Test

2024-02-18

Introduction

Loading and Preprocessing

Data is read from file and we can see what proportion of the features belong to which kind of omics data:

```
setwd("C:/Users/JIC/Documents/UNIPD/Stat Learning/proj")
omics <- read.table("data.csv", header = TRUE, sep=",")
rs.cols <- grep(paste0("^", "rs"), names(omics))
cn.cols <- grep(paste0("^", "cn"), names(omics))
mu.cols <- grep(paste0("^", "mu"), names(omics))
pp.cols <-grep(paste0("^", "pp"), names(omics))
print(paste("gene expression features:", length(rs.cols)))
```

```
## [1] "gene expression features: 604"
```

```
print(paste("copy number features:", length(cn.cols)))
```

```
## [1] "copy number features: 860"
```

```
print(paste("mutation features:", length(mu.cols)))
```

```
## [1] "mutation features: 249"
```

```
print(paste("protein abundance features:", length(pp.cols)))
```

```
## [1] "protein abundance features: 223"
```

Data types

Types of omics Distribution

Missing Value Imputation

Missing values in omics data is commonly a result of low quality reads that are removed. RS and PP are continuous variables, so it is very clear when information is missing, as indicated by a ‘0’ entry. There are two common methods for dealing with missing values: either exclude that example or feature from computation, or impute the missing values by replacing them with the features median value. Imputation introduces some bias,

but it is unavoidable in many cases such as this one, where dropping rows or features with missing values would not leave a usable amount of data. We calculate the rates of missing values for the two omics that take continuous values. It is most likely that the other forms of omics have missing values as well. Unfortunately, since they take discrete values, it is not possible to distinguish true ‘0’ readings from missing values. Thankfully 0 is the mode for these values(as we will demonstrate), so represents a reasonable imputation.

```
# Missing value imputation
# due to high dim, removing missing not an option; every row and col has missing values
rs <- omics[rs.cols]
pp <- omics[pp.cols]
rs.missing <- sum(apply(omics[rs.cols], 1, function(x) any(x == 0)))
rs.medians <- apply(rs, 2, function(x) median(x[x != 0]))
pp.medians <- apply(pp, 2, function(x) median(x[x != 0]))
# Missing value rate
rs.missing.rate <- sum(rs == 0) / (length(rs.cols) * length(rs))
rs.missing.rate
```

