CellPhe user guide

The CellPhe toolkit comprises three main stages: cell boundary coordinate extraction, feature extraction and selection, and data analysis. These stages are implemented in Matlab, C/C++ and R respectively (**Figure 1**), with stages allocated to separate software to take advantage of each software's strength. The speed of image processing using image libraries, made C/C++ ideal for the extraction of features from time-lapse images. The use of R for downstream analysis was motivated by the large repository of data analysis packages, including classification and clustering algorithms. This manual aims to guide users through the complete CellPhe workflow with a reproducible working example.

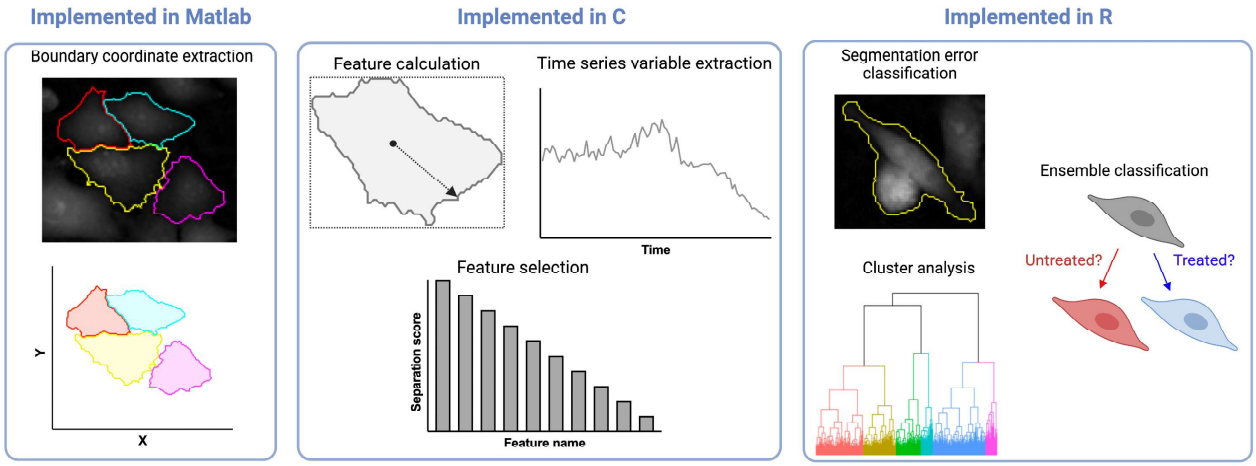


Figure 1: Stages of the CellPhe workflow, grouped according to the software used for implementation.

1 Pre-requisites

As well as the listed software (Matlab, C and R), users must have a C and C++ compiler, and required TIFF image libraries installed to execute the provided codes in full. Installation will differ depending on the user's operating system. Full instructions for code implementation are provided in GitHub. In addition to the source code, we have provided example output files at each stage of the CellPhe workflow so that the user can follow the working example included in the manual without needing to install compilers or TIFF libraries. A complete list of the files/-folders that can be found at https://github.com/llwiggins/CellPhe.git together with their descriptions, is given below.

Files for boundary coordinate extraction (Section 2):

- `ReadImageJROI.m`: MATLAB script for reading ROI files
- `ROIs2Coordinates.m`: MATLAB script for extracting boundary coordinates from ROI files

Files for extraction of features and time series variables (Section 3):

- `05062019_B3_3_Phase-FullFeatureTable.csv`: feature table obtained from Phasefocus' Cell Analysis Toolbox® software, note that similar feature tables could be obtained using alternative cell segmentation and tracking software
- `05062019_B3_3_boundaries.csv`: output file of extracted boundary coordinates, as described in **Section 2**
- `05062019_B3_3_imagedata`: a folder containing all grayscale 8-bit TIFF images to process, one TIFF image per time-lapse frame
- `05062019_B3_3_imagelist.txt`: a file containing an ordered list of images with the path to their location
- `image_info`: folder containing necessary C/C++ files for image processing
- `extract.c`: C code for feature calculation and time series extraction
- `constants.h`: file of customisable constants to tailor analysis to different experimental conditions
- `loadimage.cc and loadimage.h`: C++ and header files required for loading images
- `Makefile`: the main Makefile required for compilation of the code, as demonstrated in **Section 3**

