**Building ML Models for Digital Pathology Data Classification**

*Zephan Joseph*

Department of Electrical and Computer Engineering, Temple University

zephan.joseph@temple.edu

**Introduction:** The medical discipline of pathology is one of the most crucial fields in medicine. Effective practices in pathology, the study of disease in biological tissue, provide paths to diagnosis and treatment for a wide range of human diseases. As technology has progressed, so has the field of digital pathology. Digital pathology has been expanding over the years, and breakthroughs like whole slide imaging have allowed for digitized images of biological tissue to be expanded and thoroughly studied and examined for diseases. This has also opened up the possibility for automated systems that are able to analyze these digitized images to be implemented, specifically for AI based algorithms that are able to increase the efficiency and productivity of the pathology process. For this project, we were tasked with creating machine learning models that are able to take data from digital pathology images of breast tissue and classify the data to be associated with one of 9 specific coefficients: artifact, background, null, normal, inflammation, nonneoplastic, suspicious, ductal carcinoma in situ, and invasive ductal carcinoma.

To train the ML algorithms that we developed, we were given data in the form of the 9 most significant coefficients of a Discrete Cosine Transform (DCT) of a selected area of a digital pathology image that was manually annotated. These coefficients and their associated features were put into various .csv files for model training, validation and evaluation, but the evaluation data was unlabeled as part of a blind competition. Taking the DCT of these images allows for an informative, frequency-based representation of the contents of the image which creates low-frequency coefficients (associated with smoother structures, like that of background tissue or larger cell clusters) and high-frequency coefficients (associated with finer details in the images, important in distinguishing cells by their boundaries and fine-grain structures). Taking the most significant coefficients of the DCT preserves the most important information of the image and removes redundant data that doesn’t impact important decision making or contributes to noise. Generating data in this format creates an effective method to build and train the ML models that we desire. To implement the best models for this data, I built and tested several common algorithms and settled on a Random Forest algorithm and a Convolutional Neural Network. Each algorithm generated a file of predicated labels (hypothesis file) for blind evaluation for the competition.

**Algorithm No. 1- Random Forests:** For the first (non-neural network based) algorithm, I decided on using a Random Forest (RNF) algorithm. RNF is an algorithm that builds decision trees that work together to make decisions on data classification. The algorithm takes labeled data, generates subsets of that data using bootstrapping (random sampling with replacement), trains multiple trees on bootstrapped samples (by considering a random subset of features at each tree node and taking the best feature of each subset) then aggregate the results from each tree and makes predictions based on majority voting. I decided that RNF would be a productive method for this algorithm as it is well suited for multi-class scenarios such as this, can handle high dimensional data well, and does not make assumptions about the data like other classifiers tend to do. I did try to implement a Support Vector Machine (SVM) algorithm, but I was unable to generate any significant results from that, likely because we did not have an insight into the decision boundary of the data which made it difficult to set the kernel as opposed to how RNF does a good job with complex decision boundaries. After multiple rounds of testing, I settled on an algorithm that used 1000 trees at maximum depth of 25, and each split occurred after a minimum of 8 branches. The final decision in the tree needed to have a minimum of 27 training samples, and after each split of the tree the square root of the total number of features is randomly selected and considered for splitting. I also added a parameter that adjusted weights inversely to class frequencies (to combat class imbalances). I did some preprocessing as well- I removed low-variance features with a variance threshold and standardized the data before they were passed into the model. This was done to remove any features that were unnecessary for learning and to help stabilize the RNF trees.

**Algorithm No. 2 Description- Convolutional Neural Network (CNN):** For our second algorithm, we were tasked with building a neural network-based system. Neural networks can be significantly more complex (both theoretically and computationally) than other algorithms, so it was important to be intentional with how the model was made. I began by experimenting with a multilayer perceptron approach. Multilayer perceptrons consist of an input layer, hidden layers and an output layer all interconnected and use non-linear activation functions that can handle non-linear data well and can learn complex patterns within the data. But, this ended up being a poor approach for this solution as MLP’s are prone to overfitting, struggle with large input sizes, and need more effort than CNNs to generalize well. CNNs can preserve patterns, require fewer parameters, and inherently generalize well, so I shifted to this method instead. For my final algorithm, I created a 5 layer CNN. Each layer learns the data and local patterns (edges, textures, etc.) and uses a non-linear activation function (ReLU) to learn the non0linear relationship between the features. The 1st, 2nd and 4th layers implemented max pooling for down sampling and to keep the most important information between each layer. The last layer uses adaptive average pooling to keep a consistent output shape after data passes through the other layers. I also did some pre-processing of the data: normalization of the mean, inverting the features, and using a 90/10 split of the training and validation data all before the data entered the CNN. After passing through the CNN, I implemented a cross-entropy loss function to compute the weights and handle issues with class imbalance, used an adamW optimizer for more stability and better generalization, and ran the model over 40 epochs- I found through trial and error that the average loss plateaued at this point.

**Results:** Before submitting the final hypothesis files for scoring, I used a different scoring algorithm that was provided to us to test the results of the hypothesis files generated for the training and validation sets. Table 1 contains my scoring results, including the scores I got during testing and in the final blind evaluation. The results in the scoring algorithm gave an output in terms of error rate, so lower was better. Also, the scoring was weighted to consider the more important coefficient more than others. In the blind test, the results were as expected for the RNF. I worked under the impression that the dev set was similar to the eval set, and for the RNF dev I had an error rate of 55.12% which was reasonable in comparison to the result of the evaluation test (55.15%). But for the neural network, my results were a little more unexpected. I got very low error rates in my own testing, but in the evaluation I had an error rate of 52.54%. This may be due to overtraining of the system in my own testing, as I included some of the Dev set in the training. I thought adding in a little more data would be helpful, but in retrospect the existing training data was likely more than enough to train the model on.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Data Set** | | |
| **Algorithm** | **Train** | **Dev Test** | **Eval** |
| RNF (Local Test) | 16.26% | 55.12% | -- |
| CNN (Local Test) | 12.22% | 12.21% | -- |
| RNF (Blind Test) | 16.27% | 55.12% | 56.15% |
| CNN (Blind Test) | 12.22% | 12.20% | 52.54% |

**Conclusions:** In conclusion, this project gave me insight into the field of digital pathology and the intersection between machine learning and medicine. Reflecting on my results, I think more tuning in my RNF algorithm could have led to better performance. Also, other algorithms (specifically gradient boosting) may be better suited for this application. If I was to attempt this problem again, I would implement other data preprocessing methods as well, and I would not include any of the dev set in the model’s training. Earlier on in the process I fully pooled the training and dev set which led to great error rates, but they were the product of overtraining, which made them untrustworthy. Other data preprocessing and specific model parameter tuning may lead to better results in this application. I really see the benefits of building effective ML models in digital pathology, and I hope that as the field of machine learning progresses forward, more advances will be made here.

Table 1. Error Rates of ML DPATH Algorithms