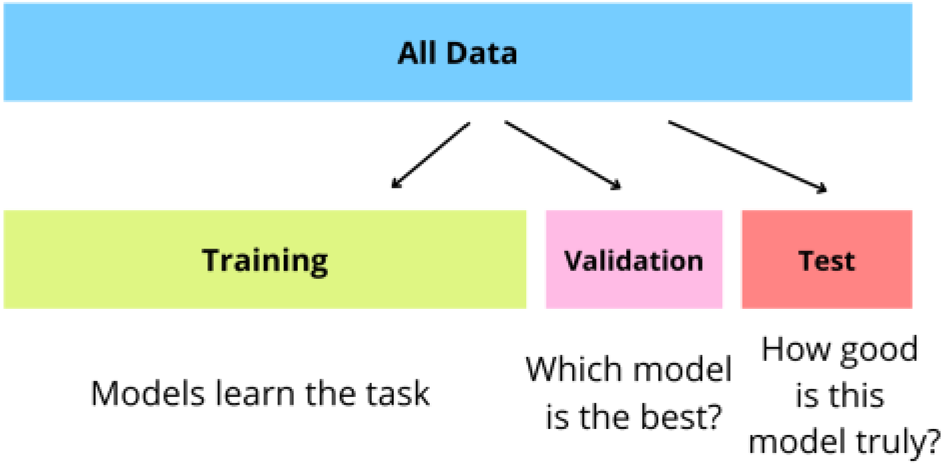


*Fig. 26: Example of Train, Dev, Eval Split*

To have this method be effective, we need to have both an even distribution of the labels through all the datasets, and a proper ratio of data in the Train, Dev, and Eval set. It's standard to have 80% of the data in the Train set, 10% in the Dev set, and 10% in the Eval set. However, as seen in Tab. 2 below, we have a slightly different ratio of around 60% for Train, 20% for Dev, and 20% for Eval. The reason this is slightly different from what is standard is due to the imbalance of labels, such as inflammation (INFL), that only make up a small amount of the total labelled regions. It's possible that if there was an 80:10:10 split followed, there wouldn't have been an even distribution of such labels.

*Tab. 2: Train, Dev, and Eval Label Distribution [18]*

| Label | Train | Dev | Eval | Total |
|---|---|---|---|---|
| **artf** | 17,147 | 6,513 | 6,881 | 30,541 |
| **bckg** | 329,404 | 110,425 | 110,599 | 550,428 |
| **dcis** | 5,626 | 1,945 | 1,900 | 9,471 |
| **indc** | 6,574 | 2,528 | 2,599 | 11,701 |
| **infl** | 1,144 | 473 | 457 | 2,074 |
| **nneo** | 15,183 | 5,684 | 5,770 | 26,637 |
| **norm** | 4,524 | 1,755 | 1,745 | 8,024 |
| **susp** | 15,445 | 5,768 | 5,607 | 26,820 |

We have now established well defined Train, Dev, and Eval datasets. We then need to do two things to each image within the data. We need to segment it as shown in Fig. 4 and perform some preprocessing as shown in Fig. 6. However, for some use cases that need the full context of the entire image before performing something like a 2D DCT. What this means is that sometimes the preprocessing will be done before the segmentation, and sometimes it will be done after. The combinations of these are shown in Fig. 11 & Fig. 9.

After the image has been segmented and undergone some pre-processing, we now have our features. However, this means that we could be sending window_size$^2$ floats for each window. This is too much for the model to handle and for it to analyze each of those numbers is something that we don't have time for especially when we can perform feature reduction. Doing something like PCA as seen in Fig. 11 allows us to reduce the amount of information the model has to process, and as a result, the amount of time the model takes to train.

Once we have established a dataset of features, we need to take the train set and fit the model. This involves feeding the data into a CNN as mentioned above. Once we have fit the model to the train set, we are then going to have the model make predictions on the development set. Using these predictions, we can attempt to optimize the model and remove some symptoms of overfitting inherited from the train set. This process can be seen in Fig. 1. Finally, we validate our model on the eval set. Once we are happy with our results, this leaves us with a trained and refined model.

With a finished model, we can effectively generate predictions for individual frames. However, we need to bring these individual predicted frames together to make a predicted region, like the annotations made in Fig. 30. This amounts to an inverse operation of the slide segmentation algorithm described in section 1.2. By aggregating frame predictions in a slide, we can form a prediction for the slide as a whole and ensure consistency among frame predictions in regions of a slide.

