BMEG 424 Project: Machine Learning in Neuronal Cell Type Classification

Emily Flaschner and Leon Li

April 12, 2024

Contents

1 Introduction

1.1 Overview of the study

Single-cell sequencing has emerged as a critical tool in cellular bioengineering and broader cellular research. Its high resolution enables the discovery of new cell types within tissues, the assessment of gene expression, cell-cell communication and a deeper understanding of transcriptomic landscapes. As such, it is widely employed in tissue characterization and to better understand cellular mechanisms and behaviour. While standard preprocessing steps for single-cell sequencing data are widely applicable, a pivotal challenge remains with cell annotation. Typically, this involves examining a cell's transcriptomic profile and assigning cell types based on known genetic markers. This method, however, is time-intensive, subjective, and operates under the assumption that all cells within a cluster are homogeneous. Furthermore, it is not scalable to other datasets and consequently limits the comparison of cells across samples and datasets.

The study titled, "Machine Learning for Cell Type Classification from Single Nucleus RNA Sequencing Data" aims to tackle these issues by applying various machine learning techniques to support cell annotation. It primarily focuses on the most popular supervised classification methods including random forest, logistic regression, neural network, light GBM and support vector machine (SVM) [1]. The study also examines highly granular and heterogeneous samples that contain multiple cell types. The selection of biological sequencing data highlights this focus by including a brain dataset, specifically a single-nucleus RNA sequencing (snRNA-seq) dataset of the middle temporal gyrus (MTG), containing 75 previously identified neural cell types. Additionally, the study includes a kidney dataset to test the platform's robustness. Although the kidney dataset was less well-characterized due to potentially requiring more precise parameter tuning for the kidney system, the model successfully differentiated between cell types. This suggests that even with the marginally lower accuracies in comparison to the MTG results, the platform is effective across biological systems.

Out of the supervised machine-learning models tested, LightGBM and multinomial logistic regression were found to demonstrate the best performance when tested using the kidney dataset. This still required testing specific subsets of data based on separate coefficient of variation thresholds but their overall performance was best. Ultimately, supervised machine learning methods present a promising methodology for scalable cell-type classification but will require further models, methods or tests to help distinguish between noisy and valuable features.

1.2 Rationale

We chose to reproduce this study primarily due to the inconsistent nature of cell classification within the field of single-cell sequencing. The reproduction of this study holds significance in determining the robustness and best methods for cell type classification using supervised machine learning methods which present a promising modality in improving the versatility of this analysis. By reproducing the analysis, we can test additional methods, and the combination of models while verifying the findings of this study. Furthermore, if we can validate the robustness of this study or find methods that improve the model's cell classification, we could confirm these methods as a comprehensive tool for classifying cell types within different niches in the human body.

1.3 Scope of the re-analysis

For the re-analysis, we focus on verifying the pre-processing pipeline, and testing another supervised machine learning method. To limit the scope, we are focusing on only one of the metrics presented, the Coefficient of Variation (COV). The study introduced a binary metric in conjunction with COV but concluded that COV was the more appropriate metric for comparison and as such, we plan to exclude the binary metric from our analysis. The steps of our analysis include re-running their pre-processing, generating and running a 1D convolutional neural network (CNN), and plotting their performance. The 1D CNN will be tested using the different subsets based on COVs presented in the paper and we will move forward with the best-performing COV subset for comparison.

2 Methods and Results

2.1 Pre-processing

The pre-processing pipeline described in the Le et al 2022 study contains two separate files in R and Python respectively. The first pre-processing steps are completed in R and consist of reformatting the data, performing normalization, and integrating the different files. This includes the exon and intron counts files, as well as the samples and gene metadata files. The R script is also responsible for generating a file of labels corresponding to the cell types in the MTG dataset and for generating a file of dataset sizes. The later pre-processing is done in Python and consists of calculating the COV and generating files for training, validation and testing based on different COV thresholds.

2.1.1 Pre-processing in R

The raw data is imported as two separate csv files containing the exon and intron data. The exon and intron data contain 50,281 genes as rows and 15,928 nuclei as columns. The files are combined, creating a comprehensive matrix which can then be normalized using counts per million (CPM).

Our interest in the dataset is to compare nuclei and classify the cell type. As such, normalizing based on the nuclei columns is expected rather than by the features as they distinguish between cell types. CPM follows this normalization pattern but to confirm whether CPM is a viable normalization method for the data, we took the average of every nuclei column. Between nuclei columns, we observed a large variance at $1.38x10^{11}$. This variance supports the need for normalization and because CPM normalizes based on columns and between samples, it is the better option over alternatives such as RPKM.

After CPM normalization, the data is then transformed using log2(cpm+1) which brings the distribution of data closer to a normal symmetric distribution, allowing for gene expression comparison.

```r
# Loads in data
intron_matrix <- read.csv("human_MTG_2018-06-14_intron-matrix.csv")
exon_matrix <- read.csv("human_MTG_2018-06-14_exon-matrix.csv")
full_matrix <- intron_matrix + exon_matrix

full_matrix_log2cpm <- log2(cpm(full_matrix)+1)

# Replace first row of gene id's
full_matrix[,1] <- intron_matrix[,1]
full_matrix_log2cpm[,1] <- intron_matrix[,1]
```

The samples csv file is imported to R and includes metadata that describes the 15,928 nuclei that were in the raw datasets. This allows us to apply previously completed external cell annotation performed by the researchers collecting the data. The following steps outline labels based on columns in the metadata including samples, cluster and filter out those without a specific cell-type assignment (cluster).

