Domain Adaptation Approach to Labeling Cell Types Using scRNA-seq Data

Dylan Xu, Forrest Gao, Thomas Chung

May 2025

1 Introduction

In the field of molecular biology, one of the most important experimental advances is single cell RNA-sequencing (scRNA-seq). Researchers now have the ability to analyze expression information at a single-cell resolution, providing richer information on transcriptomic readouts, expression changes, and splicing changes compared to bulk rna-sequencing. One of the largest challenges in scRNA-seq is labeling cells, as doing so often requires a combination of manual labeling, which is costly in terms of time, and dimensionality reduction, which is variable by algorithm. Individual labelers may introduce biases, and labeling often requires significant domain knowledge.

A major advantage in addressing the cell-labeling problem is the availability of large consortia that provide high-quality, expert-annotated datasets, especially for healthy cells. One large consortium from the past decade includes the Human Cell Atlas (HCA) project, which created a comprehensive map of all healthy human cell types. However, most molecular biology lab experiments as well as clinical experiments deal with diseased cells. Diseased cells may have significantly up or downregulated gene expression profiles compared to healthy cells. So, labeling diseased cells provides a novel challenge compared to labeling healthy cells.

All of this background information suggests a domain adaptation approach: there exists substantial pre-labeled data for healthy cells, but because most workflows involve diseased cells, we can 1. train on the healthy cells 2. see how well the model generalizes on the disease dataset. This machine learning paradigm is fundamentally one of multi-class classification (into different cell types), and training different classifiers of different complexity could offer varying levels of generalizability to the disease set.

Implementing this approach introduced several practical challenges that affected all of our models, regardless of their complexity. Here, we provide an overview of the general challenges we faced, while more model-specific details are described in the Methodology section. One key challenge is that many machine learning models rely on the data being approximately linearly separable, but gene expression patterns are often highly complex and nonlinear. Also, slight differences in gene expression profiles may be sufficient to drive different cell fates, making the classification problem more difficult. Furthermore, the high dimensionality of the gene expression data (around 30,000 features per cell) made training computationally intensive and time-consuming across all models. To address this, we applied principal component analysis (PCA) for dimensionality reduction, retaining the top 100 components that capture the most variance in the gene expression data. Another persistent challenge was the risk of overfitting, as gene expression data tends to be noisy. These factors required careful regularization, dimensionality reduction, and other strategies to ensure that the models would generalize well to diseased cell data.

Throughout this whole project, we never use the ground truth labels of the diseased cells for training to simulate real world conditions: in most biochemical experiments, there would be no pre-labeling of the cells.

The ground truth labels of the diseased cells were simply used to benchmark the accuracy of the models on the out of distribution (diseased) samples.

2 Data

Link to Dataset: The dataset file is named `biopsy_RNA_h5ad`, and it can be downloaded from this URL:

`https://figshare.com/articles/dataset/biopsy_RNA_h5ad/21919425?file=38883240`

This dataset consists of 29,332 healthy cells and 49,060 diseased cells, each with 28,638 gene-expression features.

To perform our experiment, we required a dataset that included both healthy cells and diseased cells, both of which had already been labeled. The base format scRNA-seq data. scRNA-seq follows the traditional N by d tabular format, where N is the number of samples and d is the number of features. Here, N corresponds with the number of cells, and d corresponds with the number of genes. The value of each feature is the UMI (unique molecular identifier) count of each gene, which equals to the expression content (i.e. how highly active or non-active a certain gene is). Furthermore, we required a rich variety of cell types for the prediction task so that our model learns to distinguish between many cell types.

With these requirements in mind, we leveraged a single-cell atlas recently published by Mennillo et al. (2024), which profiled ulcerative-colitis (UC) patients and healthy controls. The study involved twelve donors; colonic biopsies from right and left colon were cryopreserved, pooled by condition, and processed in a single 10x Chromium run to minimize batch effects (a form of data regularization in sequencing). The same patients also contributed peripheral-blood leukocytes, but we confined our experiment to biopsy data because cellular composition in tissue is the primary obstacle for clinical cell annotation.

To make the prediction task more difficult, we decided to use the "fine annotation" labels as the ground truth target. Initially, the cells were bucketed into broad cell groups, but the fine cell annotations offer a more fine-grained difference between the expression profiles of related cell types, thus making the learning task more difficult.

This dataset fits our domain-adaptation objective. There exists a rich, labeled source domain of healthy control cells that provide high-fidelity reference labels. In addition, the authors of the paper found that there was a pronounced distribution shift from healthy to diseased cells: UC biopsies show widespread transcriptional remodeling—both inherent to inflammation and potentially exacerbated by medication (5-ASA or vedolizumab). This makes generalizing from the healthy control cells to UC a more difficult task.

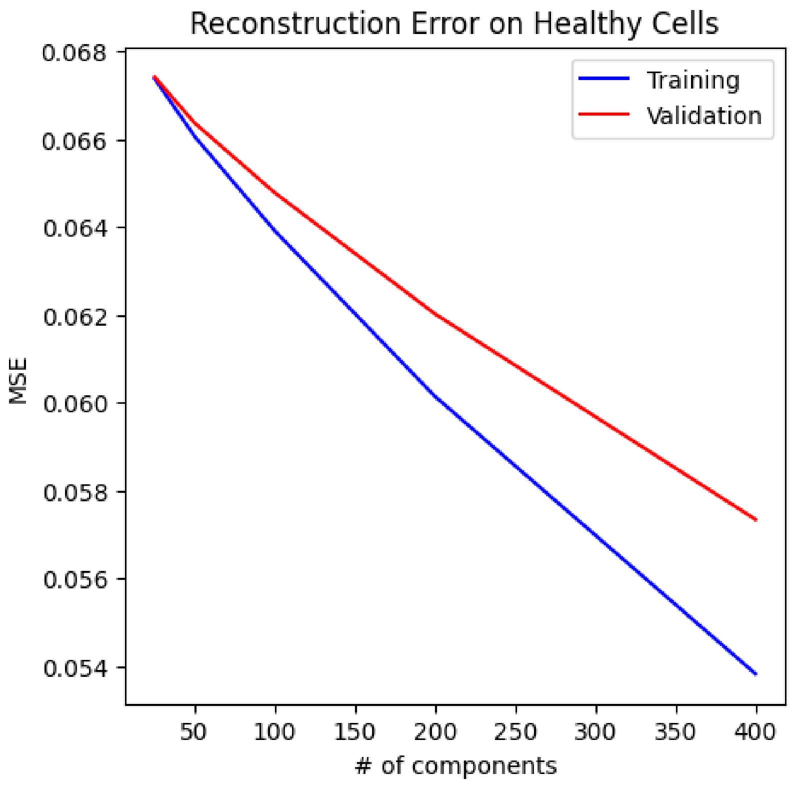
3 Dimensionality Reduction

Preface: Although principal component analysis (PCA) and highly variable gene (HVG) selection are typically considered preprocessing steps, we include this discussion here (rather than in implementation details) because they provide essential context for our models, which are trained on differently processed versions of the data.

Our dataset contains nearly 30,000 features, posing a significant challenge due to its high dimensionality. To address this, we applied PCA. After testing reconstruction loss across different numbers of components (see figure below), we determined that 100 components provided a good balance between information retention and dimensionality reduction.

In addition to PCA, the biological field commonly uses a dimensionality-reduction adjacent technique called highly variable gene (HVG) selection, which selects only the top x most variable genes. This technique helps reduce noise by focusing on genes that show the most meaningful biological variability, which is helpful for improving model performance and efficiency. Using PCA as a heuristic, we also experimented with selecting the top 100 most variable genes, using Scanpy to perform the HVG selection.