## 2.3 Prediction Report & Example Use Case

Having a trained, refined, and tested model allows us to start making predictions in a production environment. In order, to make our predictions useful to end users unfamiliar with our system, we must have a Graphical User Interface (GUI). This is essentially a slightly higher-level view of the results. This includes an overlay of the predicted regions over the image of the tissue fed in, the type of tissue of these regions, and an overall report of cancer for the image. This can be seen below in Fig. 27.

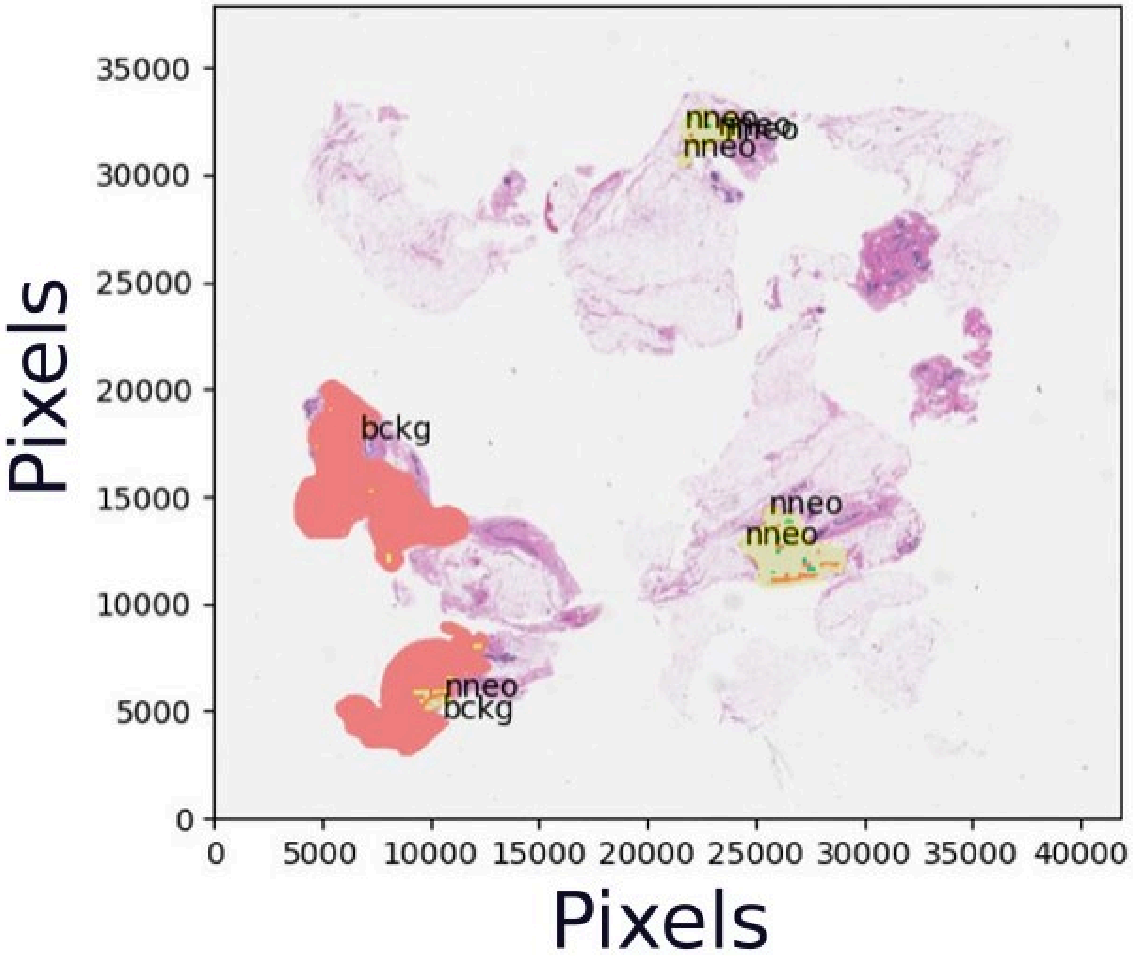


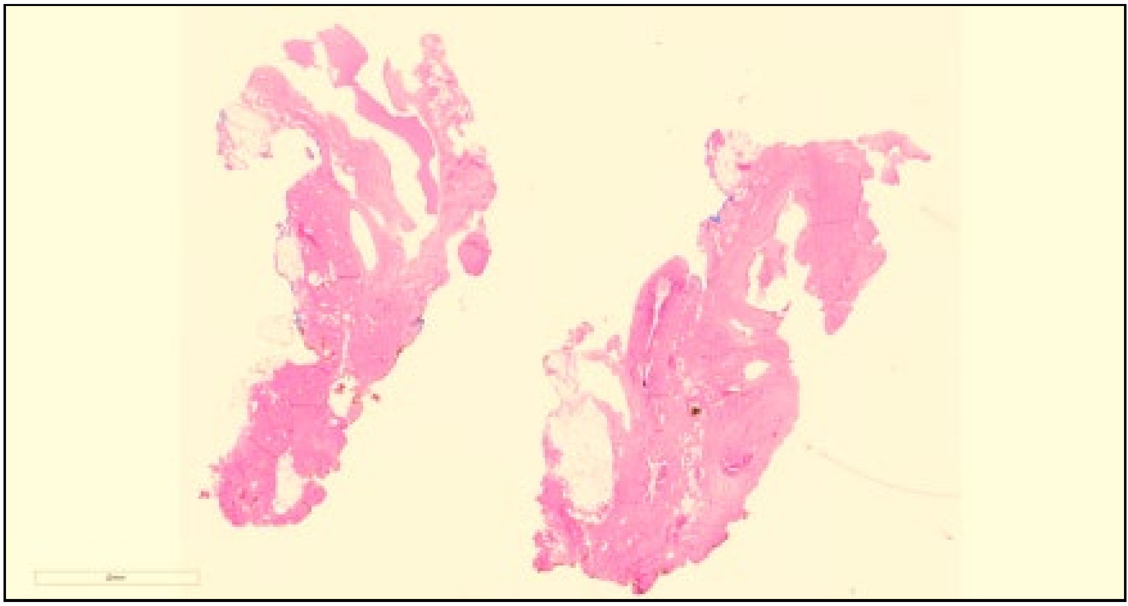
*Fig. 27: Example graphical representation*

One use case for these reports is to be used in an urgency system. If the volume of biopsied tissue slides that need to be reviewed is greater than the amount of work that pathologists can do, it’s possible for there to be a buildup of these images. A hospital may be able to utilize the predictions to organize the quote of images, so the highest risk patients are evaluated first. Due to the inherent aggressiveness of breast cancer, evaluating these higher risk patients can reduce their time to treatment and therefore their overall risk factor. Once a pathologist gets the chance to evaluate an image processed by the model, the pathologist can utilize the prediction report shown above in Fig. 27 to validate their diagnosis.

## 2.4 Temple University Health Digital Pathology (TUHDP) Corpus

To train a model, we must start with data. Temple University has a large open-source dataset being developed for a variety of diverse types of tissue. The subset of data we are focusing on is a 3,505 images (1.23 terabyte) collection of breast tissue. For each tissue image, there is an associated annotations file that contains the following information: microns per pixel, height, width, and labeled regions. [18]

Microns per pixel is a floating-point number that represents a mapping. This is calculated by taking the size of the object being scanned and mapping it to the resolution of the image being stored. This allows us to create a mapping of each pixel of the image to an actual physical representation of what was being scanned originally. This is vital to the success of precise image analysis for machine learning. It will eventually allow us to normalize the images, allowing us to analyze various images taken with different from various devices. [19] As you can see below in



, knowing the microns per pixel will allow us to understand what kind of information and how much we are getting.