The genes csv file is then imported to R and used to replace the row names of our full matrix with genes. Filtering is performed to remove any genes with 0 expression across all samples.

```r
# Create cluster columns
samples <- read.csv("human_MTG_2018-06-14_samples-columns.csv", stringsAsFactors =
↪  FALSE)
labels <- samples[c("sample_name", "cluster")]
labels <- labels[labels$cluster != "no class",]
labels <- cbind(labels, str_split_fixed(labels$cluster, " ", n=Inf))
colnames(labels) <- c("sample_name", "cluster", "higher", "layer", "intermediate",
↪  "granular")

cell.types <- unique(labels$cluster)
```

```r
# Gets list of column indices for a particular cell type to subset
for(type in cell.types) {
  t(assign(paste0(type, ".celltypes"), labels[which(labels$clu == type),1]))
}

# Read in gene.rows file
gene.rows <- read.csv("human_MTG_2018-06-14_genes-rows.csv")

# Replace row names of full matrix with genes
rownames(full_matrix_log2cpm) <- gene.rows[["gene"]]

# Remove genes from whole matrix w/ 0 expression
keep <- rowSums(full_matrix_log2cpm[, 2:15929]) > 0
kept.genes <- full_matrix_log2cpm[keep,]
```

After standard filtering, the Median Gene Expression within each Cell Type is calculated to determine genes present in the majority of cell types. This is performed to limit the number of housekeeping genes that are included. Housekeeping genes refer to genes generally necessary across cell types. As the model is being created to distinguish cell types from one another, these are then filtered out as they pose little value in distinguishing cell types.

```r
# Subset cell subtypes
for (n in cell.types) {
  temp.cell.types <- get(paste0(n, ".celltypes"))
  temp.subset <- kept.genes[, temp.cell.types]
  assign(paste0(n, ".mm.subset"), temp.subset)
}

# Create empty median common gene matrix
cg.median.mm.df <- data.frame(matrix(ncol = 0, nrow = length(kept.genes[,1])))
rownames(cg.median.mm.df) <- rownames(kept.genes)

# Create empty count common gene matrix
cg.count.mm.df <- data.frame(matrix(ncol = 0, nrow = length(kept.genes[,1])))
rownames(cg.count.mm.df) <- rownames(kept.genes)

# Fill median common gene matrix with row medians from every cell subtype
for (n in cell.types) {
  temp.subset.common <- get(paste0(n, ".mm.subset"))
  temp.subset.median.common <- rowMedians(temp.subset.common)
  cg.median.mm.df <- cbind(cg.median.mm.df, temp.subset.median.common)
  temp.subset.count.common <- rowSums(temp.subset.common > 0)
  cg.count.mm.df <- cbind(cg.count.mm.df, temp.subset.count.common)
}

colnames(cg.median.mm.df) <- cell.types

# Calculate variance of the medians and remove genes w/ 0 variance
cg.median.mm.var <- apply(cg.median.mm.df, 1, var)
keep.var <- which(cg.median.mm.var != 0)
cg.median.mm.var <- cg.median.mm.var[cg.median.mm.var != 0]

# Subset list of genes after removing variance criteria
cg.median.mm.df.var <- cg.median.mm.df[keep.var, ]


# This line of code takes a long time
```

```
full_matrix_log2cpm_subset <- rbind(subset(full_matrix_log2cpm, select = -X),
↪  samples[["cluster"]])

keep.var.genes <- names(keep.var)
rownames(full_matrix_log2cpm_subset)[50282] <- "Classification"
final.matrix <- full_matrix_log2cpm_subset[c(keep.var.genes, "Classification"),]

write.csv(final.matrix, file = "everything.csv", row.names = TRUE, col.names = TRUE)
```

This filtered expression matrix is then saved to undergo additional processing in Python.

Lastly, we need to make two additional files for later steps in the machine learning model section. One for the cell type labels with a number assigned per type/cluster and another that contains the dataset sizes.

```
# Creating labels file
labels_list <- sort(unique(labels$cluster))
labels_df <- data.frame("Labels" = labels_list)
labels_df$Cluster <- c(0:74)
write.csv(labels_df, file="labels.csv")

# Creating dataset sizes file
subset.sizes = data.frame(matrix(ncol = 2, nrow = 0))

for (n in cell.types) {
  print(n)
  temp.subset.common <- get(paste0(n, ".mm.subset"))
  temp.size <- dim(temp.subset.common)[2]
  subset.sizes = rbind(subset.sizes, c(n, temp.size))
}

write.csv(subset.sizes, file = "subset_sizes.csv", row.names = TRUE, col.names = FALSE)
```

2.1.2 Pre-processing in Python

This processing section reviews the generation of the two feature selection metrics the paper uses for model construction: binary expression score (BIN) and coefficient of variation (COV). As mentioned previously, we are only using COV and as such only ran through the sections relevant to this metric.