Files for feature selection (Section 4):

- `VarSelect.c`: C code for calculation of separation scores, as demonstrated in **Section 4**
- `TreatmentClassify_trainingA.txt` and `TreatmentClassify_trainingB.txt`: full training set feature tables (obtained as in **Section 3)**, separated according to true class label - A for untreated cells, B for treated cells

Files for data analysis (Sections 5, 6 and 7):

- `SegmentationErrors_training.txt` and `CorrectSegmentation_training.txt`: full feature tables of ground truth segmentation errors and correctly segmented cells, used for training decision trees in **Section 5**.
- `Segmentation_testset.txt`: full feature table of test set cells, used for prediction of segmentation errors in **Section 5**.
- `TreatmentClassification_training.txt`: full feature table of training set cells (ground truth segmentation errors removed) used for ensemble classification in **Section 6**
- `separationscores.txt`: output file of separation scores obtained using the `VarSelect.c` code in **Section 4**, used for feature selection in **Section 6**

2 Extraction of cell boundary coordinates

Ordered cell boundary pixels, used for calculation of shape and texture features, can be extracted from the file of segmented cell ROIs (`05062019_B3_3_Phase.zip`) using the following MATLAB code:

```
>> cvsROIs = ReadImageJROI('05062019_B3_3_Phase.zip');
>> boundaries = ROIs2Coordinates(cvsROIs);
>> dlmwrite('05062019_B3_3boundaries.csv', boundaries)
```

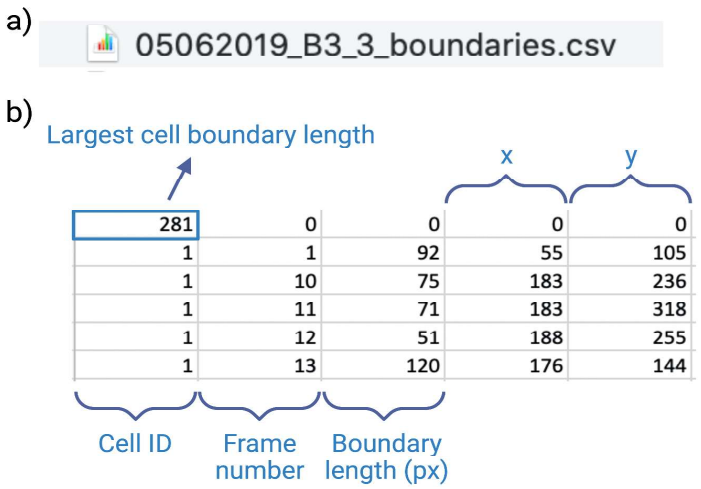


Figure 2: Output following execution of MATLAB code. a) 05062019_B3_3_boundaries.csv, the produced output file. **b)** The format of 05062019_B3_3_boundaries.csv. The value in the top-left corner is the largest cell boundary length in pixels (largest cell perimeter) within the data set. The remainder of the first row is just filled with zeros and can be ignored. For remaining rows, the first column lists cell IDs, the second frame numbers and the third each cell's boundary length in pixels. Remaining columns list x and y coordinates alternately until all boundary coordinates have been listed.

3 Extraction of features and time series variables

3.1 Adjusting constants

Constants such as image width/height, number of frames in a time-lapse, and the minimum track length for cells to be considered, can be customised within the constants file (`constants.h`). Ensure that all necessary changes to the constants file have been made prior to compilation.

3.2 Code execution

The feature extraction code expects 8-bit grayscale TIFF images as input. Other formats can be converted using image processing programs such as ImageJ (https://imagej.nih.gov/ij/). An ordered list of the images to be used should be placed in the directory with the code and include path information. This image file list, the boundary file obtained using Matlab and the feature table file are read in and should all have the same start to the name, e.g. `05062019_B3_3` as in our example. This allows the program to be run using only this and a class identifier as command line arguments. Filenames are completed by the code which adds `_imagelist.txt`, `_boundaries.csv` and `_Phase-FullFeatureTable.csv` as required.