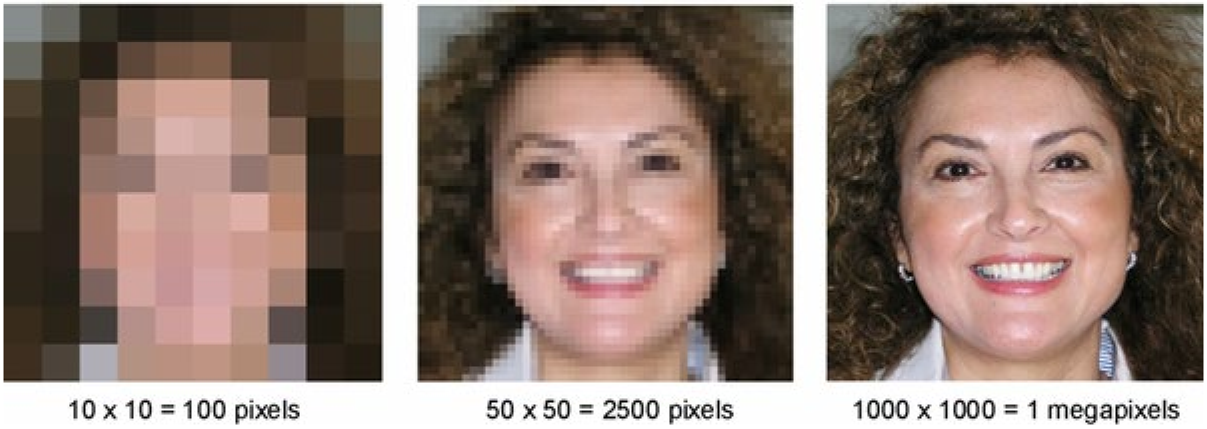


*Fig. 28: Effects of Microns Per Pixel [20]*

The height and width of the image are measured in pixels and allow us to segment the image for consistent analysis. We plan to separate the image into many overlapping windows. The reason for this overlap is that it may allow us to avoid noise when processing individual frames. It is a commonly known phenomenon that by having an overlap of at least 50% when analyzing a signal, it is possible to reduce random noise due to the included context of the previous frame. It is analogous to teaching a guitar player to play the mandolin to eventually teach them the violin. If you gave a guitar player the violin directly, they would have had to learn the new note layout of the strings and the new physical techniques of how to hold the bow and place their fingers at the same time. However, since the mandolin features a similar playing technique to the guitar and the same note layout of the strings as the violin, the guitar player would have an intermediate step to understand the context of the violin. As seen in Fig. 29 below, this is the kind of smoothing we are hoping for. Opening the analysis up in this way allows for less signal noise. [21]  Without testing, we are not sure if this will provide benefit to our processing, but we will perform rigorous tests to see what level of overlap works best for our use case.