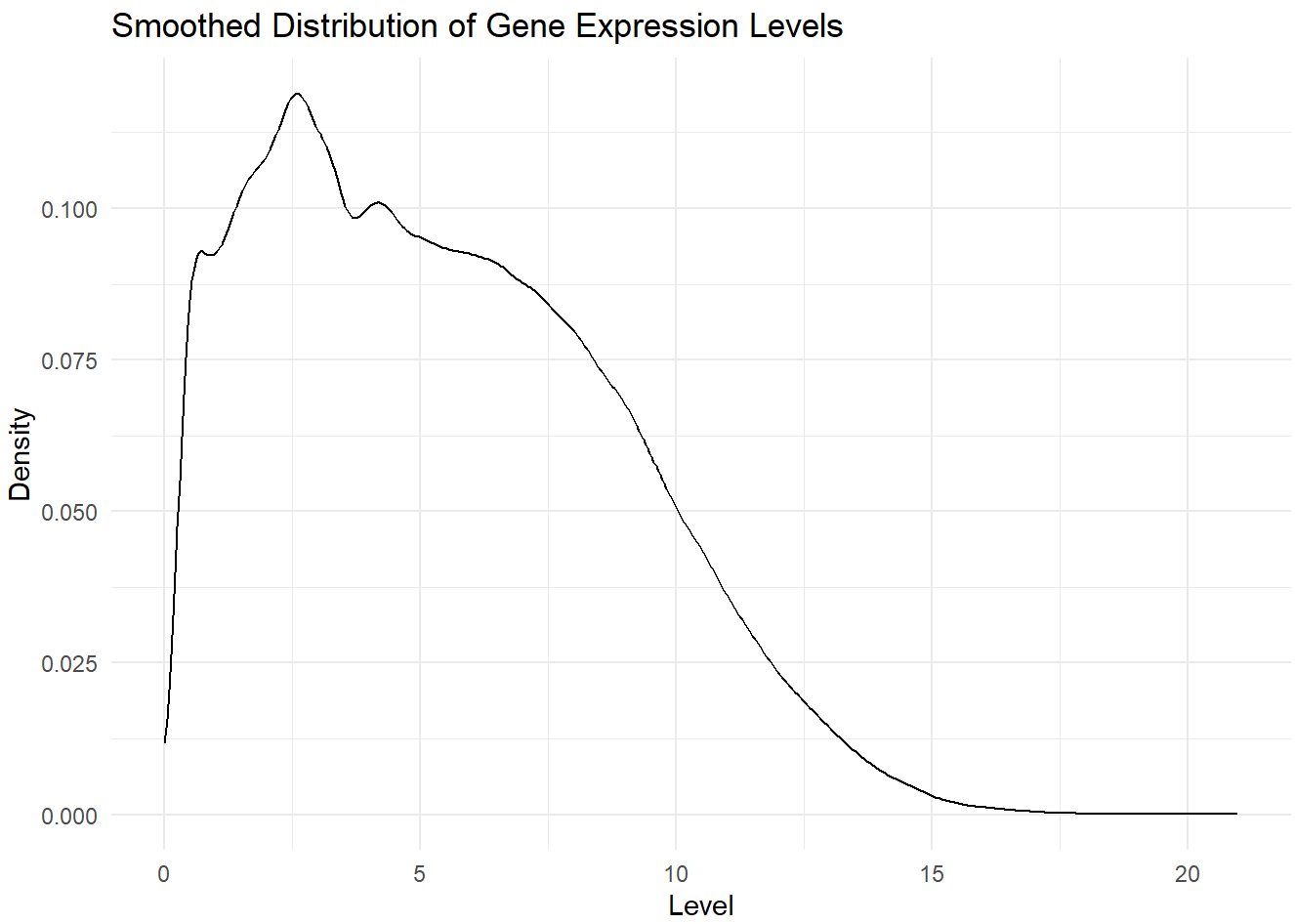
```
## [1] 0.09570578
```

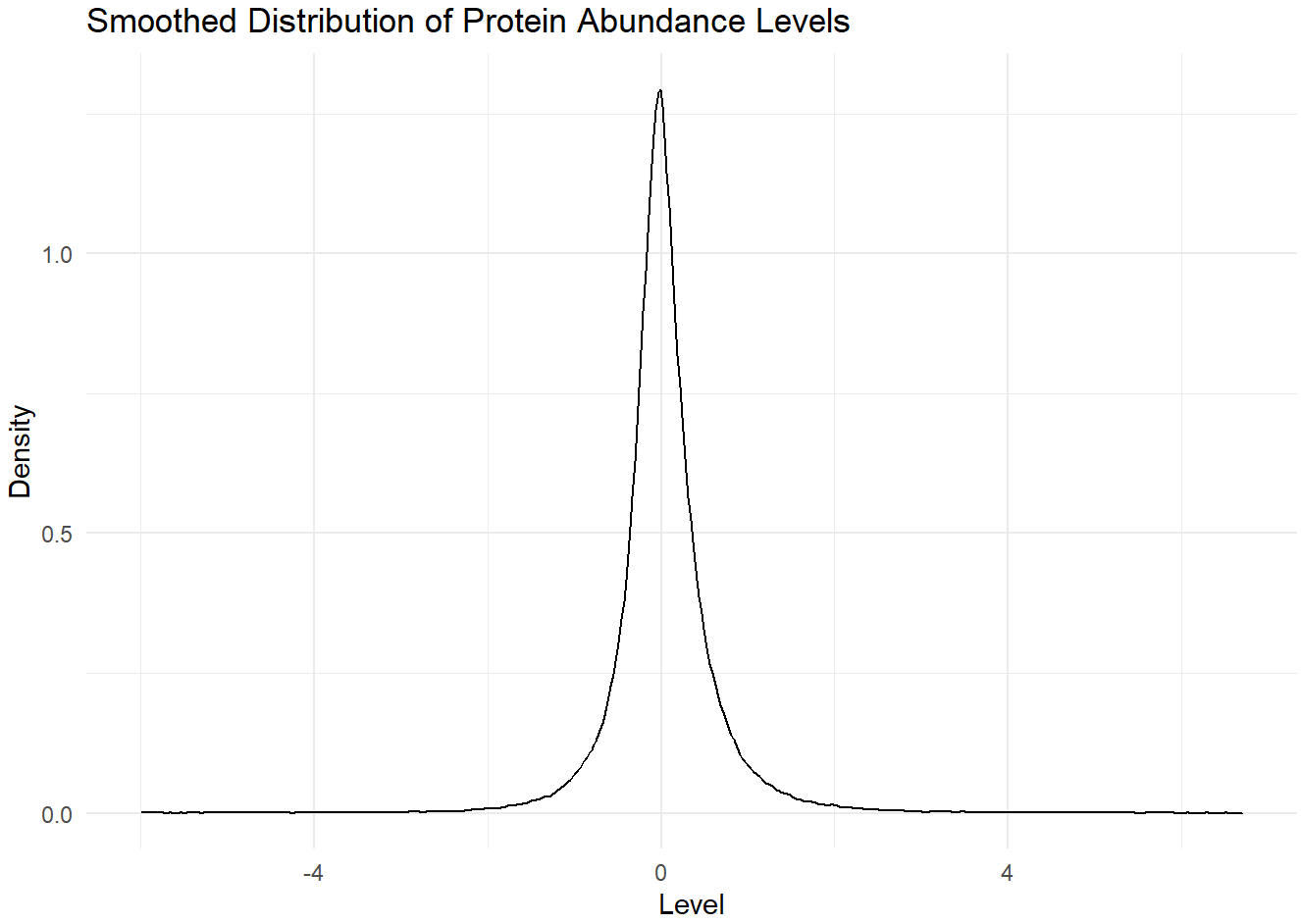
```
pp.missing.rate <- sum(pp == 0) / (length(pp.cols) * length(pp))
pp.missing.rate
```

```
## [1] 0.005751171
```