Empirically, we found that PCA outperformed HVG selection in our setting. As a result, all our models, except one (*4.2.2*), were ultimately trained using the data that was projected onto 100 PCA components.



We also experimented with Kernel PCA (*4.2.2*). Kernel PCA is a nonlinear dimensionality reduction technique that can capture complex, nonlinear relationships by projecting it into a higher-dimensional space before reducing its dimensionality. This exploration aims to assess whether the nonlinearity of Kernel PCA can extract richer features that improves classification performance beyond what is achievable with linear methods.

4 Methodology

4.1 Multi-Class Perceptron (Benchmark)

For benchmarking purposes, we implemented our own multi-class perceptron. We selected this model as our baseline because we hypothesized that it would yield the worst performance, given its simplicity and strong reliance on the data being linearly separable.

We approached the problem using a one-vs-all strategy, dividing the feature space into separate regions with distinct hyperplanes. Each hyperplane is responsible for determining whether a data point belongs to a specific class (yes or no).

For each class, we trained a binary perceptron that learns to distinguish that class from all others. During prediction, the model computes the output for each hyperplane, and the class with the highest confidence score is selected as the final prediction.

A separate hyperplane for each class:

$$\mathbf{w}_c \in \mathbb{R}^d \quad \text{for } c \in \{c_k\}_{k=1}^C \quad \text{and} \quad \mathbf{W} \in \mathbb{R}^{d\times C}$$

Consider the output of $\mathbf{w}_c^\top \mathbf{x}$ as a confidence score:

$$h(\mathbf{x}) = \arg\max_c \mathbf{w}_c^\top \mathbf{x}$$

Then, our goal became finding the hypothesis that minimizes the objective function

$$\mathcal{L}(\mathbf{W}) = \frac{1}{N}\sum_n \mathbb{I}\big(h(\mathbf{x}^{(n)}) \neq y^{(n)}\big)$$

To minimize the objective function, we implemented the following algorithm:

Initialize $\mathbf{w}^{(0)} = \mathbf{0}$

For $t = 0, 1, 2, \ldots$:

1. Find the misclassified examples:

$$h(\mathbf{x}^{(n)}) \neq y^{(n)}, \quad \text{let } \hat{c} = h(\mathbf{x}^{(n)}) \text{ and } c^* = y^{(n)}$$

2. Compute the loss:

$$\mathcal{L}(\mathbf{W}^{(t)}) = \frac{1}{N}\sum_n \mathbb{I}\big(h(\mathbf{x}^{(n)}) \neq y^{(n)}\big), \quad \text{where} \quad h(\mathbf{x}^{(n)}) = \arg\max_c \mathbf{w}_c^{(t)\top}\mathbf{x}^{(n)}$$

3. Update the weights for each incorrect prediction:

$$\mathbf{w}_{\hat{c}}^{(t+1)} \leftarrow \mathbf{w}_{\hat{c}}^{(t)} - \mathbf{x}^{(n)}, \quad \mathbf{w}_{c^*}^{(t+1)} \leftarrow \mathbf{w}_{c^*}^{(t)} + \mathbf{x}^{(n)}$$

Repeat (1)–(3) until convergence.

Convergence is typically defined as the point at which all training examples are correctly classified (i.e. the training loss is 0). However, in practice–especially when the data is not linearly separable–convergence is instead determined by stopping criteria such as reaching the maximum number of training iterations or when improvements in the loss fall below a specific threshold. For our implementation, we stop training after 100 iterations (or if all training examples are correctly classified, we have an early stop).

4.2 Logistic Regression

For our second model, we implemented a multi-class logistic regression model using scikit-learn's built-in function. We selected this model because logistic regression is widely used for classification tasks, and we wanted to compare its performance against the mutli-class perceptron.

The multi-class logistic regression models the probability of each class as a linear function of the input features passed through the softmax function. Like the perceptron, logistic regression models linear decision boundaries. However, unlike the perceptron, it outputs probabilities, which allows it to handle cases where the data is not perfectly linearly separable.

The softmax function computes the probability of class c as:

$$p(y = c \mid \mathbf{x}) = \frac{\exp(\mathbf{w}_c^\top \mathbf{x})}{\sum_{c'=1}^{C} \exp(\mathbf{w}_{c'}^\top \mathbf{x})}$$

Now, our learning task became finding the hypothesis that minimizes the objective function (cross-entropy loss)

$$\mathcal{L}(\mathbf{W}) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} \mathbb{I}(y^{(n)} = c) \log \Big(p\big(y^{(n)} = c \mid \mathbf{x}^{(n)}\big) \Big)$$

During training, the model minimizes the cross-entropy loss between the true class labels and the predicted probability distribution using the L-BFGS solver for optimization. When predicting, the model assigns each data point to the class with the highest predicted probability.

L-BFGS minimizes the cross-entropy loss in multinomial logistic regression by following a gradient-descent-esque approach. For each weight update, the optimizer uses the gradient, as well as an approximation of $\mathbf{H}_{\text{old}}^{-1}$ (where $\mathbf{H}$ is the Hessian matrix).

The gradient of the objective function is:

$$\nabla \mathcal{L}_{\mathbf{w}_c} = \frac{1}{N} \sum_{n=1}^{N} \Big(p\big(y = c \mid \mathbf{x}^{(n)}\big) - \mathbb{I}\big(y^{(n)} = c\big) \Big) \mathbf{x}^{(n)}$$

Using the gradient above, at each iteration, L-BFGS updates the weights as:

$$\mathbf{W}^{\text{new}} = \mathbf{W}^{\text{old}} - \mathbf{H}_{\text{old}}^{-1} \nabla L(\mathbf{W}^{\text{old}})$$

The full algorithm can be sketched as (stopping after a certain number of iterations):

Initialize: $\mathbf{W}^{(0)} = \mathbf{0}$

For $t = 0, 1, 2, \ldots$

$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \mathbf{H}_{(t)}^{-1} \nabla L(\mathbf{W}^{(t)})$$

4.2.1 Regularization

After conducting preliminary tests with our baseline logistic regression model, we observed a noticeable imbalance between the training and testing accuracies. Specifically, the training accuracy was significantly higher than the testing accuracy, suggesting that the model might be overfitting to the training data. To address this, we experimented with varying degrees of L2 regularization to constrain the model's complexity and hopefully improve generalization.

After L2 regularization is applied, the objective function becomes:

$$\mathcal{L}(\mathbf{W}) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} \mathbb{I}(y^{(n)} = c) \log \Big(p\big(y^{(n)} = c \mid \mathbf{x}^{(n)}\big) \Big) \; + \; \frac{\lambda}{2N} \|\mathbf{W}\|_F^2$$

$$\|\mathbf{W}\|_F^2 = \sum_{j=1}^{d} \sum_{c=1}^{C} (w_{jc})^2$$

We continued to use scikit-learn's L-BFGS solver to optimize the regularized objective function. Therefore, the optimization algorithm still follows the exact same gradient-descent-esque approach, but (to account for the regularization term) the gradient is now:

$$\nabla \mathcal{L}_{\mathbf{w}_c} = \frac{1}{N} \sum_{n=1}^{N} \Big(p\big(y = c \mid \mathbf{x}^{(n)}\big) - \mathbb{I}\big(y^{(n)} = c\big) \Big) \mathbf{x}^{(n)} + \frac{\lambda}{N} \mathbf{w}_c$$

Although we added regularization, the model still relies on the softmax function to predict class probabilities and assigns each data point to the class with the highest predicted probability.

