Master of Engineering, Bioengineering

Introduction to Machine Learning in Computational Biology

Final Project Report

Term: Spring 2025

Cagin Tunc

PART 1

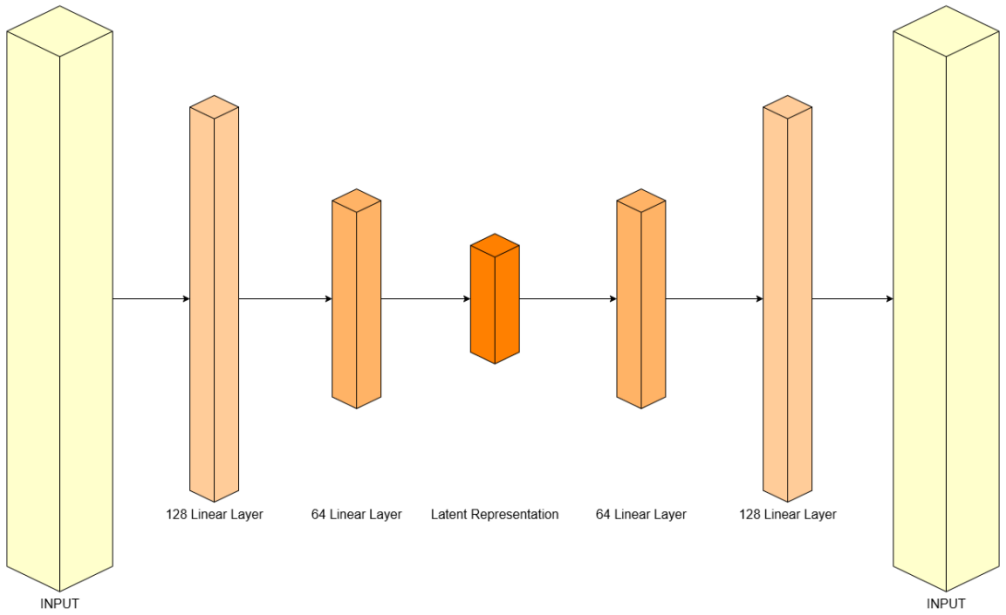

Figure 1: The architecture of the autoencoder model used during the project

Figure 1 shows the architecture of the autoencoder architecture. As we can see, both the input and the output size are the same that's why output name is also input to show that their size is equal. Also, since the output is actually the reconstructed form of the compressed input, we can say that we are trying to make the output as close as possible to the input. Both input and latent representation sizes are parameters that are going to be adjusted to get the best model.

The loss function that we are using is **MSE** (Mean Squared Error) which is the average of $(y^i - y^i_{real})^2$. It measures the reconstruction error between input and output. So, it looks how similar our output is to the original input.

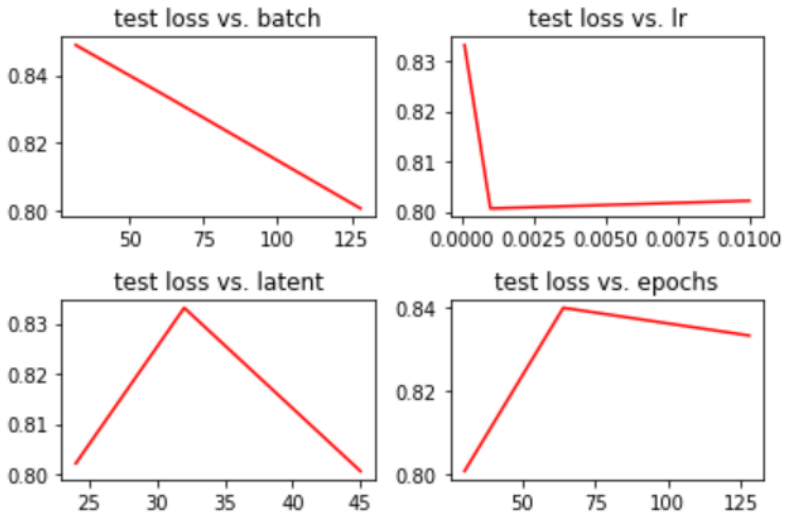

Figure 2: Test training before the real training process to find the best parameters; by using training size of 80% and testing size of 20% of the total data.