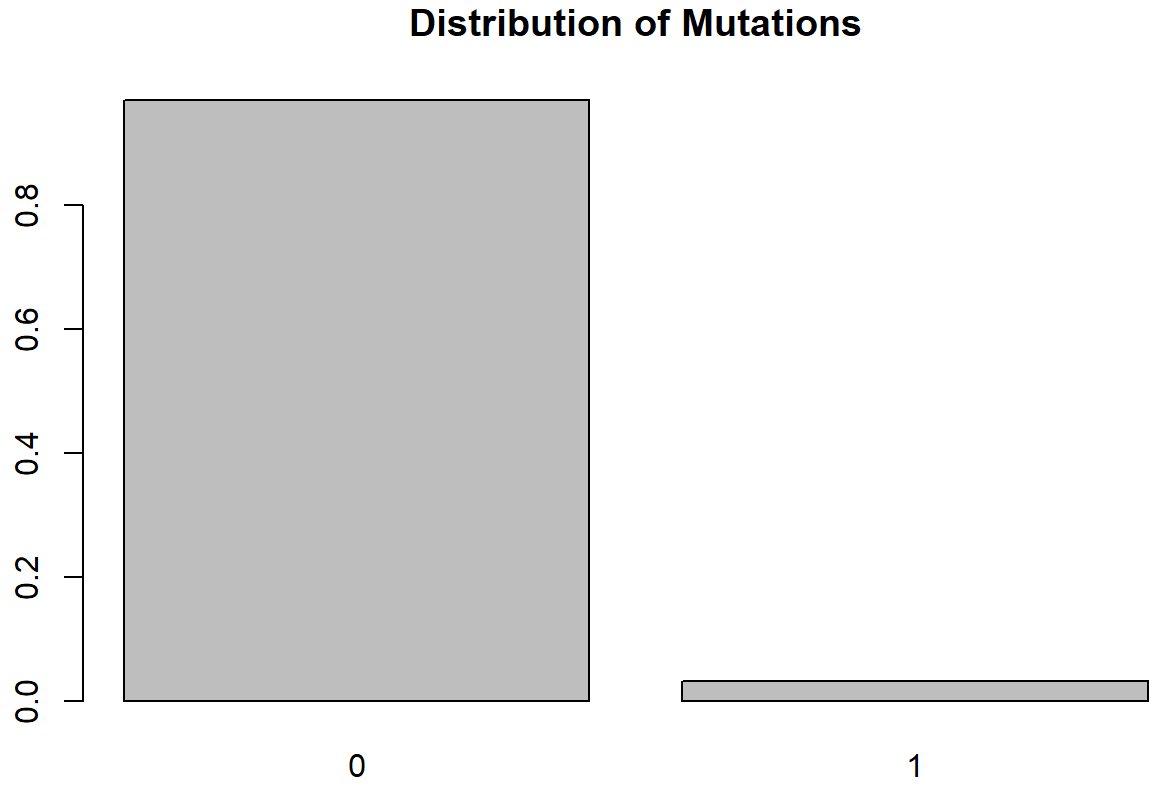
```
for(i in seq_along(rs.medians)) {
  rs[rs[, i] == 0, i] <- rs.medians[i]
}
for(i in seq_along(pp.medians)) {
  pp[pp[, i] == 0, i] <- pp.medians[i]
}
omics[c(rs.cols,pp.cols)] <- c(rs,pp)
```

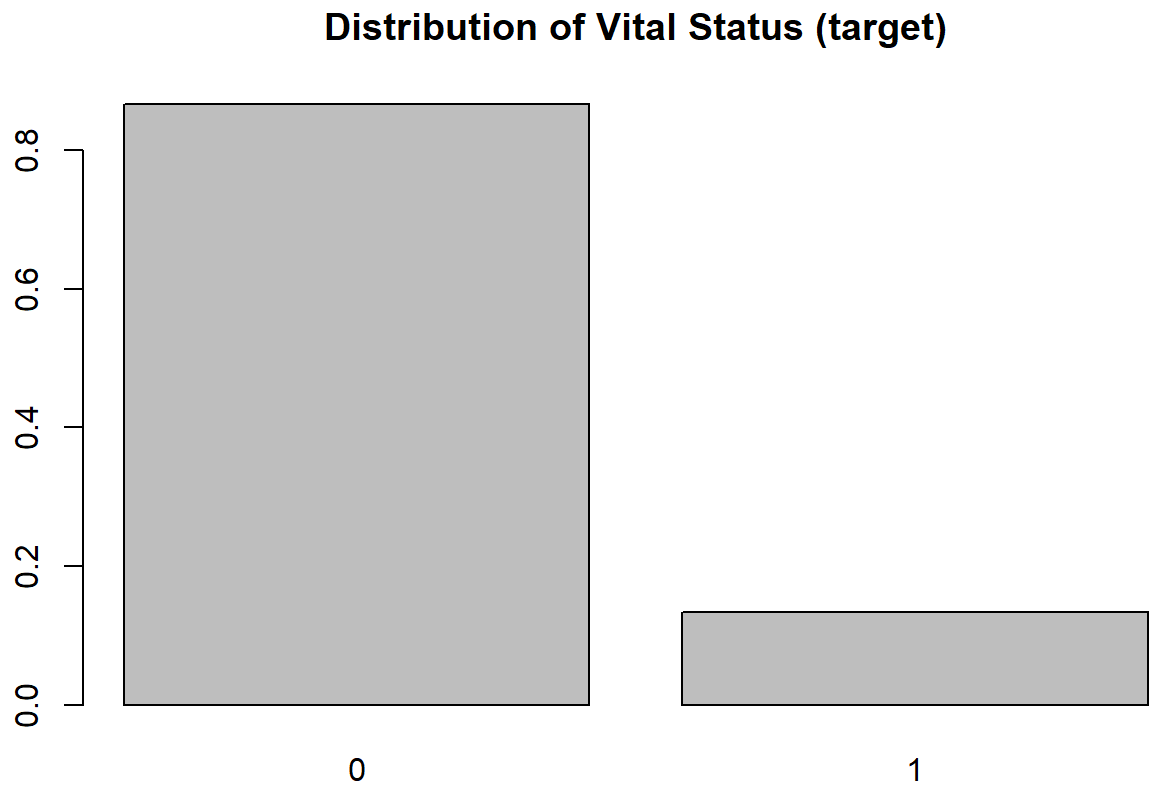
Here we visualize the distribution of values taken by all features of a given type of omics.
TODO: calculate population statistics for each omics type.
TODO: Remake the continuous distribution plots, something is wrong with them