Median Matrix - Cell Type This section creates a median matrix to be used in the calculation of the coefficient of variation.

```
cell_types = sorted(set(list(everything["Classification"])))
median_df = pd.DataFrame(np.zeros((len(cell_types), everything.shape[1]))) # 75 rows ×
↪  13945 columns
median_df.columns = everything.columns
median_df.index = cell_types
for classification in cell_types:
    subset = everything.loc[everything["Classification"] == classification]
    subset = subset.iloc[:, 0:subset.shape[1] - 1]
    median_df.loc[classification, :] = subset.median(axis = 0)
median_df = median_df.drop(columns = ["Classification"], axis = 1)
```

Coefficient of Variation This section of the code subsets the coefficient of variation based on the threshold value provided. For testing our later CNN model, we decided to use those addressed in the paper: 0.52, 1.5, 2.5, 3.5, and 4.5.

```
median_df_mean = median_df.mean(axis=0)
median_df_var = median_df.var(axis=0)
median_df_cov = median_df_var/median_df_mean
cov_threshold = 4.5
cov_filtered = everything[median_df_cov[median_df_cov > cov_threshold].index]
cov_filtered["Classification"] = everything["Classification"]
```

Stratification and Saving This section of the code saves training, validation and test files by stratifying the data and splitting it up 60/20/20, respectively.

```
X1 = cov_filtered.iloc[:, :cov_filtered.shape[1] - 1]
y1 = cov_filtered["Classification"]
training, test_val, y_train, y_test_val = train_test_split(X1, y1, stratify = y1,
↪  test_size=0.4, random_state=314)

X2 = test_val.iloc[:, 0:test_val.shape[1]]
test, validation, y_test2, y_val2 = train_test_split(X2, y_test_val, stratify =
↪  y_test_val, test_size=0.5, random_state=314)
training["Classification"] = y_train
test["Classification"] = y_test2
validation["Classification"] = y_val2

training.to_csv("filepath")
validation.to_csv("filepath")
test.to_csv("filepath")
```

2.2 Machine Learning

Now that the pre-processing is complete, we moved forward with the machine learning aspect of the paper. This does not involve us re-running their solutions but rather building and testing another supervised machine learning method, convolutional neural network (CNN). Although CNNs are typically used with spatial or structural data, their ability for localized and general pattern recognition makes them an interesting alternative to the other supervised machine learning methods presented. Furthermore, CNNs can learn from complex features in high dimensional spaces and potentially be used with multi-omics single-cell sequencing data. This could allow the combination of spatial sequencing data features with single-cell annotation. Consequently, we were interested in the model's performance when used with snRNA-seq for cell type classification.

2.3 Convolutional Neural Network (CNN)

As previously mentioned, we chose to build a CNN as another supervised machine learning model to compare against those used in the paper. This requires us to adapt the technique, so it can work for the snRNA-seq data we have, to a 1 dimensional CNN.

The design of the initial CNN architecture was based on established practices and general principles commonly adopted in the machine learning community. This foundational model serves as a first attempt of our CNN architecture, incorporating our own estimation. Since the model is merely used to select the base COV threshold within the preprocessing step, the absolute performance is less of an interest than the relative performance. The model is initialized with the function below.

```
def create_1d_cnn(input_size, hls = 5):
    model = tf.keras.Sequential()

    filters = 32

    model.add(Conv1D(filters=filters, kernel_size=3, activation='relu', input_shape =
    ↪  (input_size[1], 1)))
    for i in range(hls):
        model.add(Conv1D(filters=filters, kernel_size=3, activation='relu'))
        filters *= 2

        model.add(MaxPooling1D((3)))
        model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.50))
    model.add(Dense(75, activation='softmax'))
    loss = tf.keras.losses.SparseCategoricalCrossentropy()
    model.compile(optimizer="adam", loss=loss)
    model.summary()
    return model
```

We recognize that the classification of 75 cell types may require a deeper neural network, therefore 5 hidden convolutional layers for each of the following: Conv1D, MaxPooling1D, and dropout were chosen as the convolutional section. Throughout the convolutional layers, the filters were doubled with an initial value of 32 for every layer, allowing the model to capture a wide range of features from basic to complex. The MaxPooling layer is used to down-sample the feature representations, reduce dimensionality and allow the network to focus on the most relevant features. The dropout is used for regularization and helps prevent overfitting. After flattening the features, the dense layer is used to learn interactions and encode them back to the representation needed for the output. The last layer uses the softmax activation function to calculate a probability distribution over the 75 cell types. Lastly, the loss function chosen is Adam, a commonly used loss function that is not too sensitive to initialization and leads to faster convergence.