As we can see from figure 1, the best configuration is formed when the batch size is 125, learning rate is 0.001, latent space size is 45, and epoch number is 30.

batch	lr	latent	epochs	test_loss
128	0.001	45	30	0.800667

Then, we can train our model by using these configurations. In order to make testing into one part, the *Test* class was created with its associated test training function. It gets parameters such as latent space size, number of epochs, etc. And then trying all different combinations to get the best (least) test loss.

In the test training, following combinations of parameters are used (best parameters are written in bold):

- epochs: **30**, 64, 128
- learning rate: 1e-4, **1e-3**, 0.01
- latent representation dimension: 24, 32, **45**
- batches: 32, 64, **128**

After the model is trained, we can get the latent representation of our data by using our model. In order to do that, what we need to do is creating a new function and call model for each batch of data to get the encoded versions and then accumulate these encoded versions in a new array to return it as a result.

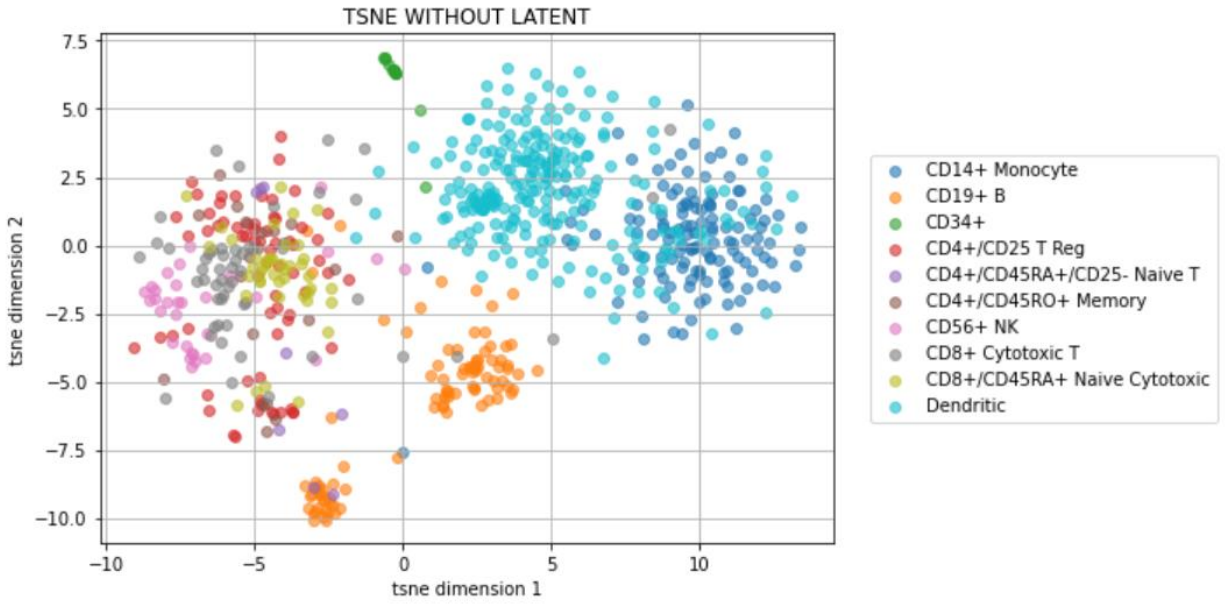

Figure 3: Dimensionality reduction technique (TSNE in that case) is used on the original data to visualize it.