Smoothed Distribution of Gene Expression Levels
Density
0.100
0.075
0.050
0.025
0.000
0
5
10
15
20
Level


Smoothed Distribution of Protein Abundance Levels
Density
1.0
0.5
0.0
-4
0
4
Level


Distribution of Copy Number
0.0
0.1
0.2
0.3
0.4
-2
-1
0
1
2


Distribution of Mutations
0.0
0.2
0.4
0.6
0.8
0
1



The final step of pre-processing is to transform the features to have zero mean and variance 1.
TODO: explain why this is important for logistic regression

```
omics_norm <- omics
omics_norm[c(rs.cols,pp.cols)] <- scale(omics[c(rs.cols,pp.cols)])
colMeans(omics)[1:5]
```

```
##  rs_CLEC3A    rs_CPB1 rs_SCGB2A2 rs_SCGB1D2    rs_TFF1
##   5.765444   6.778227   9.734685   7.644255   8.711891
```

Covariance analysis

Logistic regression

Definition

Full model(?) - maybe just go straight to k-fold CV

```
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 4.1-8
```

```
X <- as.matrix(omics_norm[,-ncol(omics_norm)]) # Convert to matrix for glmnet
y <- omics_norm$vital.status

# Split the data into training (70%) and testing (30%) sets
set.seed(123) # for reproducibility
sample_size <- floor(0.7 * nrow(omics_norm))
train_indices <- sample(seq_len(nrow(omics_norm)), size = sample_size)
X.train.niave <- X[train_indices, ]
y.train.niave <- y[train_indices]
X.test.naive <- X[-train_indices, ]
y.test.naive <- y[-train_indices]
glm.naive <- glm(y.train.niave~., data=data.frame(y.train.niave, X.train.niave), family=binom
ial)
```

```
## Warning: glm.fit: algorithm did not converge
```

```
#summary(glm.naive)
```

The model achieves 0 classification error on its training data.
Problem of dimensionality leading to overfitting allowing for perfect classification.
To demonstrate the issues with this model we have implemented a train/test split, where the model was trained on a subset of available examples and we can now evaluate it on the remaining test set.

```
pred.naive <- predict(glm.naive, newdata = data.frame(X.test.naive), type = "response")
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from rank-deficient fit; attr(*, "non-estim") has doubtful cases
```

```
pred.naive.classes <- ifelse(pred.naive > 0.5, 1, 0)
table(predicted = pred.naive.classes, actual = y.test.naive)
```

```
##          actual
## predicted  0  1
##         0 89 13
##         1 97 13
```

```
accuracy.naive <- sum(pred.naive.classes == y.test.naive) / length(y.test.naive)
accuracy.naive
```

```
## [1] 0.4811321
```