To properly build, tune and test the model we went through the following steps that mimicked that of the paper:

1. Training and COV selection
2. Parameter optimization and statistical comparison
3. Graphing and comparison

2.3.1 Training and COV selection

The following code encompasses our CNN and the steps involved in running it.

```
cm_dict = {}
# Get the pre-determined numerical representation of each cluster/cell type
labels = pd.read_csv("/.../data/labels.csv", index_col = 0)
labels.columns = ["Label", "Cluster"]
labels_dict = {}
for i in range(75):
    labels_dict[labels.iloc[i,0]] = i

dataset_sizes = pd.read_csv("/.../data/subset_sizes.csv", header = None)
dataset_sizes.drop(dataset_sizes.index[0], inplace=True)
dataset_sizes[2] = dataset_sizes[2].astype(int)

cov_threshold = [4.5, 3.5, 2.5, 1.5, 0.52]

for t in cov_threshold:
    print("Start for: ", str(t))
    # Import the train and val dataset

    train = pd.read_csv("/.../data/cov_filtered_" + str(t) + "_training.csv", index_col
    ↪  = 0)
```

```
validation = pd.read_csv("/.../data/cov_filtered_" + str(t) + "_validation.csv",
↪  index_col = 0)

train["Classification"] = train['Classification'].replace(labels_dict)
validation["Classification"] = validation['Classification'].replace(labels_dict)

X_train = train.drop(["Classification"], axis = 1)
y_train = train["Classification"]
X_val = validation.drop(["Classification"], axis = 1)
y_val = validation["Classification"]

print("The shape of the X_train: ", X_train.shape)
print("The shape of the X_val: ", X_val.shape)

# Initialize the model
model = create_1d_cnn(input_size = X_train.shape)
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0,
↪  patience=5, verbose=1)
print("This is the model summary for ", str(t))

# Log the loss for each iteration
history  = model.fit(X_train.to_numpy().reshape(X_train.shape[0], X_train.shape[1],
↪  1), y_train, validation_data=(X_val.to_numpy().reshape(X_val.shape[0],
↪  X_val.shape[1], 1), y_val), epochs=100, batch_size=1024, callbacks=callback,
↪  verbose=1)

predictions = tf.nn.softmax(model.predict(X_val.to_numpy().reshape(X_val.shape[0],
↪  X_val.shape[1], 1))).numpy()

hidden_layers = 5
cm = PredictionTable(predictions)

# Save the cm
cm_dict[t] = cm

fbeta_dict = {}
activation_function = "relu"
fbeta_dict[activation_function] = []
fbeta_dict[activation_function].extend(list(cm["fbeta"]))


Title = str(hidden_layers) + "Hidden Layers for threshold" + str(t)

print(Title)
fig = cm.plot.scatter(x="log_size", y="fbeta",ylim=[-0.05, 1.05],
↪  title=Title).get_figure()
# path = "Plots/" + Title + ".png"
# plt.savefig(path)
# Loss
ax = plt.figure().gca()
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.plot(history.history['loss'], label="Train Loss")
plt.plot(history.history['val_loss'], label="Val Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title(Title + " Loss")
plt.legend(loc="best")
plt.ylim(0, 1.5)
```

Note: The Prediction Table and the plotting section of the above code referred to the paper's original script.

Our first run was performed on all COVs to determine F-beta scores and which COV threshold was ideal to move forward with. F-beta scores refer to the performance of the model in terms of the number of cells properly classified. Correspondingly, the higher the F-beta score, the more cells are correctly classified.

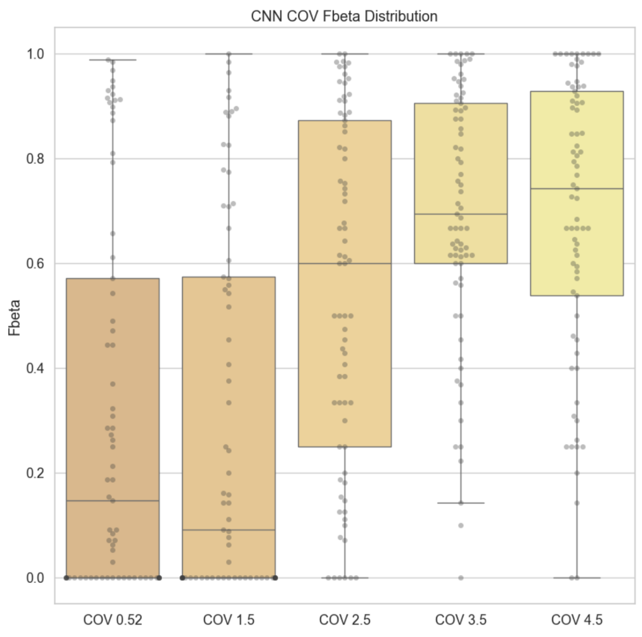

Figure 1: Comparing COV threshold training subset F-beta scores

The trend observed in the above graph (Figure 1), suggests that our model relies on COV filtering to achieve better results. Due to its increasingly improved performance at the more stringent thresholds, we can determine that our default CNN model is unable to pull out the key features regardless of the COV value. This demonstrates similar performance to the binary logistic regression, SVM, and neural network models evaluated in the paper.

Similarly to the paper, we aimed to determine which COV was the most statistically significant by performing multiple statistical tests. First, we conducted a Kruskal-Wallis test to evaluate the null hypothesis that all of the distributions were the same. Through this test, we determined enough statistical evidence that they were dissimilar to move forward with a Mann-Whitney U test which is a non-parametric test that allows for the comparison of two samples based on their means. The test was conducted on all the combinations and we found that the F-beta scores of COV 3.5 and 4.5 were statistically larger than the rest. Between the two, there was no statistical significance of one having the larger F-beta score so we used the same method as described in the paper of selecting the COV with the highest overall median. COV 3.5 had an overall median of 0.694 and COV 4.5 had an overall median of 0.743, leading us to move forward with the latter (Figure 1).