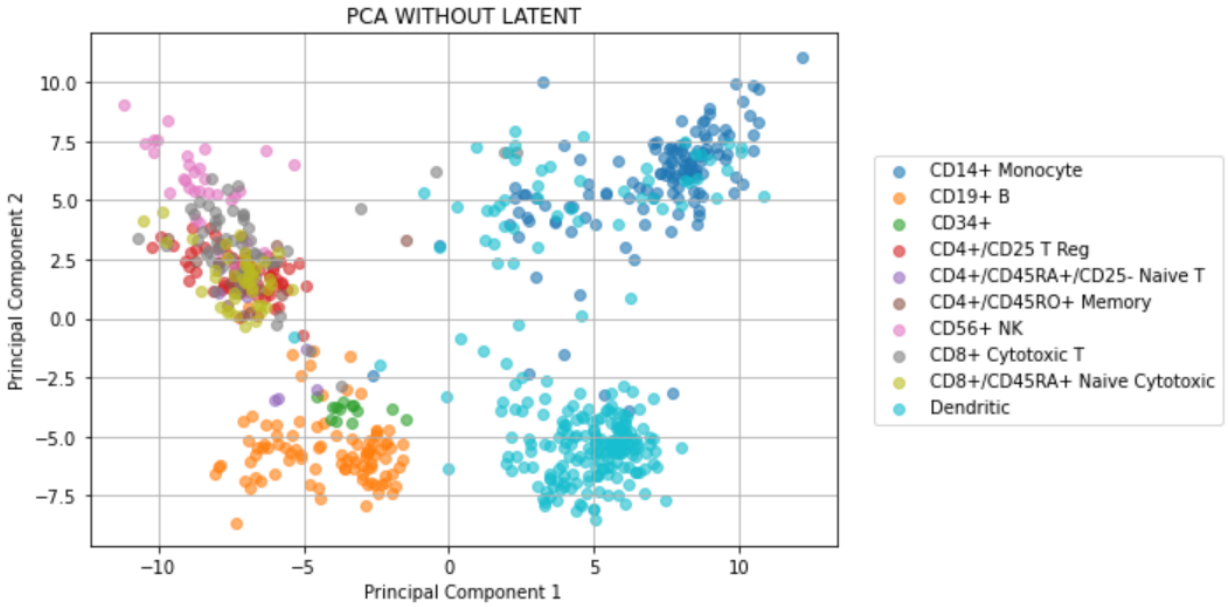

Figure 4: Dimensionality reduction technique (PCA in that case) is used on the original data.