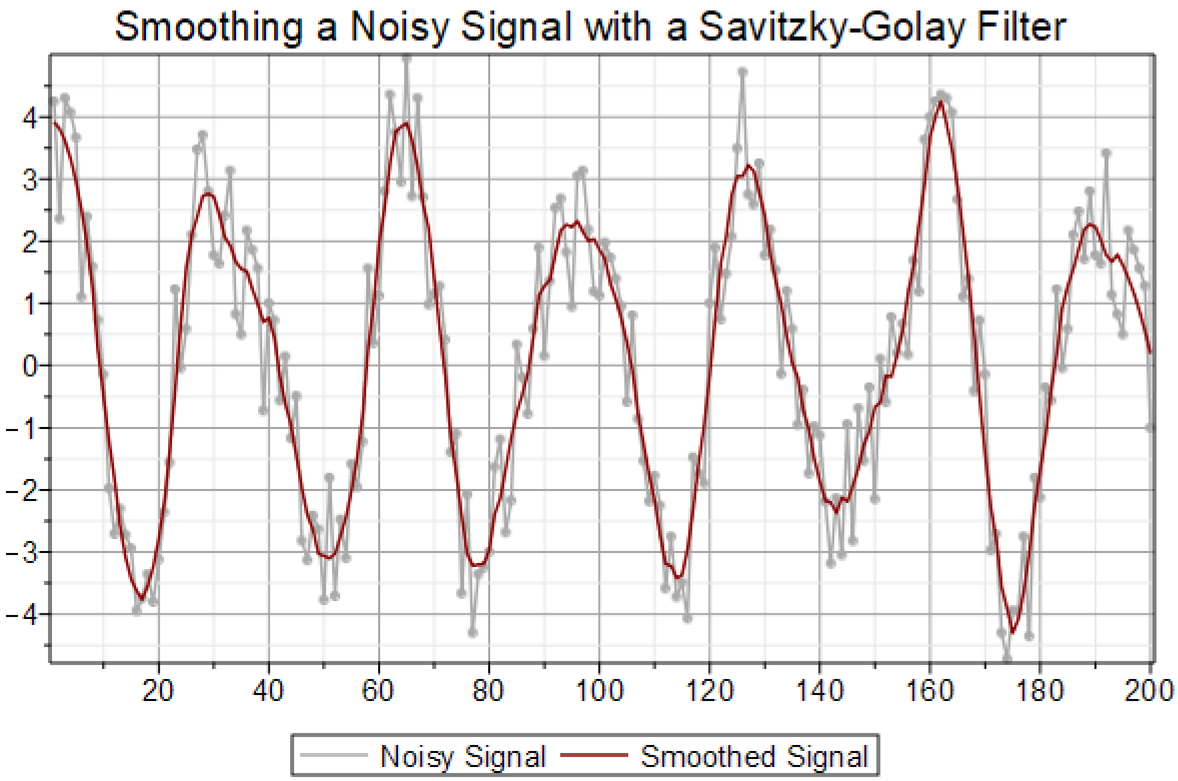


*Fig. 29: Example of Signal Smoothing [22]*

Finally, we need to move onto the labeled regions. Firstly, it's important to understand what regions are in this context. A region is simply an outlined portion of the image identified by a list of coordinates and associated with a label. See Fig. 30 below for an example of what an actual region outlined on an image of breast tissue may look like. The two dark textboxes are labels. These are abbreviated version of tissue types shown below in Fig. 30 taken directly from a TUHDP TUBR abstract of the previous model trained to identify these regions.

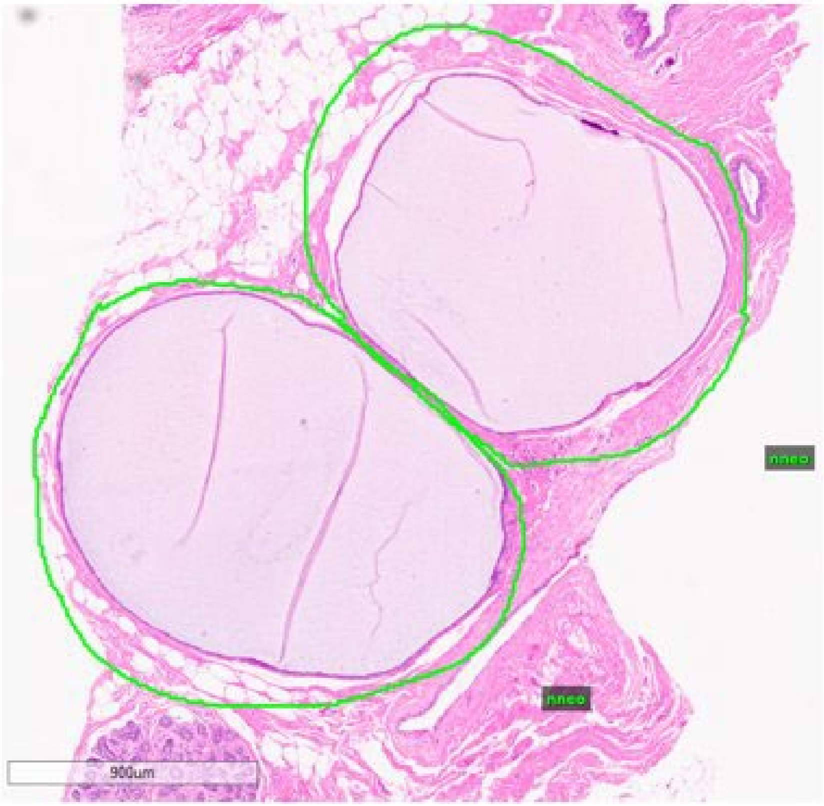


*Fig. 30: Example of Outlined Regions [23]*

# 3 Engineering Design Immediate Goals

## 3.1 One to Two-Dimensional DCT

Currently, our system is extracting all RGB values from the image into separate vectors containing each color. This essentially leaves us with 3 vectors of length (window height [pixels] * window width [pixels) containing integers between 0 and 255. On each vector, we then perform a one-dimensional DCT and keep an arbitrary number of the most significant frequencies. This essentially performs an analysis on the color spectrum for the window and ignores the rest of the context in the window. Moving forward, we want to perform a two-dimensional DCT as mentioned above in section 1.3.2.