Table 1: Table of significant Wilcoxon COV comparisons

Comparison of COVs	**T-statistic**	**p-value**
$0.52 < 1.5$	2786.5	0.690
$0.52 < 2.5$	1601.0	1.412e-05
$0.52 < 3.5$	1121.5	6.753e-10
$0.52 < 4.5$	1140.5	1.050e-09
$1.5 < 2.5$	1528.0	3.627e-06
$1.5 < 3.5$	1051.0	1.074e-10
$1.5 < 4.5$	1056.5	1.201e-10
$2.5 < 3.5$	2015.0	0.006
$2.5 < 4.5$	2007.0	0.005
$3.5 < 4.5$	2599.0	0.400

2.3.2 Hyperparameters optimization and statistical comparison

Hyperparameters were optimized to produce the highest F-beta scores on the validation dataset. Based on the resources available, we selected the following hyperparameters to participate in a randomized search for the optimal set.

a) **Number of 1D convolutional layer**
The range of values is selected to be between 1 and 5 inclusive, based on the number of input features of the subsetted data with a COV threshold of 4.5. Calculations were performed to ensure that the maximum value of the convolutional layer paired with the other hyperparameters does not exceed and output negative dimensions within the layers. The number of convolutional layers controls the complexity, generalization, and feature extraction capability of a model. With a wide range of options, the randomized search has a higher possibility of striking a balance between the complexity of the model and its ability to generalize to another dataset.

b) **Kernel size**
The values selected are 3, 5, or 7. The kernel size controlled the receptiveness and its ability to extract meaningful features. A larger kernel size may help the network recognize the relationship between a larger set of genes, which may carry biological significance. In addition, a smaller kernel also leads to overfitting. Tuning an optimal kernel size can again strike a balance between the complexity and generalizability of the model.

c) **Number of dense layer**
The values selected range from 1 to 3. The dense layers after the convolutional layers carry the functionality of expanding the flattened features into higher-dimensional representations. A deeper dense network in the architecture, can help the model recognize more complex patterns and better distinguish the differences between classes. The dataset has 75 different cell types that are translated into 75 class labels. The model may need deeper layers to find patterns or features to distinguish such a high number of classes. Hence, tuning the number of dense layers may provide a balance between the model's ability to discriminate between classes and generalizability.

d) **Learning rate**
The range of values selected is from 1e-4 to 1e-2 with logarithmic sampling. The sampling method is well-established for learning rate optimization and it concentrates on smaller values, which can be beneficial for loss function stability. The learning rate was initialized for the Adam optimizer. Although the optimizer chosen is less sensitive to initialization than others, an appropriate initial learning rate can ensure convergence on time, especially for deeper neural networks.

The following code demonstrates the basic architecture of the CNN for randomized search of optimal hyperparameters.

Creating the 1D CNN

```
def create_1d_cnn_tuning(hp):
    model = tf.keras.Sequential()

    filters = 32
    kernel_size = hp.Choice("kernel_size", values=[3, 5, 7])
    hls = hp.Int('hls', min_value=1, max_value=7, step=1)
    padding = 'valid'
    if hls >= 5 or kernel_size >=5:
        padding = 'same'
    model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='relu',
    ↪  input_shape = (709, 1), padding = padding)) # 709 comes from the ncol of cov
    ↪  4.5

    for i in range(hls):
        model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='relu',
        ↪  padding = padding))
        filters *= 2

        model.add(MaxPooling1D((3), padding = padding))
        model.add(Dropout(0.25))

    #### End of conv layers ############################
    model.add(Flatten())
    #### Beginning of dense layers #####################
    nodes = 32
    for i in range(hp.Int('num_dense_layers', 1, 3)):
        model.add(Dense(nodes,
            activation='relu'
        ))
        model.add(Dropout(0.50))
        nodes *=2

    # Final layer of softmax for all 75 cell types
    model.add(Dense(75, activation='softmax'))


    loss = tf.keras.losses.SparseCategoricalCrossentropy() # from_logits=True

    model.compile(optimizer=Adam(hp.Float('learning_rate', min_value=1e-4,
    ↪   max_value=1e-2, sampling='LOG')), loss=loss, metrics=[f_beta] )

    model.summary()

    return model
```

The code above uses TensorFlow's hyperparameter Tuning module and the input hyperparameters are the randomly selected values for this trial. This function also permits changing the padding parameters for the convolution section of the model from "valid" to "same". This is specifically for deeper and larger values of kernel sizes to mitigate diminishing dimensions of deeper convolutional layers. Furthermore, the dense layers after flattening the convolutional layers are expanded so that there can be at most three layers with a 50% dropout rate. As mentioned before, the dense layers carry the function of re-expanding the features to distinguish between the 75 classes and a 50% dropout rate introduces regularization and reduces overfitting.