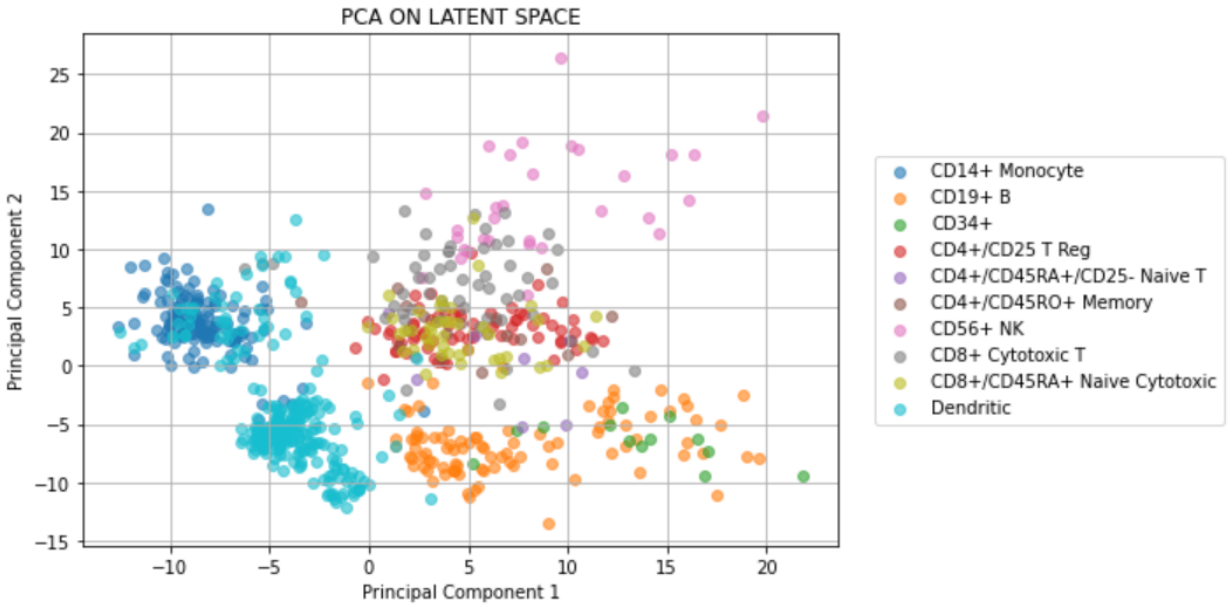

Figure 5: Principal component analysis on the latent space with size 45.

Based on our observations, the autoencoder appears to have performed well. As seen in Figures 4 and 5, the relative orientations of the clusters in the latent space are largely preserved compared to the original data. Although the absolute positions have changed, which is expected due to the dimensionality reduction from 765 to 45, the clusters remain well separated. This suggests that for tasks such as classification or clustering, operating in the latent space may be more effective than using the original high-dimensional data.

However, it is important to acknowledge that some information is inevitably lost during compression. If the primary goal is visualization rather than modeling, t-SNE may be more suitable. As shown in Figure 3, t-SNE effectively emphasizes local structure, making nearby points appear even closer and pushing outliers further from cluster centers, thereby enhancing visual separation. For example, in T-SNE we can distinguish *CD8+/...Naïve Cytotoxic cluster* from *CD4+/...T reg cluster* better than we do in other two figures.

PART 2

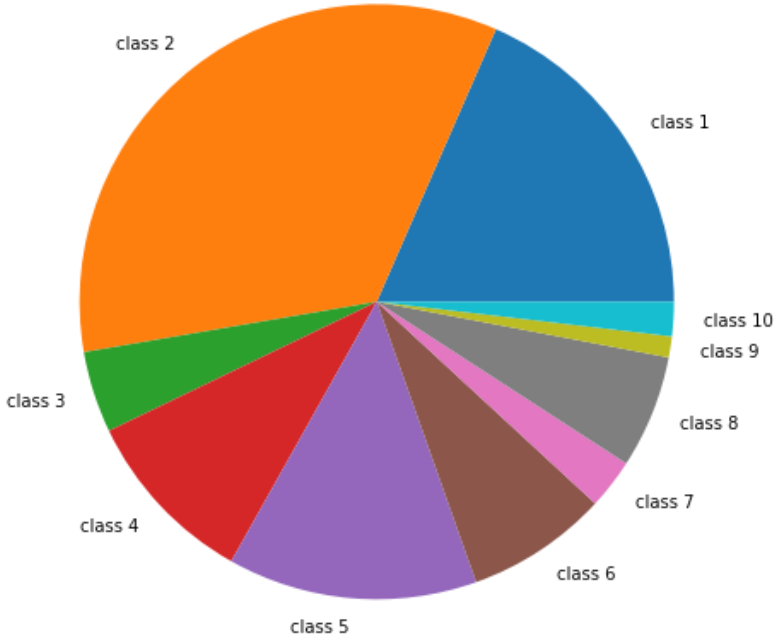

Figure 6: Class distributions in the dataset.

As figure 6 shows, there is a drastic class imbalance in the class distributions. It is important to note since we will decide our metrics and parameters based on our data.

In order to find the best parameters, the Grid Search algorithm is used with 5 cross validations to ensure that the score is not dependent on the test data position. This algorithm looks at the different combinations of each parameters to find the best one.

We cannot use "accuracy" as our metric since the class distribution is not balanced. Since we want to balance per-class quality and class importance. The best choice is **f1_weighted** for our case since we want a metric which can give us an avarage of f1 accross all classes weighted by class sizes.