4.2.2 Kernel PCA

Kernel Principal Component Analysis (Kernel PCA) is a nonlinear extension of traditional PCA that allows us to perform dimensionality reduction in a high-dimensional feature space. It relies on the *kernel trick*, which implicitly maps the input data into a higher-dimensional space without explicitly computing the transformation.
Let $K(\mathbf{x}, \mathbf{x}')$ denote a positive semi-definite kernel function. Then, the kernel matrix $\mathbf{K} \in \mathbb{R}^{N \times N}$ is defined as:

$$K_{ij} = K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$$

We center the kernel matrix $\mathbf{K}$ and perform eigendecomposition:

$$\mathbf{K} = \mathbf{U} \boldsymbol{\Lambda} \mathbf{U}^\top$$

where $\mathbf{U}$ contains the eigenvectors and $\boldsymbol{\Lambda}$ the corresponding eigenvalues. The low-dimensional representation of the data is obtained by projecting onto the top n eigenvectors:

$$\mathbf{Z} = \mathbf{U}_n \boldsymbol{\Lambda}_n^{1/2}$$

where $\mathbf{U}_n$ consists of the top n eigenvectors and $\boldsymbol{\Lambda}_n$ the corresponding eigenvalues.
Unlike linear PCA, which captures variance along straight-line directions, Kernel PCA can uncover complex nonlinear structures in the data, depending on the choice of kernel function.

Kernel Choices and Hyperparameter Considerations. When applying Kernel PCA followed by logistic regression, there are three main hyperparameters that influence downstream classification performance:

- **Kernel Type:** We explored both the radial basis function (RBF) kernel and the polynomial kernel, as discussed in class. The kernel determines how the input data is mapped into a higher-dimensional feature space:
  - *RBF kernel:*
  $$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2\right)$$
  which captures localized nonlinear structure by assigning high similarity to nearby points and low similarity to distant ones.
  - *Polynomial kernel:*
  $$K(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^\top \mathbf{x}' + r)^d$$
  which captures global nonlinear relationships depending on the degree d. We restricted our search to degrees 2 and 3 for the polynomial kernel because these degrees capture meaningful nonlinear interactions without introducing excessive model complexity. A degree-2 polynomial kernel maps the input space into quadratic interactions, which often align well with moderate nonlinearity in biological data. A degree-3 kernel extends this to cubic interactions, allowing for

more expressive boundaries. Higher-degree polynomial kernels (e.g., degree 4 or more) quickly become computationally expensive and tend to overfit, especially in high-dimensional settings like scRNA-seq, where noise is prevalent and the number of features is large. Lower degrees offer a good balance between expressivity and generalization, which aligns with our goal of building a model that generalizes well to the disease dataset.

- **Number of Components:** This controls how many principal components are retained after the kernel transformation. A higher number of components captures more variance in the data but may lead to overfitting and longer training time, whereas fewer components provide a more compact (but possibly less expressive) feature space.
- **Regularization Constant C:** In logistic regression, C controls the strength of L2 regularization. Smaller values of C correspond to stronger regularization, which helps prevent overfitting but may underfit the training data. Larger values reduce the penalty on large weights, potentially improving training accuracy at the expense of generalization.

Choosing the right combination of these hyperparameters is essential for balancing expressivity, generalization, and computational cost in a Kernel PCA + classifier pipeline to best determine if a nonlinear structure in data exists.

Because we are still using scikit-learn's Logistic Regression, the objective function and algorithm are both described in Section *4.2*.

4.3 Neural Network

For our most advanced models, we started by implementing a neural network. Although we had originally planned on fine-tuning a foundation model for this task (as mentioned in our report), we discovered a pre-existing pipeline for this task during our preliminary research. As we wished to write our own model for greater flexibility with the domain adaptation approach, we instead switched our attention to writing an artificial neural network. Then, we compared it to a more sophisticated domain adaptation model that was built on top of the ANN.

The loss function for the neural network is the traditional cross entropy loss, similar to that of the logistic regression. The only difference is that we utilize a per-batch update. Following the convention in the above sections, we will let the softmax function for the class c on a given input-output pair $(x^{(n)}, y^{(n)})$ be written as $p(y^{(n)} = c \mid \mathbf{x}^{(n)})$. So, for a batch of size B, we have the following loss function:

$$\mathcal{L}(\mathbf{W}) = -\frac{1}{B}\sum_{n=1}^{B}\sum_{c=1}^{C}\mathbb{I}(y^{(n)} = c)\log\left(p(y^{(n)} = c \mid \mathbf{x}^{(n)})\right)$$

For the learning algorithm, we used a traditional stochastic gradient descent (SGD) optimizer. For each epoch, we iterated through each of the batches to perform an update. On a given batch, we performed a forward pass through the current function $h(\mathbf{x})$, computed the loss function, and then updated the parameters via backpropagation. We let the torch package inherently compute the partial derivatives via the chain rule. In addition, we computed the mean loss per batch to help tune our hyperparameters.

For our deep neural network architecture, we utilized a fully feed-forward neural network on 3 hidden layers, plus one output layer (the logit layer). Each of the hidden layers involved 1024 nodes, and the logit layer had C nodes, where $C = 12$ because we classified 12 cell types.

Each hidden layer consisted of a linear transformation followed by batch normalization to stabilize activation distributions and improve gradient flow. We used leaky ReLU activations to prevent vanishing gradients, and applied dropout for regularization. This design promotes stable training and better generalization in high-dimensional gene expression space. The implementation section has further details about the hyperparameters.

4.4 Domain Adaptation

For the domain adaptation task, we adopt a regularization term onto the traditional cross entropy loss. We use s to represent our source domain and t to represent our target domain. Recall that we previously presented the result of the softmax function for class c on some logit z produced by input-output pair $(x^{(n)}, y^{(n)})$ as $p(y^{(n)} = c \mid \mathbf{x}^{(n)})$. For ease of notation, we will replace this term with $P_c^{(n)}$. When we specify the distribution this becomes $P_{s,c}^{(n)}$ for the source distribution and $P_{t,c}^{(n)}$ for the target distribution. Thus, our total loss function becomes:

$$\mathcal{L}_{\text{total}} = -\frac{1}{B_s}\sum_{m=1}^{B_s}\sum_{c=1}^{C}\mathbb{I}\big(y_s^{(m)} = c\big)\ \log(P_{s,c}^{(m)})\ -\ \lambda_{\text{ent}}\,\frac{1}{B_t}\sum_{n=1}^{B_t}\sum_{c=1}^{C}P_{t,c}^{(n)}\ \log(P_{t,c}^{(n)})$$

We used the same architecture here as for the vanilla neural network: a fully feed-forward neural network on 3 hidden layers, plus one output layer (the logit layer). Each of the hidden layers involved 4096 nodes: we pushed up the number of nodes in the hidden layer here relative to the vanilla neural network model because we were more confident that it would add greater expressivity to the model while preventing overfitting via the entropy minimization term.

Again, within each hidden layer, we applied a linear transformation followed by batch normalization, a leaky ReLU activation and dropout for regularization. The logit layer once again had $C = 12$ nodes. The full set of hyperparameters can be found in the implementation details.

5 Implementation Details

5.1 Data Preprocessing Steps

We will first briefly discuss the preprocessing steps that we did. In the original dataset, there were roughly 50 cell types, but many of them only applied to a couple of cells. Due to this data sparsity, we first subset for all cell types that had at least 300 samples in both the diseased and healthy distributions. In addition, some of the cell types were labeled with "NOS" (not otherwise specified), which according to the authors, represented cell types that were labeled with a broad cell type, as they were not confident enough to label them with a more granular cell type. Because we wanted to ensure the same level of granularity across all of our labels (e.g. one would not want "fruit" and "apple" to be two labels in an image classification problem), we decided to drop these cell types. This left us with 12 distinct cell types.