Defining our F-beta function

```
def f_beta(y_true, y_pred, beta=0.5):

    y_true = tf.cast(y_true, tf.float32)
    y_pred = tf.round(y_pred)

    # Calculate tp, fp, and fn
    true_positive = tf.reduce_sum(y_true * y_pred)
    false_positive = tf.reduce_sum((1 - y_true) * y_pred)
    false_negative = tf.reduce_sum(y_true * (1 - y_pred))

    # Precision, Recall, and F-beta
    precision = true_positive / (true_positive + false_positive +
    ↪   tf.keras.backend.epsilon())
    recall = true_positive / (true_positive + false_negative +
    ↪   tf.keras.backend.epsilon())

    fbeta = (1 + beta**2) * (precision * recall) / (beta**2 * precision + recall +
    ↪   tf.keras.backend.epsilon())

    return fbeta
```

This function calculates the F-beta score for the predictions, referencing the original paper's function. This is used as the compiler metric as shown above.

Defining and running our tuner

```
tuner = kt.RandomSearch(
    create_1d_cnn_tuning,
    objective=kt.Objective("val_f_beta", direction="max"),
    max_trials=30,
    executions_per_trial=2,
    directory='/.../tuning',
    project_name='CNN tuning'
)

tuner.search(
    X_train.to_numpy().reshape(X_train.shape[0], X_train.shape[1], 1),
    y_train,
    validation_data=(X_val.to_numpy().reshape(X_val.shape[0], X_val.shape[1], 1),
    ↪  y_val),
    epochs=100,
)
```

In the above code section, a RandomSearch tuner is initialized to obtain a higher F-beta score. Based on available resources, we decided to cover approximately 75% of the hyperparameter space and searched 30 unique combinations of values, each with 2 executions. This criterion allows us to explore our model appropriately with reduced variability in its training and tuning.

Later on, using the previous analysis, we imported training and validation data from the COV 4.5 dataset. The epoch parameter is chosen to be 100 because it can allow the model to converge to a stable solution and earn meaningful representations of the data.

With the CNN model trained and optimized for hyperparameters, we compared our base model to the optimized model and found that they were quite similar (Figure 2). Typically, we would expect to see a clear distinction between the default and optimal sections but we found that the distribution of points on the scatter plot overlapped significantly (Figure 2).

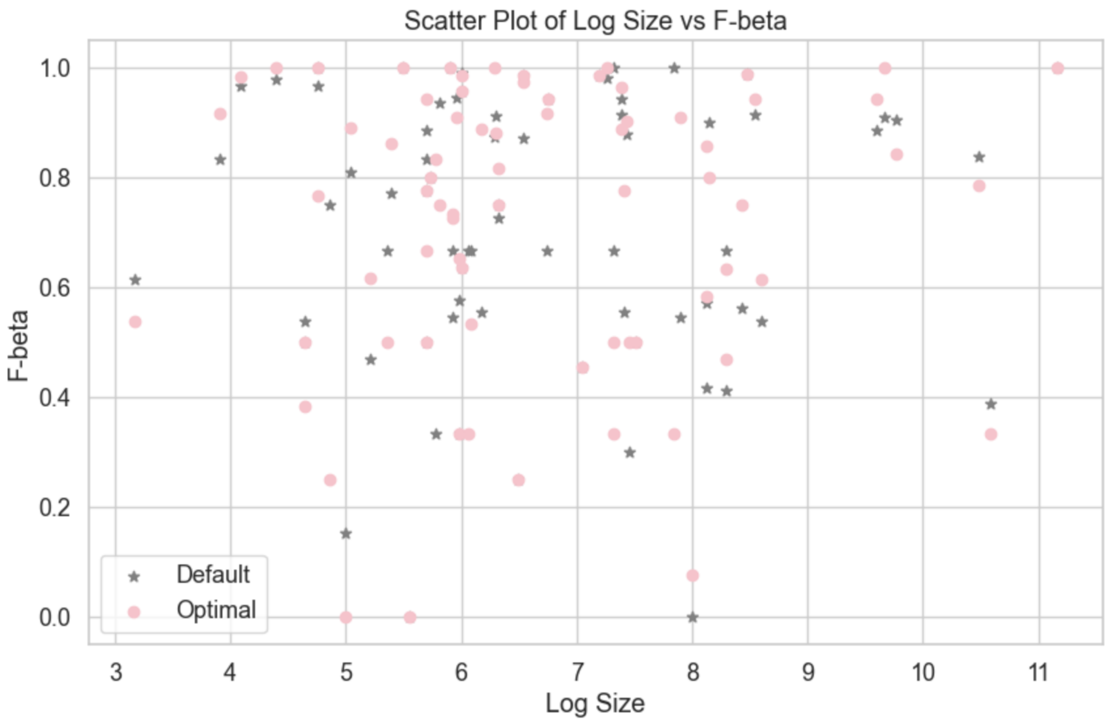

Figure 2: Scatter plot of the F-Beta scores between our default and optimized models

This is likely due to the depth of our hyperparameter optimization as well as our initial set-up. When constructing the model, we anticipated needing multiple layers due to the complexity of characterizing 75 cell types. As such, the model was constructed with a default number of 5 layers instead of the standard 1, to better functionally model which COV threshold would be optimal. We also chose our number of dense layers to be 1 in our testing and consequently, we happened to use the best options for two of our hyperparameters based on those we optimized (Table 2).