Classifier Selection

1- **Random Forest Classifier**
   It is a complex model and it is working well when the relationships in the data are non-linear. And we can reach high-accuracy without spending too much time on finding

the best parameter. Since we used Grid Search algorithm, we defined a set of parameters to look.

- n_estimators: 1000, 1300, 1500
- criterion: “gini”, “entropy”, and “log_loss”
- max_depth: [10 - 23]
- class_weight: “balanced_subsample”

We specifically used *balanced_subsample* for *class_weight* since as we mentioned earlier, our classes are not evenly distributed. The “balanced” mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as *n_samples / (n_classes * np.bincount(y))*. The “balanced_subsample” mode is the same as “balanced” except that weights are computed based on the bootstrap sample for every tree grown.

At the end of the grid search, when we look at the top 5 best results, we can see that all of them used “gini”. And 60% of the top 5 models used 9 as a maximum depth of the trees.

param_class_weight	param_criterion	param_max_depth	param_n_estimators
balanced_subsample	gini	9	300
balanced_subsample	gini	9	1000
balanced_subsample	gini	9	1500
balanced_subsample	gini	13	1000
balanced_subsample	gini	9	1300

When we do the real training, we are using the parameters that we have found. Since classes are imbalanced, we need to look the confusion matrix and weighted f1 score.

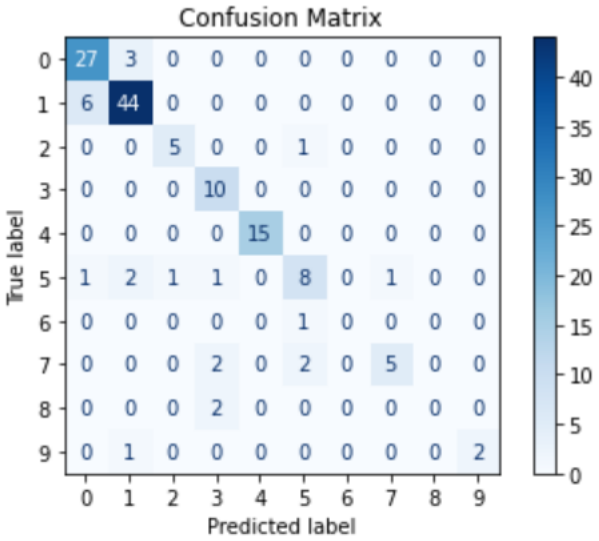

Figure 7: Confusion matrix of the random forest model

In figure 7, the diagonal shows the correct predictions. As we can see, more correct predictions have been made for the classes which has more points in the training dataset. However, we need to get a number to understand whether it is good or not. In order to do that, as we said before, we are using weighted f1 score. And it gives **0.8167**, which is relatively good.

2- Logistic Regression Classifier

Logistic regression is also used because of its speed. It is also the baseline model in machine learning tasks. If it performs bad, then we can focus on random forest. It assumes that the relationships are linear.

In this case, we need to look a lot of different sets of parameter configurations. And since some parameters are dependent on other parameters, we need to put conditions when we are doing grid search before our real training task.

If solver is “lbfgs”, “newton-cg”, or “sag” then penalty will be L2.

If solver is “saga” and penalty is “elasticnet” then we can define “L1_ratio” since elasticnet uses both L1 and L2 regularization at the same time and we can adjust their ratio. However, if solver is “saga” and penalty is either L1 or L2 then we can’t since we are only using one of them, we don’t need a ratio.

Notice that we can’t use “liblinear” in our case since liblinear can only handle binary classification. Since our classification task is multiclass, we can’t use it. “newton-cholesky” is not a good choice as well since the memory usage of this solver has a quadratic dependency on n_features * n_classes because it explicitly computes the

full Hessian matrix. Therefore, for our task we can use solver: “lbfgs”, “newton-cg”, “sag”, and “saga”.

In the first grid search, when C: either 0.1, 1, or 10 and solver can be any solver other than “liblinear”. It turns out the best results occured when C:0.1 or 1, and solver is “lbfgs”, or “newton-cg”. The best result has **0.8068** f1 score. So now, we can make our boundries smaller.