A flowchart giving an overview of the feature extraction code in shown in **Figure 3**.

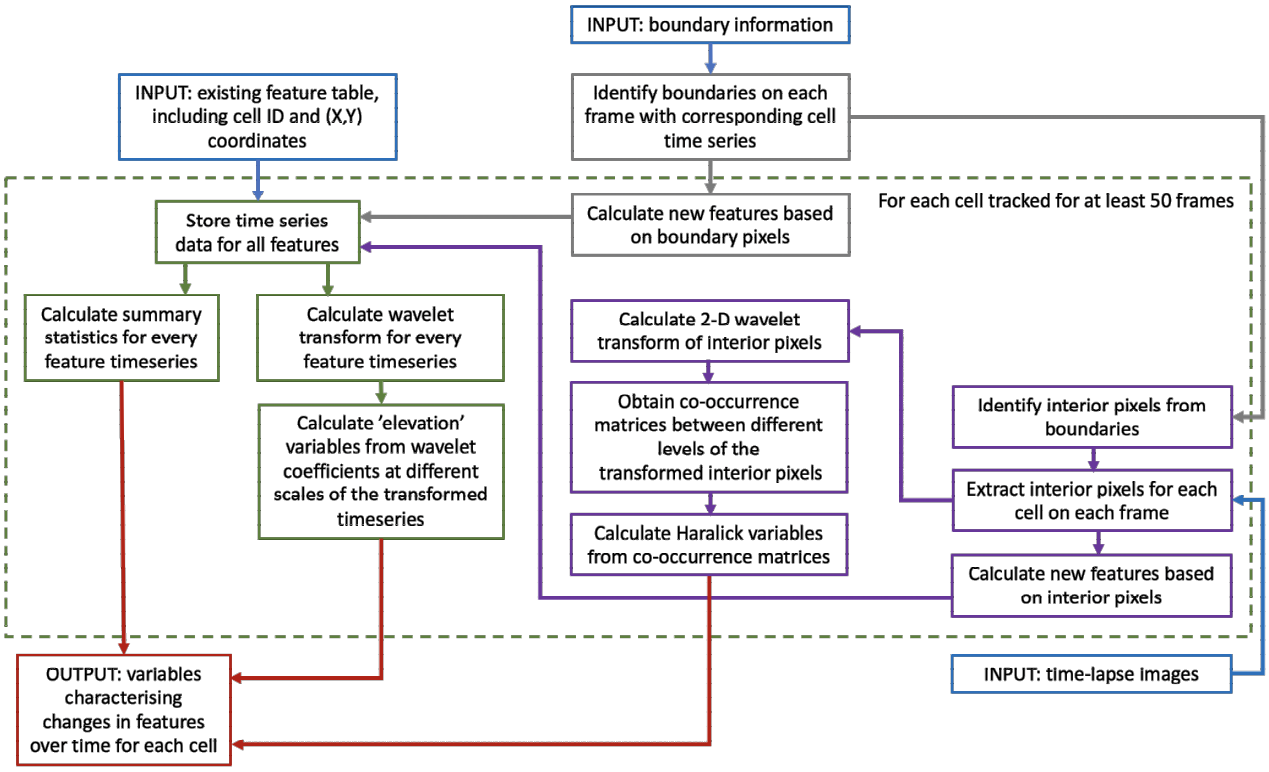


Figure 3: Flowchart giving an overview of the feature extraction code.

Instructions for installing, compiling and running the C code are given in README.md, available on GitHub (https://github.com/llwiggins/CellPhe.git) in the "Extraction of features and time series variables" folder.