Table 2: Table of default and optimized hyperparameters

Parameters	**Default Values**	**Optimized Values**
Kernel Size	3	5
Hidden Layers	5	5
Number of Dense Layers	1	1
Learning Rate	0.001	0.000126

Nevertheless, the hyperparameter optimization performed was limited due to resources and a more extensive or exhaustive random search could have led to improved optimization of our parameters and fit. We anticipate that at a larger depth and with larger ranges of values to test, for instance, if we tested up to 7 layers, this may yield better results.

2.3.3 Graphing and comparison

Following our review of the hyperparameter optimization, we then aimed to review the optimized model across all three stratified datasets (training, validation and testing) as well as in comparison to the models described in the study. Looking at the scatter plots of our model at training, validation and testing, we generally see similar findings as to those presented in the paper but with decreased accuracy. The F-beta scores across the presented supervised machine learning models are generally very high with a score of almost 1 for every point (Figure 3A). Whereas our training dataset generally has high scores but contains a distribution of points at lower F-Beta scores (Figure 3B). This may be due to the limited depth of our hyperparameter optimization or the number of trials. With regards to the validation and testing, the distributions were very similar to one another and did not have many high F-Beta scores. However, the trends in the distribution of the scores did not appear to be meaningfully different from the plots presented in the paper and generally demonstrate a higher quantity of upper F-Beta scores than lower.

The plots can also used to assess the overfitting of the optimized model. Although it is generally expected to have the training performance much better than of validation and testing, we see that the difference between the distribution across datasets are less clearly defined than of the other models described in the paper.

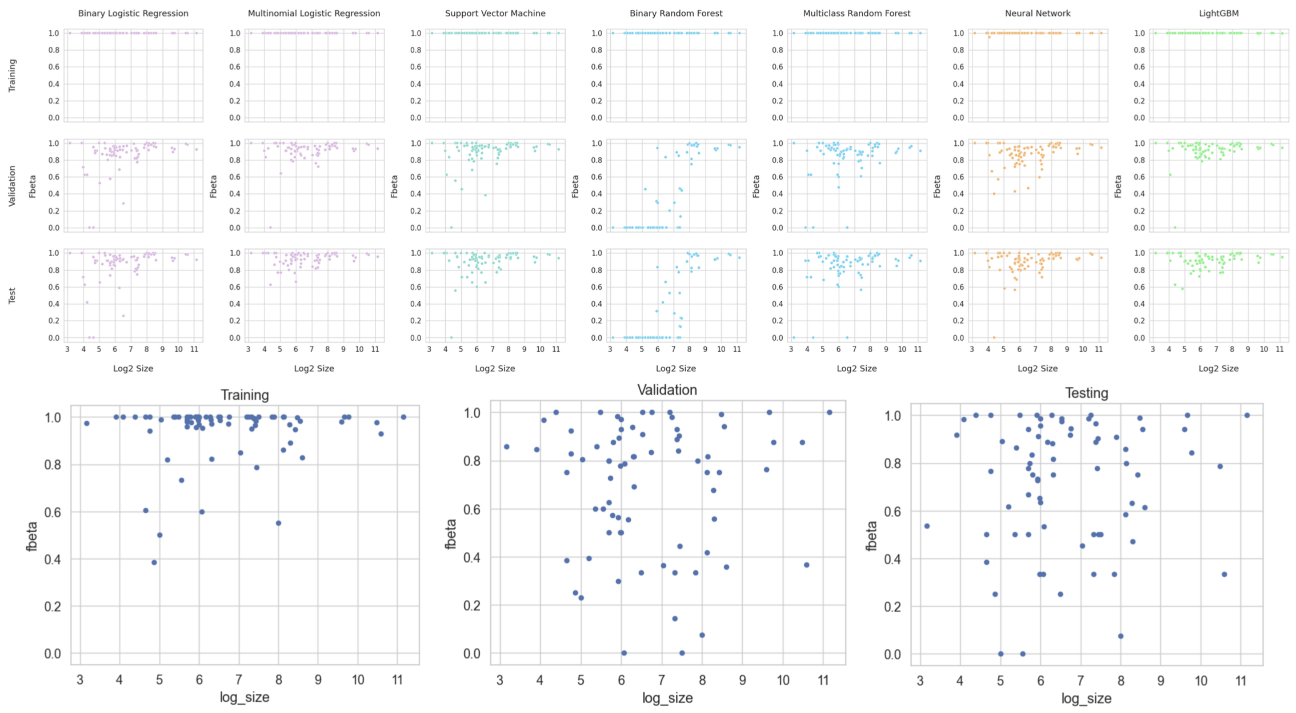

Figure 3: (A) Boxplots of the F-beta score distributions from the paper, (B) Boxplots of the F-beta scores on our own CNN model

To further assess the effect of overfitting, we conducted the following analysis with the Wilcoxon signed-rank test. We were able to demonstrate that similarly to the other models, CNN_1D had significant differences in the default training vs default validation and the optimal training vs optimal validation. When it came to comparing the default validation vs optimal validation, our model once again fell within similar ranges of the different machine learning models. This may be due to the limited hyperparameter optimization that was conducted. We were also able to determine our model does not appear to be overfitting the data by comparing our optimal validation and optimal test and finding no significant differences. Overall, this suggests our model is comparable with regard to supervised machine learning methods presented in the study.