In the second grid search, we are just using two solver (“lbfgs”, or “newton-cg”) and C is between 0.1 and 2. It gives **0.8088** f1 score by using C around 0.2 and both solver.

In the third grid search, we are trying to walk around between 0.1 and 0.3. It gives us **0.8352** f1 score, which is pretty good.

Now that we have found our best parameters (C:0.18, solver: lbfgs, penalty: l2), we can train our real model and use it for predictions. After we trained our model by using our training data, we can see that logistic regression confusion matrix shows better predictions (see. Figure 8).

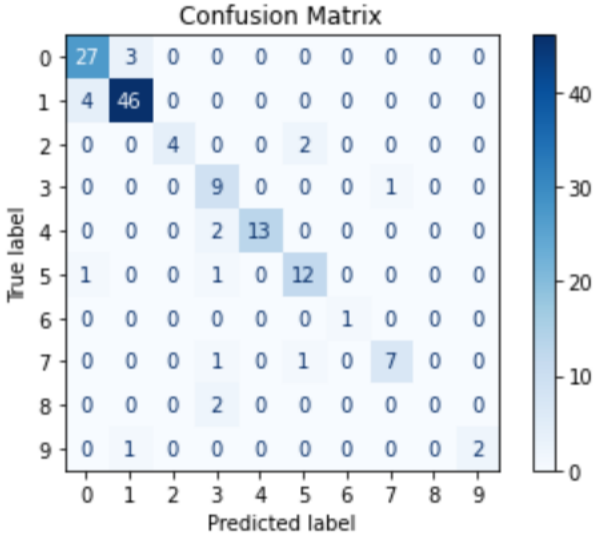

Figure 8: Confusion matrix of the logistic regression classifier. It shows **0.860** weighted f1 score.

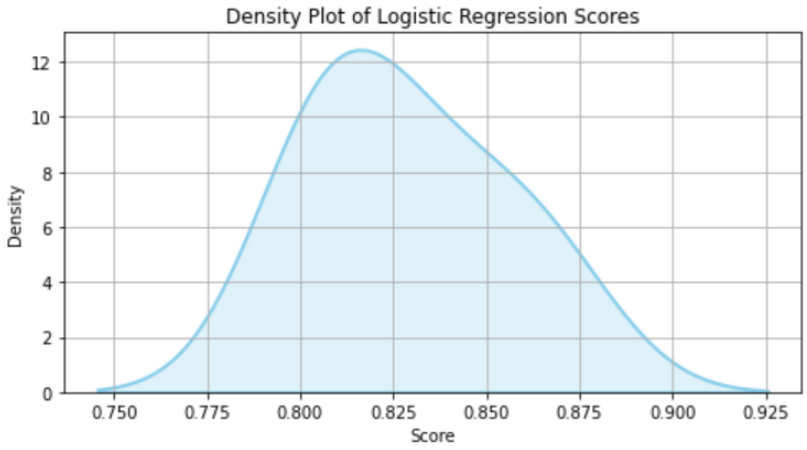

Figure 9: The density plot shows the density of the scores that we got. So, we can expect something around **0.82** for f1 score.

Discussion

As a result, logistic regression achieved a slightly better F1 score than random forest. I chose these two models to compare a simple linear model (logistic regression) with a more complex ensemble model (random forest). This dataset, with around 700 samples, provides a valuable opportunity to observe the behavior of models with different levels of complexity. It is well known that when working with small datasets, simpler models tend to generalize better, as they are less prone to overfitting. In contrast, complex models like random forests often require larger datasets to learn stable patterns. In this case, logistic regression outperformed random forest, likely due to its simplicity and robustness in low-data regimes. Additionally, random forests do not extrapolate beyond the patterns they have seen in the training data, which can be a limitation in biological tasks involving continuous variation. Therefore, logistic regression was preferred, not only for its computational efficiency, but also because it generalizes better under the conditions of this dataset.