CellIDs will be printed consecutively to indicate where the code is up to, with the final listed CellID being the total number of cells tracked.

a)

b)

Group	cellname	frames	trajarea	vol_asc	vol_des
B	05062019_B3_32	481	0.31033	5.890407	-5.844845
B	05062019_B3_33	144	55.656051	19.485202	-21.533468
B	05062019_B3_35	481	3.371012	11.00664	-9.117712
B	05062019_B3_36	230	1.351267	7.30574	-7.338793
B	05062019_B3_37	155	7.139714	37.592969	-22.822149
B	05062019_B3_3214	162	3.205177	17.286295	-14.561709
B	05062019_B3_38	138	0.908973	17.674467	-16.803368
B	05062019_B3_39	94	7.236885	42.913556	-36.983957

Figure 4: Output following code execution. a) 05062019_B3_3outputdata.txt, the produced output file. **b)** The format of 05062019_B3_3outputdata.txt. The first column is a list of true class labels, in this case all cells are from group "B", the treated set. The second column lists concatenations of data file and cell ID, e.g. 05062019_B3_32 references cell 2 from the 05062019_B3_3 data set. The third column lists the number of frames each cell has been tracked for, and the remaining columns list the complete set of variable scores for each cell.

4 Feature selection

The output file from **Section 3** contains the full list of 702 variables for each cell tracked for 50 frames or more. Although many of these variables will provide good discrimination between untreated and treated cells, a greater proportion of them will be redundant and therefore need to be filtered out prior to classification. Our `VarSelect.c` code can be executed from the command line to calculate a separation score for each variable. The higher the separation score, the more discriminatory the variable is. The following needs to be entered on the command line to begin execution of the C code:

```
$ gcc -c VarSelect.c
$ gcc -o output VarSelect.o
$ ./output TreatmentClassify_trainingA.txt TreatmentClassify_trainingB.txt
```

Where `TreatmentClassify_trainingA.txt` and `TreatmentClassify_trainingB.txt` are full training set feature tables for untreated and treated cells respectively.

The following text will then appear, prompting the user to enter the total number of variables, the number of cells in `TreatmentClassify_trainingA.txt` and the number of cells in `TreatmentClassify_trainingB.txt`:

```
Please enter nvars,nsamples1,nsamples2:
```

In our example, the total number of variables is 702, and the number of cells in training set A and B are 646 and 600 respectively. Note that these numbers need to be entered exactly as shown below, separated by commas and with no spaces in between:

```
702,646,600
```

If all has been entered correctly and the input files have been found and successfully opened for reading, the following text will appear and an output file called `separationscores.txt` will be written and saved in the current working directory:

```
opened TreatmentClassify_trainingA.txt
opened TreatmentClassify_trainingB.txt
```

A threshold on the separation score can then be used to select the most discriminatory features.

a)

b)

0	trajarea	0.077094
1	vol_asc	0.000621
2	vol_des	0.000067
3	vol_max	0.000197
4	vol_l1_asc	0.001005
5	vol_l1_des	0.000106
6	vol_l1_max	0.000163
7	vol_l2_asc	0.000348
8	vol_l2_des	0.000002
9	vol_l2_max	0.000001
10	vol_l3_asc	0.000029
11	vol_l3_des	0.00028
12	vol_l3_max	0.000188
13	rad_asc	0.338259
14	rad_des	0.285402
15	rad_max	0.086008
16	rad_l1_asc	0.172595
17	rad_l1_des	0.149049
18	rad_l1_max	0.079362
19	rad_l2_asc	0.137877
20	rad_l2_des	0.070628

Variable index | Variable name | Separation score

Figure 5: Output following execution of VarSelect.c. a) separationscores.txt, the produced output file of separation scores. **b)** The format of separationscores.txt. The first column is a list of variable indices, the second a list of variable names, and the third a list of calculated separation scores.

5 Segmentation error removal

Ground truth instances of correct segmentation and segmentation errors are required for classification training. An example feature table of correctly segmented cells and segmentation errors can be found in the files `CorrectSegmentation_training.txt` and `SegmentationErrors_training.txt` respectively.

```
## import files containing ground truth examples of correct segmentation and segmentation errors
Segerrortraining <- read.delim("~/Desktop/DataForCellPhe/SegmentationErrors_training.txt")
Correctsegtraining <- read.delim("~/Desktop/DataForCellPhe/CorrectSegmentation_training.txt")

## collate correct segmentation and segmentation error data into one data frame
data = rbind(Segerrortraining, Correctsegtraining)
segdata<-Segerrortraining[,-c(1,2,3)]
nonsegdata<-Correctsegtraining[,-c(1,2,3)]
alldata = rbind(segdata, nonsegdata)

## first column lists ground truth data labels
class1 = rep("segerror", dim(segdata)[1])
```