Table 3: Performance comparison of models

Model	**Default Train vs. Default Val**	**Optimal Train vs. Optimal Val**	**Default Val vs. Optimal Val**	**Optimal Val vs. Test**
lr	3.50E-12	1.64E-12	0.795829538	0.920981475
lr_multi	1.10E-11	1.13E-13	1.13E-13	0.186445086
svm	3.55E-13	5.15E-12	0.00122993	0.807155438
rf	1.84E-14	1.84E-14	0.801590942	0.25840447
rf_multi	5.19E-13	2.40E-12	0.001373975	0.64697198
nn	5.21E-13	5.16E-12	2.53E-05	0.11984641
lgbm	1.13E-13	1.12E-12	5.28E-14	0.402887032
cnn_1D	3.88E-13	1.64E-12	0.248650256	0.124677149

Finally, when comparing our F-Beta scores with the models presented in the study, it is clear our model only outperforms binary random forest. Our distribution of scores is quire spread out in comparison and generally has a lower median F-Beta score than the other models presented. As mentioned previously, we believe further hyperparameter optimization and training trials could result in an improved model. However, our median accuracy is still reasonable and it shows the effectiveness of the other supervised machine learning models presented.

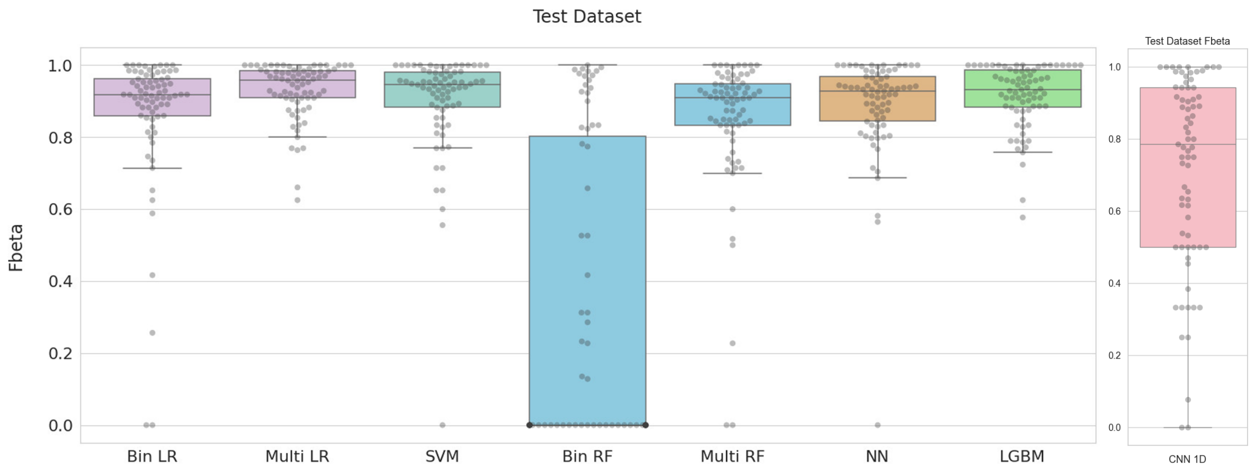

Figure 4: Boxplots of the F-beta score distributions from the paper and our own CNN model

3 Conclusion

This study was proficient at addressing the need for scalable statistical methods in supporting cell type classification. It presents multiple supervised machine learning models that could be used for classification with Light GBM and multi-logistic regression as the best-performing options. Although the pre-processing is at times unclear, the steps and logic presented are easy to follow and relevant to the work conducted in this study.

Our analysis incorporates an extensive investigation of the performance of 1D convolutional neural networks on cell-type classification problems. We hypothesized that a CNN architecture may provide benefits on feature selection, both local and global pattern learning. With a high level of features within our dataset, a CNN model may not depend on a manual filtering process within the preprocessing stage of the pipeline. In addition, with the right depth and hyperparameters, a CNN model can detect basic and complex patterns with the features, which may be valuable to differentiate closely related cell types.

Although our model did not outperform the original paper's best two models, it showed notable potential and outperformed binary random forest in our tests. From the statistical analysis, we deduced that our default model relies on COV threshold filtering and that the model's own feature selection does not outperform the preprocessing steps. After optimization, we compared F-beta scores between the default and optimized model and discovered that the parameters were not dissimilar and that a larger hyperparameter space is needed for training and optimization. From further analysis of the performance of the optimized model on all training, validation and testing datasets, we see that the model did not overfit as the other models attempted by the original paper. This further confirms that the model requires better optimization to increase the complexity of the model and capture more discrete patterns of the dataset. With regularization and dropout in place, the model was robust towards overfitting. Through our analysis, we can confirm the importance of pre-processing steps and hyperparameter tuning to improve the performance of machine learning models in biological data classification tasks.

4 References

1. Le, H., Peng, B., Uy, J., Carrillo, D., Zhang, Y., Aevermann, B. D., & Scheuermann, R. H. (2022). Machine learning for cell type classification from single nucleus RNA sequencing data. PLOS ONE, 17(9), e0275070. https://doi.org/10.1371/journal.pone.0275070

enumerate