## 3.2 Implementing PCA

As mentioned in section 3.1, we are arbitrarily selecting a frequency cutoff. The reason we implement this is due to speed and memory of the machine, as well as the optimization of the algorithm to train the model. By keeping only the most significant frequencies of the color spectrum, we are able to keep a good portion of the information without taking up very much memory and reducing compute time.  However, since we are selecting feature extraction arbitrarily, this method will fail to two things. One, It will lose the position. The only way to do this on a matrix would be to randomly pick sections of the frame, so position wouldn't be kept, and we would lose continuity. Secondly, as it is doing now, we are failing to consider what is worth keeping and what isn't worth keeping. To maximize efficacy, we need to keep the maximize amount of information while keeping a reasonable compute time. One way to do this is using PCA as mentioned in section 1.5.

## 3.3 Implementing PyTorch CNN

Our system is currently utilizing a Python library Sci-kit Learn for establishing a base model as mentioned in section 1.6.1. This is a Machine Learning model, but we want to add an option for deep learning to explore the affects it has on making predictions. In order to do this, we are going to be utilizing a Python library PyTorch which allows us to more easily create a pipeline for deep learning models such as a CNN. While the detailed explanation of how to use a CNN for PyTorch is outside the scope of this paper, the first step is to take our processed frames and feed them into something called a DataSet. This allows us to increase the maximum training set size as it allows us to train the model off disk instead of from memory as was done with the Random Forest implementation. After we load the data into a DataSet, we explicitly declare some of the parameters talked about in section 1.6.2. Explicitly setting some of these parameters such as the types of layers, number of layers, epochs, etcetera. Allow us to significantly manipulate how the model trains, and as a result the model's precision and accuracy on general data.

## 3.4 Implementing GUIS

Now that we have a functional pipeline that takes from an image to fully formed regions, we want to be able to showcase something like what is shown in Fig. 27. Furthermore, we want a person unfamiliar with our system to be able to enter an image. To do this, we must implement a sort of drag-and-drop image functionality so the user will be able to drop an image in and get the showcased image and some information back as shown below in Fig. 31.

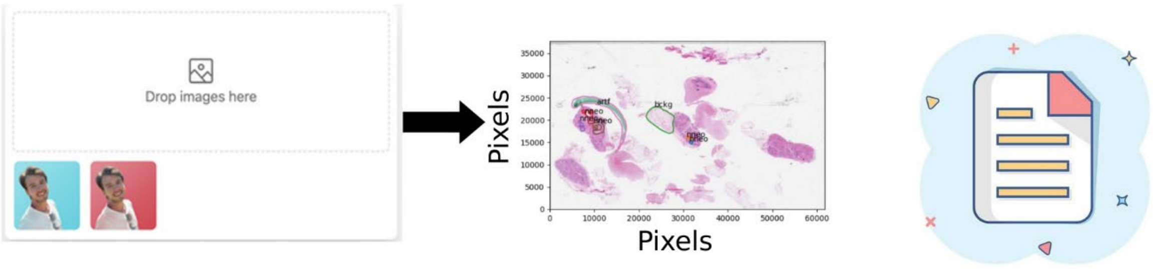


*Fig. 31: Example of GUI pipeline*

# 4 References

[1] "The ABC of Machine Learning," aiplanet.com, [Online]. Available: https://aiplanet.com/learn/5-week_Data_Science_Bootcamp/week-2/162/the-abc-of-machine-learning. [Accessed 1 October 2024].

[2] Y. K. Abdullah-Al Nahid, "Histopathological Breast-Image Classification Using Local and Frequency Domains by Convolutional Neural Network," *Information-Centered Healthcare,* 2017.

[3] R. W. S. Alan V. Oppenheim, "The Discrete Cosine Transform," in *Discrete-Time Signal Processing*, Pearson, 2010, pp. 680-681.

[4] Yumi, "Yumi's Blog," 19 July 2019. [Online]. Available: https://fairyonice.github.io/1D-DCT-vs-2D-DCT.html. [Accessed 4 October 2024].

[5] N. Matloff, The Art of Machine Learning: A Hands-On Guide to Machine Learning with R, No Starch Press, 2024.

[6] N. W. F. L. T. B. S. Andreas Lindholm, Machine Learning A First Course for Engineers and Scientists, Cambridge University Press, 2022.

[7] R. Yehoshua, "Random Forests," 24 March 2023. [Online]. Available: https://medium.com/@roiyeho/random-forests-98892261dc49.