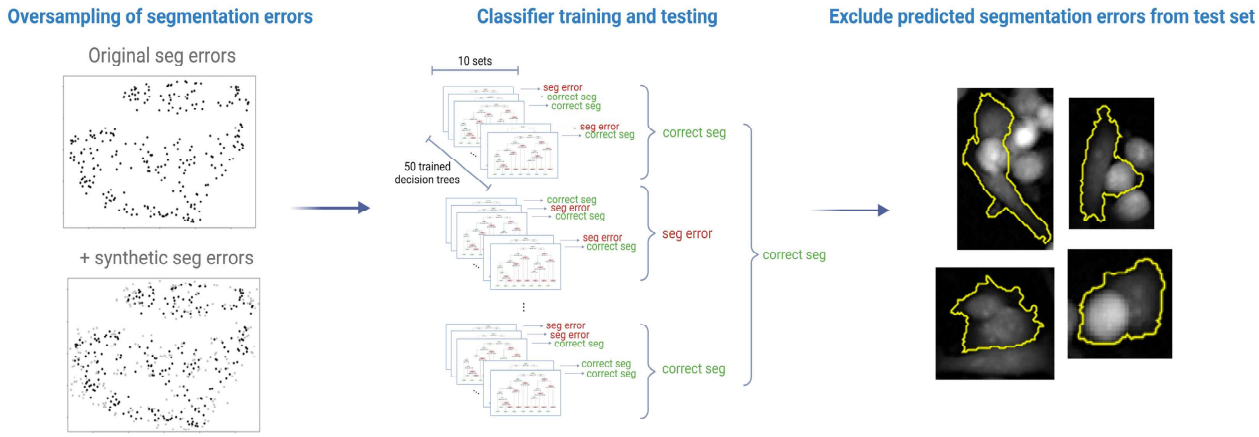


Figure 6: Summary of R code objectives for **Section 5**

```
class2 = rep("correct", dim(nonsegdata)[1])
class = c(class1, class2)
all = data.frame(class, alldata)
```

Once data tables have been imported into R, the `SMOTE()` function from the `smotefamily` package can be used to handle class imbalance by synthetically over-sampling segmentation errors.

```
## install smotefamily package
install.packages('smotefamily')
library(smotefamily)

## SMOTE() to over-sample segmentation errors
smoteseg<-SMOTE(all[,-1], all$class, K = 3, dup_size = 1)
```

Here `K` refers to the number of neighbours that should be considered during the sampling process, and `dup_size` is the number representing the desired times of synthetic minority instances over the original number of majority instances. A feature table of synthetically generated segmentation errors can be found in `smoteseg$syn_data`, these observations can now be added to the training set for use during classifier training.

```
## add smote segmentation errors to the training sets and update segmentation error data table
smotesegsyn_data<-smoteseg$syn_data[,c(703, 1:702)]
all<-rbind(all, smoteseg$syn_data)
segdata<-rbind(segdata, smotesegsyn_data[,-1])
newclass<-c(class, smotesegsyn_data[,1])
class1 = rep("segerror", dim(segdata)[1])
```

Two custom functions were produced for training and testing of multiple decision trees to detect segmentation errors. Individual decision trees were trained using the `tree()` function from the `tree` package.

The first function, `multitree`, is used for training the decision trees.

```
## install tree package
install.packages('tree')
library(tree)

## define multitree function
multitree = function (smalldata, bigdata, smallclass, bigclass, num)
{
  n1 = length(smallclass)
  n2 = length(bigclass)
  treelist = list()
  for (i in 1:num)
  {
    inds = sample.int(n2, n1)
    data = rbind(bigdata[inds,],smalldata)
    class = c(bigclass[1:n1], smallclass)
    data = data.frame(class, data)
    mytree = tree(as.factor(class)~., data=data)
    treelist[[i]] <- mytree
  }
  return(treelist)
}
```

The second function, `multipred`, is used for making predictions from the decision trees.

```
## define multipred function
multipred = function (data, trees, num)
{
  pred = matrix(" ", nrow = num, ncol = nrow(data))
  for (i in 1:num)
  {
    x = predict(trees[[i]], data, type = "class")
    pred[i,] <- x
  }
  return(pred)
}
```