Next, we needed to engineer our features. As discussed above in Section 3, due to the "curse of dimensionality" in machine learning, and the fact that only a couple hundred of genes are important for cell fate determination, we performed PCA to transform all of our gene expression information into a lower dimension subspace (100 PCs).

5.2 Training-Validation-Test Split

We divided the dataset into three distinct subsets: training, validation, and test. The healthy cells were used for both training and validation, while the diseased cells were reserved exclusively to simulate a realistic domain adaptation scenario.

In particular, the healthy cell data was split into

1. 80% for training: used to fit the models
2. 20% for validation: used to tune hyperparmaters and monitor performance

The test set consisted entirely of diseased cells. They were only used for final performance evaluation to measure how well the models generalized to the diseased cell population.

This splitting strategy ensured that model optimization and evaluation remained unbiased and reflected real-world scenarios, where labeled diseased cells are typically unavailable for training.

5.3 Hyperparameters

Note that the multi-class perceptron, and the baseline logistic regression model have no hyperparameters. However, as previously mentioned, the multi-class perceptron was run for 100 iterations, and the logistic regression was run for 1000 iterations.

5.3.1 Logistic Regression with Regularization

We performed a grid search over different C values (where $C = \frac{1}{\lambda}$). Since C is the inverse of the regularization strength, lower C values correspond to stronger regularization. The values we tested were:

$$0.0001, 0.0003, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1$$

Empirically, 0.3 yielded the best target accuracy, so we selected this as our final value for C.

5.3.2 Logistic Regression with Kernel PCA

We experimented with several hyperparameter configurations when applying logistic regression after Kernel PCA. The three key hyperparameters are:

- **Kernel type** (`kernel`): determines how input data is mapped into a higher-dimensional space. We tested `rbf` and `poly` (polynomial). For `rbf` specifically, we also experimented with gamma, which defines how far the influence of a single training example reaches. Smaller values of γ make the kernel more global (smoother), while larger values make the kernel more local and sensitive to specific patterns.
- **Number of components** ($n_{\text{components}}$): controls the number of dimensions retained after the kernel transformation. Higher values preserve more variance but increase runtime.
- **Regularization constant** (C): inverse of the regularization strength in logistic regression. Lower C values impose stronger regularization and can improve generalization on noisy data.

The table below summarizes the results of each configuration:

Kernel	**Degree**	C	γ	$n_{\textbf{components}}$	**Target Accuracy**
`rbf`	–	1.0	auto	100	48.53%
`poly`	2	1.0	–	100	56.43%
`poly`	3	1.0	–	100	45.31%
`poly`	3	0.25	–	200	45.56%
`poly`	2	0.01	–	200	24.00%
`rbf`	–	0.5	0.01	100	79.70%

Table 1: Kernel PCA + Logistic Regression performance across different hyperparameter settings.

Analysis We began with the `rbf` kernel using the default settings ($C = 1.0$, $n_{\text{components}} = 100$, $\gamma =$ `auto`), which achieved 48.53% accuracy. We then explored polynomial kernels, which are more interpretable and were emphasized in lecture. Degree 2 yielded the best result (56.43%), while degree 3 showed weaker performance, likely due to overfitting or excessive nonlinearity in high dimensions.

Next, we tested whether increasing the number of components could help. Using degree-3 poly with $C = 0.25$ and 200 components led to no meaningful improvement. We also tested strong regularization ($C = 0.01$) with degree-2 poly and 200 components, which resulted in significant underfitting (24.00% accuracy), suggesting excessive regularization suppressed important variance.

Finally, we returned to the `rbf` kernel and conducted a more principled hyperparameter search. We set $C = 0.5$ to balance regularization and chose a smaller $\gamma = 0.01$ to produce a smoother, more global kernel function. This configuration led to a substantial improvement in target accuracy, reaching 79.70%, so our final kernel PCA logistic regression model used this configuration. The results suggest that when appropriately tuned, the RBF kernel is capable of extracting complex but generalizable structure in scRNA-seq data.

5.3.3 Neural Network

Hyperparameter	**Value**
Number of hidden layers (excluding logits)	3
Size of each hidden layer	1024
Dropout rate	0.2
Batch size	64
Optimizer learning rate	5×10^{-4}
Scheduler decay rate (γ)	0.99
Steps before decay	5
Total epochs	60

5.3.4 Neural Network With Entropy Minimization

Hyperparameter	**Value**
Number of hidden layers (excluding logits)	3
Size of each hidden layer	4096
Dropout rate	0.2
Batch size	64
Entropy regularization term (λ_{ent})	0.1
Optimizer learning rate	5×10^{-4}
Scheduler decay rate (γ)	0.99
Steps before decay	5
Total epochs	60

Note that for both of the neural network models, we searched through $[2, 3, 4, 5]$ as the number of hidden layers, $[512, 1024, 2048, 4096]$ as the hidden layer size, $[0.1, 0.2, 0.3, 0.4]$ as the dropout rate, and $[32, 64, 128, 256]$ as the batch size. For the neural network model with the domain adaptation approach, we tried values in the range $[0.05, 0.10, 0.15, 0.20, 0.25]$. The hyperparameters listed above represent the optimal results of our search, and they are the ones that are included in the notebook.

6 Results

6.1 Measures of Success

6.1.1 Accuracy

Intuitively, the simplest way to evaluate our models is by using the accuracy metric, which measures the overall proportion of correct classifications. This metric is both highly interpretable and serves as our primary measure of model performance. Note that by definition, $0 \leq \text{Accuracy} \leq 1$, and a higher Accuracy is better in general.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

6.1.2 F1 Score

We also report the F1 score to provide a more detailed assesment of model performance. The F1 score offers a single metric that captures both false positives and false negatives. For multi-class classification, we calculate the F1 score for each class and then average the results to obtain an overall score. This approach is especailly useful when the dataset has class imbalance, as it ensures that each class contributes equally to the final metric. Note that by definition, $0 \leq \text{F1} \leq 1$, and a higher F1 is better in general.

$$\text{F1} = \frac{1}{C}\sum_{c=1}^{C} \text{F1}_c \quad \text{where} \quad \text{F1}_c = \frac{2\,\text{TP}_c}{2\,\text{TP}_c + \text{FP}_c + \text{FN}_c}$$