[8] J. Frost, "Introduction to Bootstrapping in Statistics with an Example," [Online]. Available: https://statisticsbyjim.com/hypothesis-testing/bootstrapping/.

[9] P. Galdi and R. Tagliaferri, "Data Mining: Accuracy and Error Measures for Classification and Prediction," in *Reference Module in Life Sciences*, Elsevier, 2018.

[10] M. Chaudhary, "Random Forest Algorithm - How It Works & Why It's So Effective," [Online]. Available: https://www.turing.com/kb/random-forest-algorithm.

[11] IBM, "What is random forest?," [Online]. Available: https://www.ibm.com/topics/random-forest#:~:text=Random%20forest%20is%20a%20commonly,both%20classification%20and%20regression%20problems..

[12] Raycad, "Convolutional Neural Network (CNN)," 14 November 2017. [Online]. Available: https://medium.com/@raycad.seedotech/convolutional-neural-network-cnn-8d1908c010ab.

[13] L. Craig, "convolutional neural network (CNN)," [Online]. Available: https://www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network.

[14] A. Rosebrock, "Convolutional Neural Networks (CNNs) and Layer Types," 14 May 2021. [Online]. Available: https://pyimagesearch.com/2021/05/14/convolutional-neural-networks-cnns-and-layer-types/#:~:text=The%20last%20layer%20of%20a,understanding%20its%20different%20layer%20types.

[15] Y. Gavrilova, "Convolutional Neural Networks for Beginners," 2 August 2021. [Online]. Available: https://serokell.io/blog/introduction-to-convolutional-neural-networks.

[16] GeeksforGeeks, "Convolutional Neural Network (CNN) in Machine Learning," 13 March 2024. [Online]. Available: https://www.geeksforgeeks.org/convolutional-neural-network-cnn-in-machine-learning/#.

[17] T. Shah, "About Train, Validation and Test Sets in Machine Learning," Medium, 6 December 2017. [Online]. Available: https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7#:~:text=The%20validation%20set%20is%20also%20known%20as%20the%20Dev%20set. [Accessed 1 October 2024].

[18] B. D. N. J. I. A. I. O. J. P. Z. Wevodaur, "The Temple University Digital Pathology Corpus: The Breast Tissue Subset," 2021. [Online]. Available: extension://bfdogplmndidlpjfhoijckpakkdjkkil/pdf/viewer.html?file=https%3A%2F%2Fisip.piconepress.com%2Fpublications%2Fconference_presentations%2F2021%2Fieee_spmb%2Fdpath%2Fabstract_v22_with_poster_v12.pdf. [Accessed 1 October 2024].

[19] "What is the resolution of my microscope in microns and image scale in microns per pixel?," Celestron, 18 December 2013. [Online]. Available: https://www.celestron.com/blogs/knowledgebase/what-is-the-resolution-of-my-microscope-in-microns-and-image-scale-in-microns-per-pixel?srsltid=AfmBOopJrlmOj7PSmBwnIdOcZRctqUWSlYcwjVt8ulMmY_XIcZSkZnAs. [Accessed 1 October 2024].

[20] A. Corning, "Honey, I Shrunk the Display: Measuring Small LEDs, Pixels, and Subpixels," RADIANT Vision Systems, 22 July 2019. [Online]. Available: https://www.radiantvisionsystems.com/blog/honey-i-shrunk-display-measuring-small-leds-pixels-and-subpixels. [Accessed 1 October 2024].

[21] "Understanding FFT Overlap Processing Fundamentals," Tektronix, [Online]. Available: https://www.tek.com/en/documents/primer/understanding-fft-overlap-processing-fundamentals-0. [Accessed 1 October 2024].

[22] S. Kahn, "Smoothing a Noisy Signal with a Savitsky-Golay Filter," Maplesoft, 17 December 2019. [Online]. Available: https://www.maplesoft.com/Applications/Detail.aspx?id=154593. [Accessed 1 October 2024].

[23] Z. W. B. D. I. O. J. P. J. Simons, "The Temple University Hospital DPATH Corpus:," Temple University Hospital DPATH Corpus, 15 January 2021. [Online]. Available: https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fisip.piconepress.com%2Fpublications%2Freports%2F2021%2Ftuh_dpath%2Fannotations%2Fannotation_guidelines_v10.docx&wdOrigin=BROWSELINK. [Accessed 1 October 2024].