These two functions can be called as follows to train 50 decision trees and acquire a list of predicted labels for each test set cell:

```
## train 50 decision trees using training set
trees = multitree(segdata, nonsegdata, class1, class2, 50)

## import test set and classify
Segmentationtestset <- read.delim("~/Desktop/DataForCellPhe/Segmentation_testset.txt")
alltest<-Segmentationtestset
predtest = multipred(alltest[,-c(1,2,3)], trees, 50)
```

Here we trained 50 decision trees but this number can be redefined by the user. To avoid the removal of correctly segmented cells misclassified as segmentation errors, we used a voting system for final predictions where cells were classified as a segmentation error if they received ≥ 35, calculated by $\frac{(3.5 \times 50)}{5}$, votes for this, otherwise they were classified as correct segmentation.

```
## obtain list of cell names for cells that were predicted as segmentation error
testnumseg = vector(mode = "integer", length = ncol(predtest))
for (i in 1:ncol(predtest))
{
  testnumseg[i] = length(which(predtest[,i] == 2))
}

testvote = rep("nonseg", length = ncol(predtest))

## decision tree votes combined to produce final predicted label for each cell
## (>= 35/50 votes for segmentation error)
for (i in 1:ncol(predtest))
{
  if (testnumseg[i] > 3.5 * nrow(predtest)/5) testvote[i] = "seg"
}

ind = which(testvote == "seg")
segtest = Segmentationtestset$cellname[ind]

## save predicted segmentation error cell names as a .txt file
write.table(segtest, "Predictedsegerrors.txt", row.names = F)
```

To add further stringency, the training of 50 decision trees can be repeated several times and a cell given a final classification of segmentation error if it receives above a set percentage of votes for this class. The following code demonstrates this by training ten sets of 50 decision trees, and providing a cell a final class label of segmentation error if it was classified as such in at least 5 of the sets.

```
## create empty list to store decision tree votes for each cell
votes<-list()

## train ten sets of 50 decision trees
## votes[[i]] contains predictions from the ith set of decision trees
for(i in c(1:10))
{
  trees = multitree(segdata, nonsegdata, class1, class2, 50)
  predtest = multipred(alltest[,-c(1,2,3)], trees, 50)
  testnumseg = vector(mode = "integer", length = ncol(predtest))
  for (j in 1:ncol(predtest))
  {
    testnumseg[j] = length(which(predtest[,j] == 2))
  }

  testvote = rep("nonseg", length = ncol(predtest))
  for (j in 1:ncol(predtest))
  {
    if (testnumseg[j] > 3.5* nrow(predtest)/5) testvote[j] = "seg"
  }

  ind = which(testvote == "seg")
  segtest = Segmentationtestset$cellname[ind]
  print(i)
  votes[[i]]<-segtest
}
```

```
## identifies cells that receive >= 5 votes for segmentation error
## and stores these in segtest
list<-NULL
for(i in c(1:10))
{
  list<-c(list, as.character(votes[[i]]))
}

segtest<-vector()
j = 1
freqdata<-as.data.frame(table(list))
for(i in c(1:dim(freqdata)[1]))
{
  if(freqdata$Freq[i] >= 5)
  {
    segtest[j] = as.character(freqdata$list[i])
    j=j+1
  }
}

## save predicted segmentation error cell names as a .txt file
write.table(segtest, "Predictedsegerrors.txt", row.names = F)

## remove predicted segmentation errors from test set ahead of treatment classification
FullTestSet<-subset(Segmentationtestset, Segmentationtestset$cellname %in% segtest == FALSE)
```