The model achives an accuracy of less than 50%, despite achieving excellent results on the training data! This clearly indicates overfitting and is a common problem with logistic regression where the number of features is

greater than the number of examples.
We can additionally observe a warning that our model failed to converge and many of the coefficients are reported as ‘NA’. This is not desired behavior as glm should return a “full” model where all features should participate. This is the result of the model achieving perfect classification error, and failing to fit additional features. As a result, we should aim to find a model that is a subset of the full model. This model should have less features, as to be successfully fit, but not too few that performance is overly compromised.
[You can choose to introduce subset models here or when you’re testing the different omics]
One thing that often makes logistic regression models useful is there intractability. Coefficients of a trained model serve as clear indicators of a features effect on a target variable. However, when there is close to 2000 such coefficients this ceases to be useful.
[the information about why it is more useful to have fewer features]
[Introduce LASSO]
[Introduce K-fold]

```r
# Note that glmnet has its own kfold functionality but we would like to analyze all models
# Need custom CV loop
library(glmnet)
set.seed(123) # For reproducibility

# Assuming omics_norm is your dataset
X.lasso <- as.matrix(omics_norm[,-ncol(omics_norm)]) # Convert to matrix for glmnet
y.lasso <- omics_norm$vital.status

# Number of folds for cross-validation
nfolds <- 4

# Create folds
folds <- sample(rep(1:nfolds, length.out = nrow(X)))

# Storage for fold-specific results
fold_results <- list()

# Init metric lists
accuracies <- numeric(nfolds)
classification_errors <- numeric(nfolds)

for (i in 1:nfolds) {
  # Split data into training and validation based on folds
  train_indices <- which(folds != i)
  test_indices <- which(folds == i)

  X_train.fold <- X.lasso[train_indices, ]
  y_train.fold <- y.lasso[train_indices]
  X_test.fold <- X.lasso[test_indices, ]
  y_test.fold <- y.lasso[test_indices]

  # Fit model on training data
  fit <- glmnet(X_train.fold, y_train.fold, family = "binomial", alpha = 1)

  # Optionally, use cv.glmnet to determine the best lambda within each fold
  cv_fit <- cv.glmnet(X_train.fold, y_train.fold, family = "binomial", alpha = 1, type.measur
e = "class")
  best_lambda <- cv_fit$lambda.min

  # Make predictions on the test set
  predictions <- predict(cv_fit, newx = X_test.fold, s = "lambda.min", type = "response")
  predicted_classes <- ifelse(predictions > 0.5, 1, 0)

  # Calculate accuracy and classification error
  correct_predictions <- sum(predicted_classes == y_test.fold)
  accuracy <- correct_predictions / length(y_test.fold)
  #classification_error <- 1 - accuracy

  # Store metrics
  accuracies[i] <- accuracy
```

```
  #classification_errors[i] <- classification_error

  # Store results
  fold_results[[i]] <- list(
    coefficients = coef(cv_fit, s = "lambda.min"), # Coefficients at best lambda
    best_lambda = best_lambda,
    cv_fit = cv_fit # You can also store the entire cv.glmnet object
  )
}

# Now, fold_results contains the detailed results from each fold,
# including the best lambda values and coefficients.
accuracies
```

```
## [1] 0.8983051 0.8579545 0.8352273 0.8806818
```

Stability Analysis (slopiness)

Another advantage of K-fold cross validation is it introduces noise into our data. Though this may sound like a bad thing, it is incredibly useful for performing sensitivity analysis. In this case, we can look at the features selected by the LASSO method by models trained on different subsets of the original data. Variation in parameters across models trained on different subsets of the original data is referred to as *sloppiness*. It can be common with data having high multicollinearity, as features can often be substituted with similar results. [compare the features selected from each fold]

metrics & evaluation

- accuracy
- R^2
- confusion matrix
- precision, recall F1, AUC

Subset analysis

for each omics type make model excluding and using only that tyoe, test means