$$\text{TP}_c = \text{Number of true positives for class } c$$

$$\text{FP}_c = \text{Number of false positives for class } c$$

$$\text{FN}_c = \text{Number of false negatives for class } c$$

6.1.3 Negative Log Likelihood

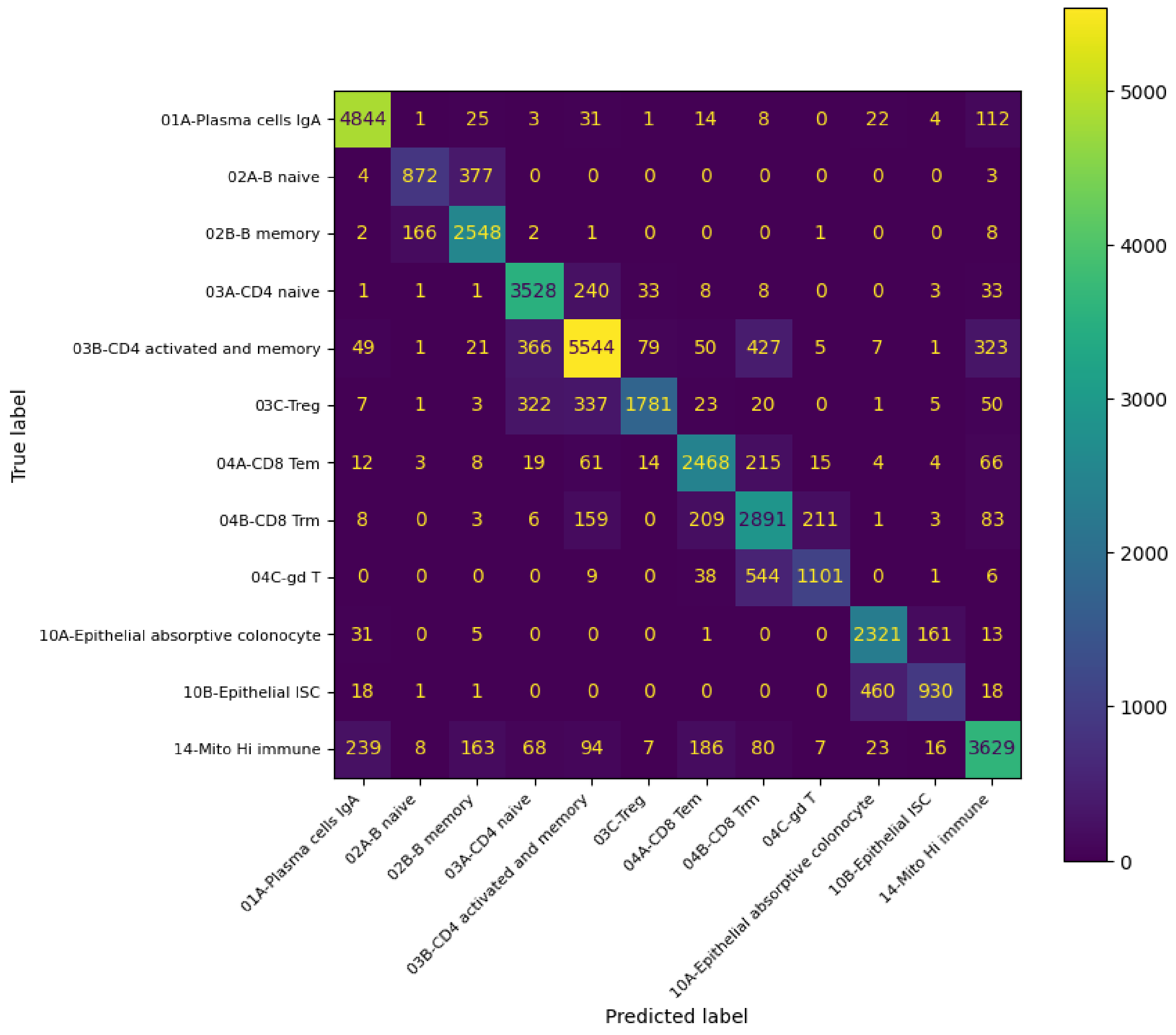
For all models except the multi-class perceptron, we are able to retrieve the predicted probability distribution over the classes. This allows us to evaluate performance using the negative log-likelihood (NLL) metric. Note that this metric is exactly the same as our objective function (i.e. cross-entropy loss) for logistic regression. NLL measures how well the predicted probabilities align with the true class labels by penalizing confident but incorrect predictions more heavily. A lower NLL indicates that the model assigns higher probabilities to the correct classes, reflecting better-calibrated predictions. Note that by definition, $NLL \geq 0$.

$$NLL = -\frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C} \mathbb{I}(y^{(n)} = c) \log\Big(p\big(y^{(n)} = c \mid \mathbf{x}^{(n)}\big)\Big)$$

6.2 Performance of Models

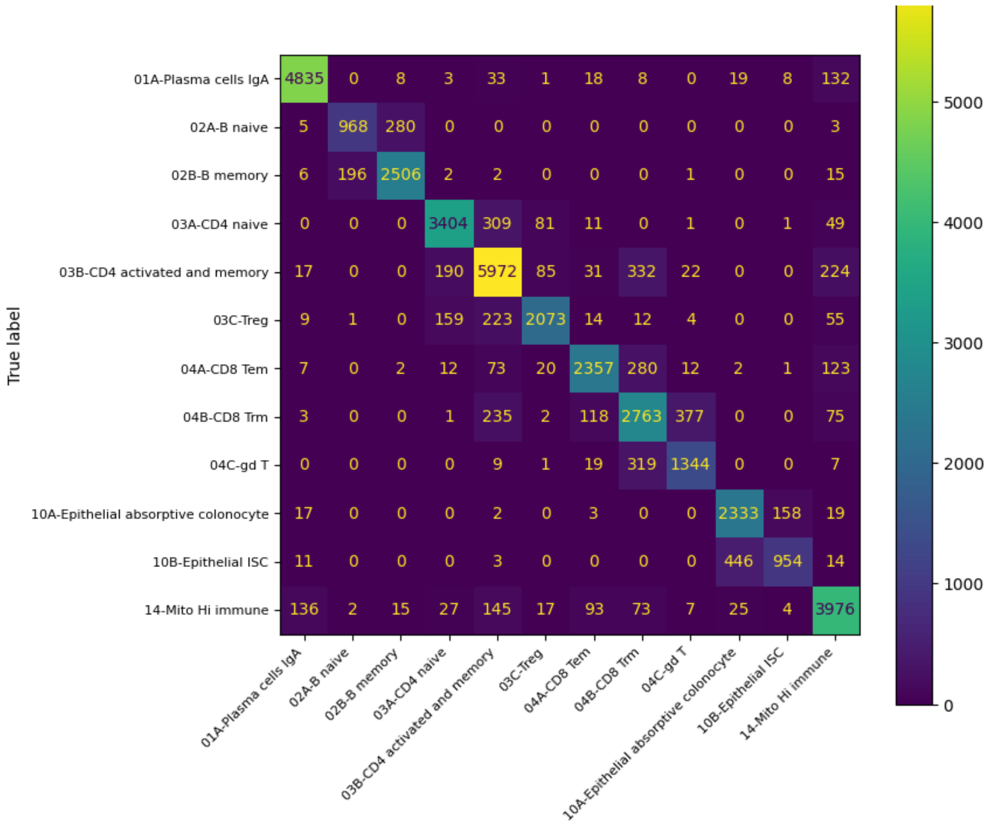
6.2.1 Multi-Class Perceptron

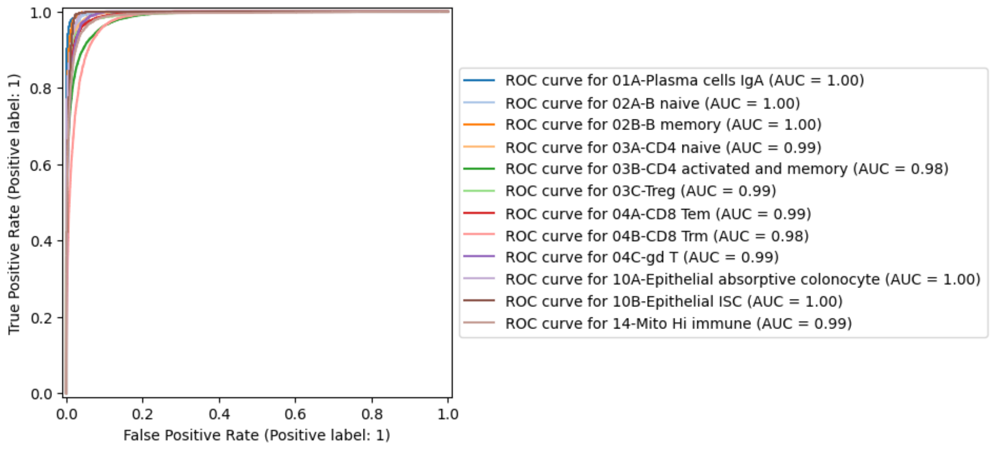
Metric	Score
Training Accuracy	87.07%
Validation Accuracy	85.89%
Target (Test) Accuracy	83.29%