6 Ensemble classification

Now that predicted segmentation errors have been excluded from the test set, this can be used together with the training set to train and test an ensemble of classifiers for untreated vs. treated cell classification. We have already performed feature selection in **Section 4** so we can import the obtained file `separationscores.txt` and use this to include only the features that achieved separation above a fixed threshold.

```
## import training set (identified segmentation errors removed)
FullTrainingSet<-read.delim("~/Desktop/DataForCellPhe/TreatmentClassification_training.txt")

## scale data prior to classification
dataforscaling<-rbind(FullTrainingSet[,-c(1,2,3)], FullTestSet[,-c(1,2,3)])
dataforscaling<-scale(dataforscaling)
FullTrainingSet[,-c(1,2,3)]<-dataforscaling[c(1:nrow(FullTrainingSet)),]
FullTestSet[,-c(1,2,3)]<-dataforscaling[-c(1:nrow(FullTrainingSet)),]

## import file of separation scores
separationscores <- read.table("~/Desktop/DataForCellPhe/separationscores.txt", quote="\"",
comment.char="")

## subset separation scores file to only include variables with separation >= 0.2
chosenvars0.2<-subset(separationscores, separationscores$V3 >= 0.2)

## subset training and test sets to only include variables with separation >= 0.2
SelectedTrainingSet<-FullTrainingSet[,c((chosenvars0.2$V1)+4)]
```

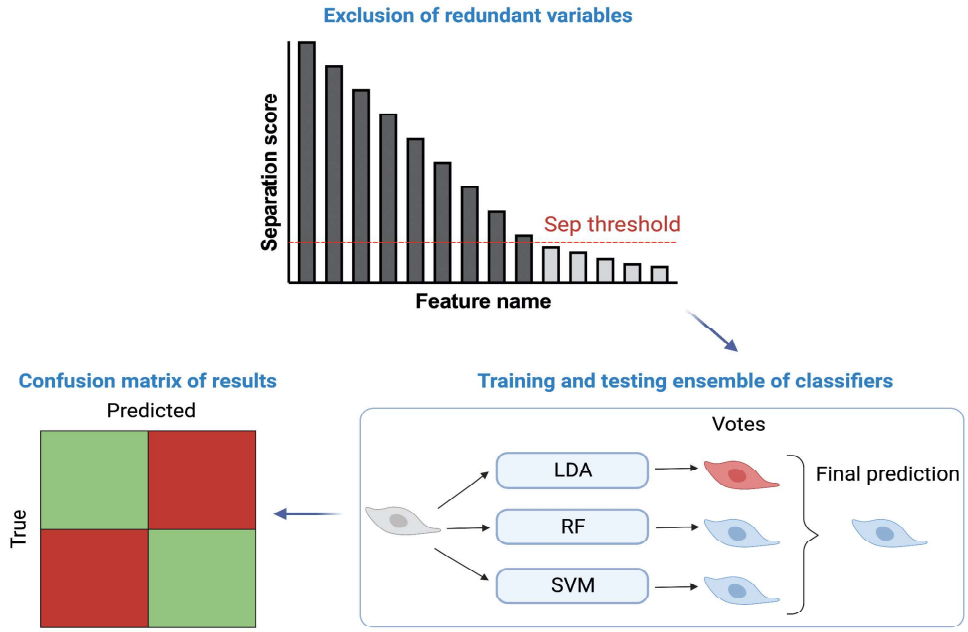


Figure 7: Summary of R code objectives for **Section 6**

```
SelectedTestSet<-FullTestSet[,c((chosenvars0.2$V1)+4)]
```

We use ensemble classification where LDA, RF and SVM classifiers are trained individually and their predictions combined to provide a final classification label for each cell based on majority vote. R packages `MASS`, `randomForest` and `e1071` can be used to train LDA, RF and SVM classifiers respectively.

```
## install packages used for classification
install.packages("MASS")
library(MASS)

install.packages("randomForest")
library(randomForest)

install.packages("e1071")
library(e1071)

## classifier training
ldamodel<-lda(SelectedTrainingSet, FullTrainingSet$Group)
rfmodel <- randomForest(FullTrainingSet$Group~., data = SelectedTrainingSet,
ntree=200, mtry=5, importance=TRUE, norm.votes = TRUE)
svmmodel<-svm(SelectedTrainingSet, FullTrainingSet$Group, kernel = 'radial', probability = TRUE)

## classifier testing
ldapred = predict(ldamodel, SelectedTestSet)
rfpred = predict(rfmodel, SelectedTestSet)
svmpred = predict(svmmodel, SelectedTestSet)
```