6.2.2 Logistic Regression with Regularization

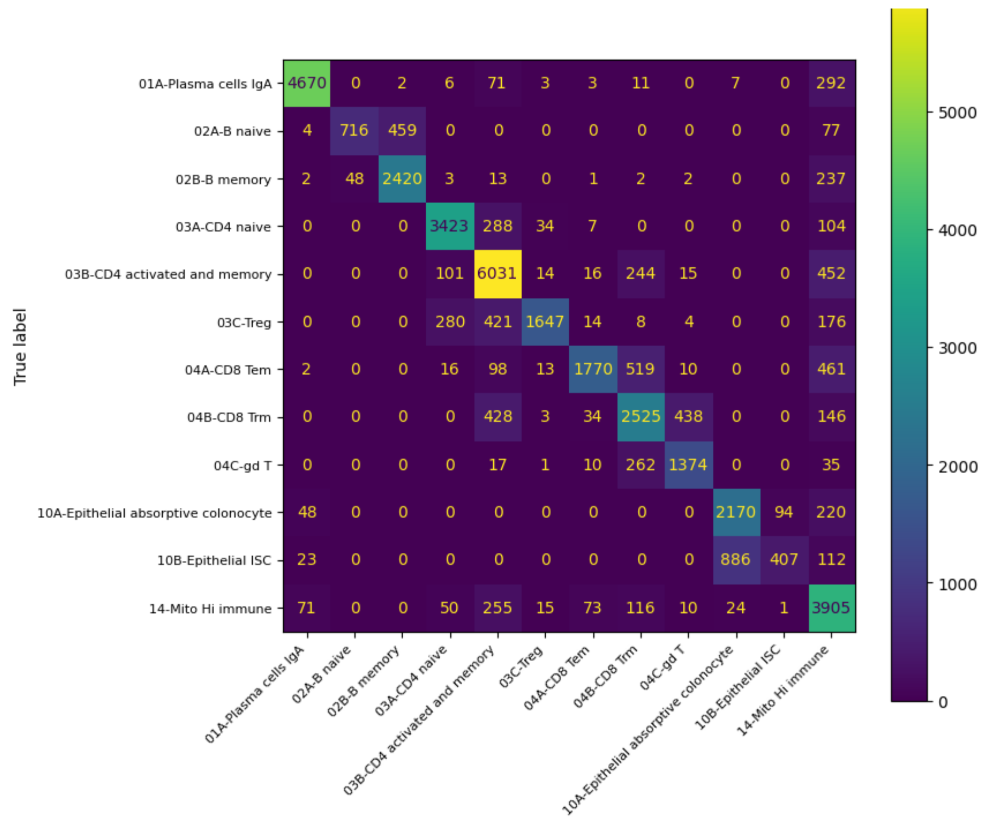
Metric	Score
Training Accuracy	90.63%
Validation Accuracy	87.82%
Target (Test) Accuracy	85.93%
NLL	0.378
F1 Score	0.844

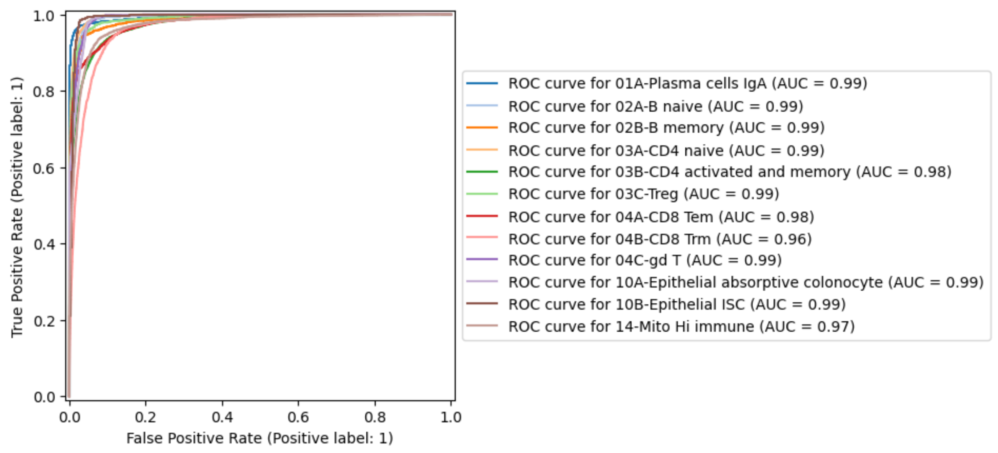




6.2.3 Logistic Regression with Kernel PCA

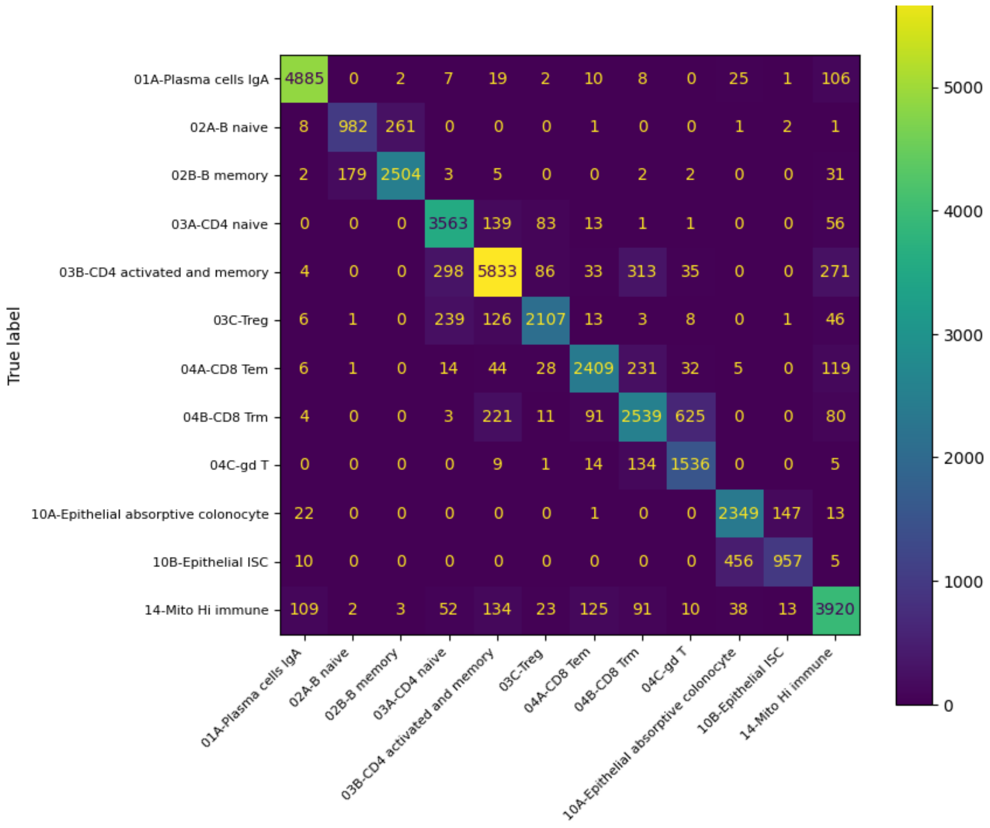
Metric	Score
Training Accuracy	84.14%
Validation Accuracy	83.29%
Target (Test) Accuracy	79.70%
NLL	0.683
F1 Score	0.761



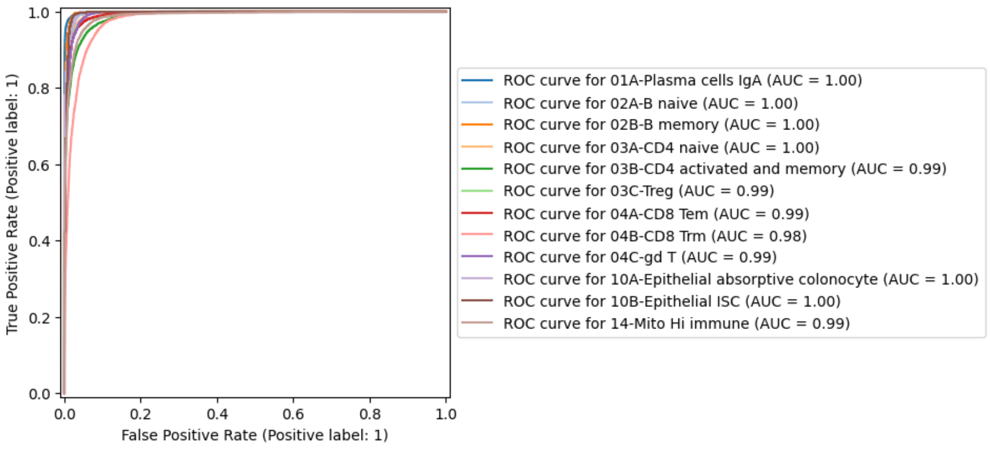


6.2.4 Neural Network

Metric	Score
Training Accuracy	90.46%
Validation Accuracy	87.84%
Target (Test) Accuracy	86.18%
NLL	0.363
F1 Score	0.846

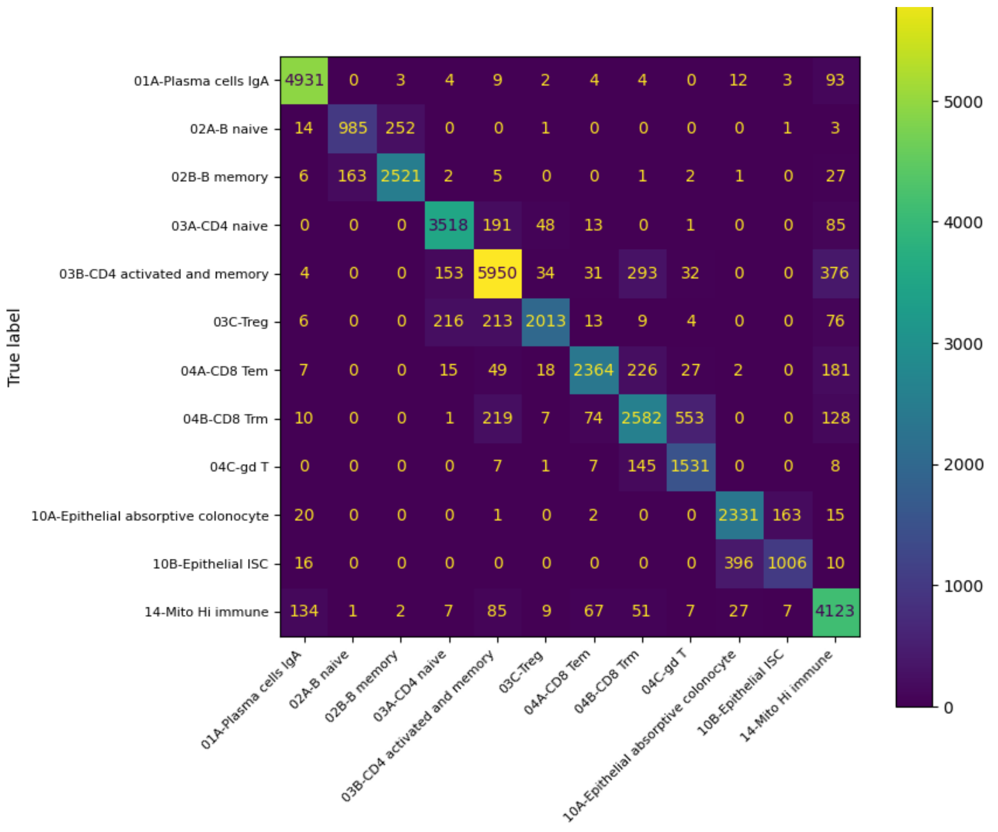


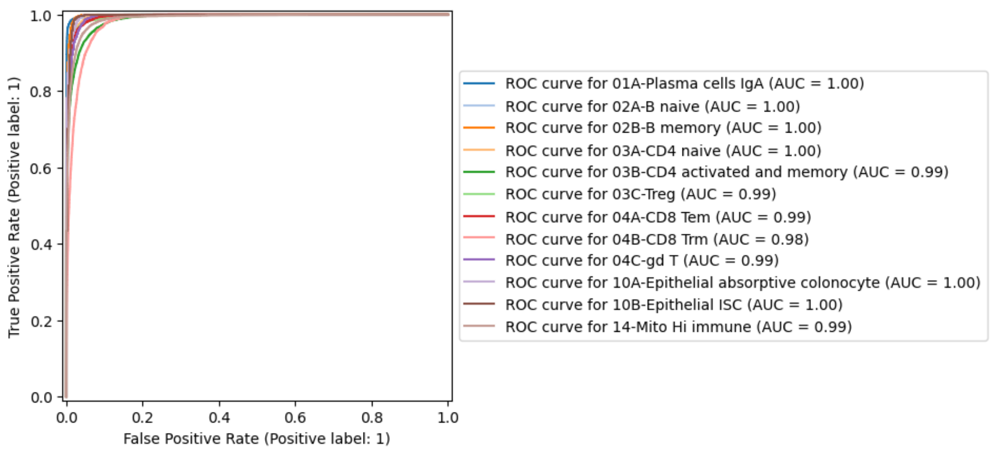
True label	01A-Plasma cells IgA	02A-B naive	02B-B memory	03A-CD4 naive	03B-CD4 activated and memory	03C-Treg	04A-CD8 Tem	04B-CD8 Trm	04C-gd T	10A-Epithelial absorptive colonocyte	10B-Epithelial ISC	14-Mito Hi immune
01A-Plasma cells IgA	4885	0	2	7	19	2	10	8	0	25	1	106
02A-B naive	8	982	261	0	0	0	1	0	0	1	2	1
02B-B memory	2	179	2504	3	5	0	0	2	2	0	0	31
03A-CD4 naive	0	0	0	3563	139	83	13	1	1	0	0	56
03B-CD4 activated and memory	4	0	0	298	5833	86	33	313	35	0	0	271
03C-Treg	6	1	0	239	126	2107	13	3	8	0	1	46
04A-CD8 Tem	6	1	0	14	44	28	2409	231	32	5	0	119
04B-CD8 Trm	4	0	0	3	221	11	91	2539	625	0	0	80
04C-gd T	0	0	0	0	9	1	14	134	1536	0	0	5
10A-Epithelial absorptive colonocyte	22	0	0	0	0	0	1	0	0	2349	147	13
10B-Epithelial ISC	10	0	0	0	0	0	0	0	0	456	957	5
14-Mito Hi immune	109	2	3	52	134	23	125	91	10	38	13	3920