Once individual classifiers have made their test set predictions, these can be used as votes where a cell receives a certain final classification label if it was predicted as such by the majority of classifiers in the ensemble. In this case,

as we are using 3 classifiers in ensemble, a cell will receive the final classification label that 2 or more classifiers voted for.

```
## ensemble classification, final predicted label based on majority vote
classificationvotes<-cbind(as.character(ldapred$class),
as.character(rfpred), as.character(svmpred))
classificationvotes<-as.data.frame(classificationvotes)

for(i in c(1:dim(FullTestSet)[1]))
{
  if((classificationvotes[i,1]=="A" && classificationvotes[i,2]=="A")
  ||(classificationvotes[i,1]=="A"
  && classificationvotes[i,3]=="A")
  ||(classificationvotes[i,2]=="A" && classificationvotes[i,3]=="A"))
  {
    classificationvotes[i,4] = "A"
  }
  else
  {
    classificationvotes[i,4] = "B"
  }

}

colnames(classificationvotes)<-c("LDA", "RF", "SVM", "Ensemble")

## confusion matrix of ensemble classification results
table(true = FullTestSet$Group, predicted = classificationvotes$Ensemble)
```

7 Cluster analysis

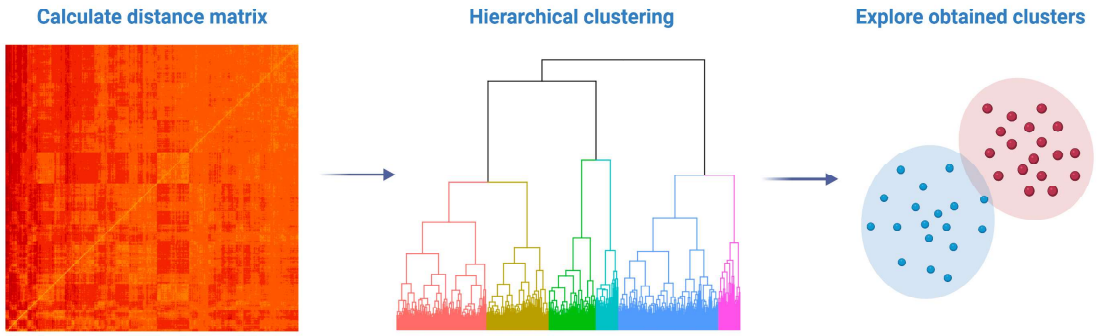


Figure 8: Summary of R code objectives for **Section 7**

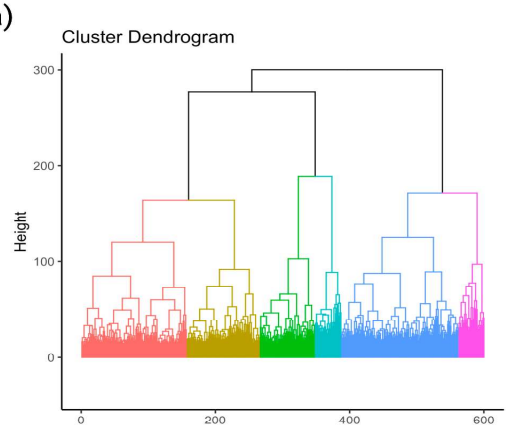
```
## install factoextra for hierarchical clustering and plotting dendogram
install.packages("factoextra")
library(factoextra)
```

```r
## subset of just training set treated cells
TreatedCells<-subset(FullTrainingSet, FullTrainingSet$Group == "B")

## distance matrix and hierarchical clustering
d<-dist(scale(TreatedCells[,-c(1,2,3)]), method = "euclidean")
hierclust<-hcut(d, hc_func = "agnes", hc_method = "ward.D", hc_metric = "euclidean", k = 6)

## plot the acquired dendogram
fviz_dend(hierclust, show_labels = FALSE)+theme_classic(base_size = 20)
```

a)



b)

```
> hierclust$size
[1] 157 109  82 175  39  38
```

Figure 9: Outputs obtained from running the R code provided in **Section 7**. **a)** Plot of acquired dendogram. **b)** Code to output the number of cells in each cluster.