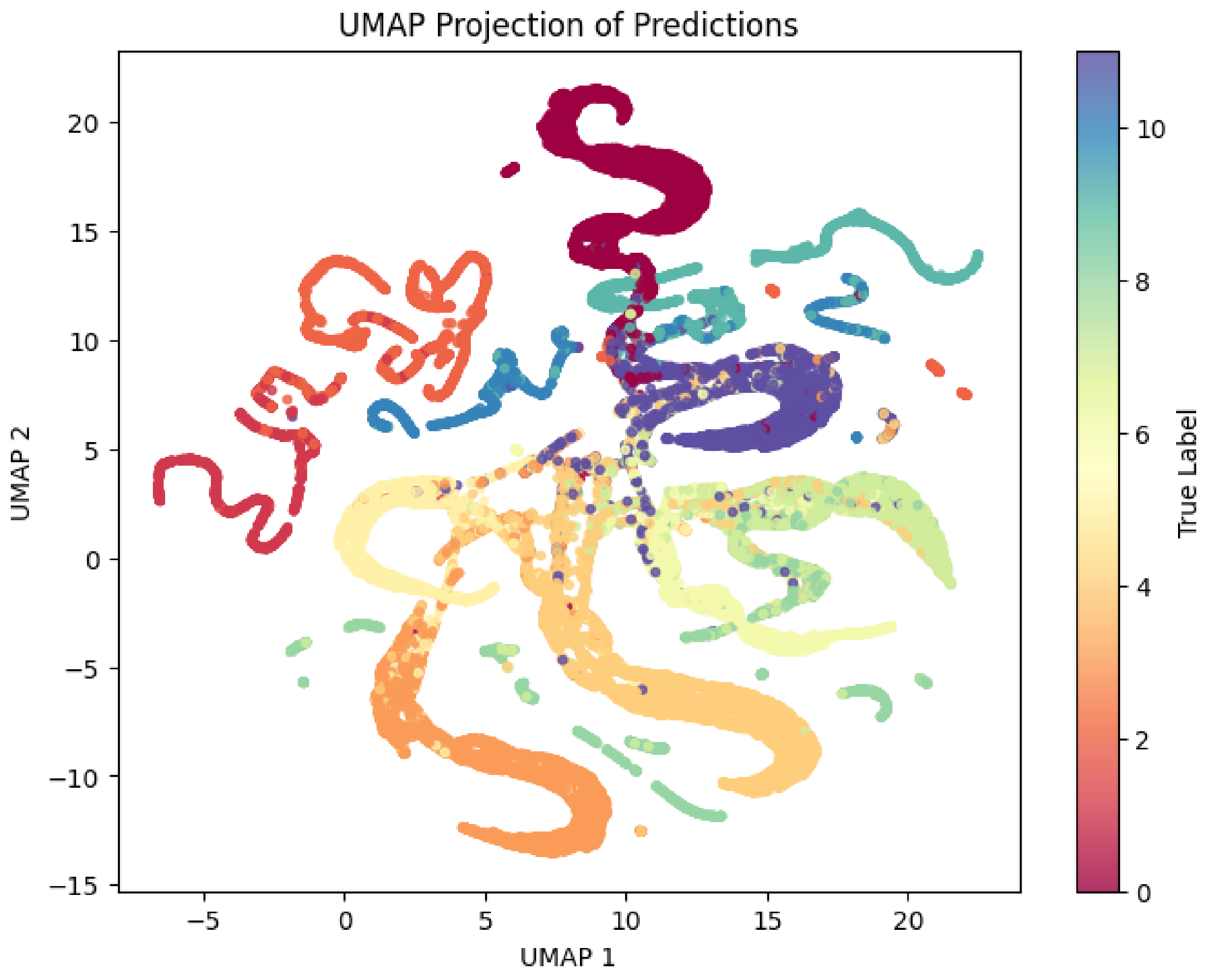


6.2.5 Neural Network with Domain Adaptation

Metric	Score
Training Accuracy	93.35%
Validation Accuracy	88.96%
Target (Test) Accuracy	86.87%
NLL	0.351
F1 Score	0.855







7 Analyis of Results

We will now provide a summary of the results displayed above, as well as interpretation and conclusions.

First off, our multi-class perceptron offered a lower bound on achievable accuracy. Despite its simplicity, linear separability produced reasonable results: training accuracy reached 87.1%, and validation reached 85.9%. On the out of distribution, i.e. the ulcerative colitis (UC) cells, the accuracy score was 83.29%.

Adding probabilistic outputs via the softmax activation, function and $L2$ regularization markedly improved robustness. The regularized logistic-regression model achieved a training accuracy of 90.6%, a validation accuracy of 87.8%, and a UC test accuracy of 85.9%. Note that the logistic regression converges better compared to the multi-class perceptron, which is expected due to the additional expressivity from the softmax function. For our other metrics, the logistic regression produced a Macro-F1 of 0.844 and a negative log likelihood score (NLL) of 0.378. We use these as the baseline Macro-F1 and NLL when comparing to the results of the neural network models.

Now, we are ready to analyze our non-linear models, i.e. the RBF kernel and the neural networks. Our experiment with kernel PCA significantly underperformed compared to the other models. Despite efforts to regularize and address the non-linearity introduced by kernel PCA, the model consistently lagged behind across all evaluation metrics. This outcome highlights that greater model complexity does not inherently lead to better performance; in this case, introducing complex features through kernel PCA failed to improve generalization on diseased cells.

We will now examine the performance of the two neural network models. The vanilla neural network reached a 90.46% training accuracy, which is comparable to that of the logistic regression, showing approximately equal levels of convergence. The validation accuracy was also similar to that of the the logistic regression. Notably, the neural network performed marginally better on the testing accuracy compared to the logistic regression, which shows that it may have generalized slightly better. The NLL score, which is a measure of prediction confidence, was also marginally lower for the neural network compared to that of the logistic regression, showing the vanilla neural network is slightly more confident. The vanilla neural network also performed almost identically to the logistic regression in terms of $F1$ score. The conclusion was that both approaches learned similar decision boundaries, with only a small generalization gain from the deeper network.

Finally, we examine the results of our domain adaptation approach. Notably, the domain-adapted neural network (with entropy regularization) was the top performer at 86.87% accuracy on the out-of-distribution UC diseased cells. This indicates a modest but consistent benefit from the domain adaptation strategy in generalizing to the new (diseased) domain.

Overall, model complexity improved performance only up to a point. Regularized logistic regression, trained on PCA features, already captured most class-separating structure, and the deeper network added little, implying that domain shift, not the model complexity, limits accuracy. Entropy-based domain adaptation provided the best, though modest, gain (percent accuracy, highest F1, lowest NLL) by boosting recall for rarer stromal and epithelial cells and yielding better-calibrated probabilities. In contrast, kernel PCA over-fit: it fit the source domain perfectly yet collapsed on diseased cells, underscoring the need for validation on target-like data. Overall, careful regularization and a light domain-aware objective outperformed both under-powered linear baselines and overly expressive, poorly constrained alternatives. In the future, we are interested in exploring hierarchical classification techniques. Because cells under a broad class are inherently very similar to each other, we believe that a model which first decides on a broad category (such as immune, epithelial, stromal, etc.) and then refines its prediction within that branch will increase accuracy. Splitting the task this way keeps each decision small, boosts recall for rare sub-types, and ensures any mistakes stay biologically "close" to the correct label.