Final Homework

G01 Team

A) Data and Goal

Data Loading/Pre-processing

Explain how we are going to handle the exercise and our first thoughts regarding the data, the mel-spectrum and the stuff we talked about on last monday.

TODO: comment plot and strategy

Build the CNN

Following Schreiber et al. we implemented a Convolutional Neural Network to pre-process and synthesize the mel-spectrum associated to each audio sample. As mentioned before, the mel-spectrum gives us the information regarding the frequencies composing the audio signal at each time interval ($t_i \in \{1, ..., 171\}$); the underlying idea is apply to this spectrum the same techniques that usually apply to the image processing tasks, in order to extract progressively patterns from the frequencies and finally have as output the predicted tempo and the associated probability distribution.

In doing so, the data samples flow across many different convolutional layers having different dimensional filters up until the output of the convolutions passes through the dense layers, which are in charge of produce the final prediction.

Our architecture slightly differs from the one in the reference in the dimension of the input and hence in the dimension of the filters: the mel-spectrum is given us into a flatten form, having the columns of the original matrix chained in a unidimensional vector; for this reason, the number of input nodes reflects the number of mel-features (40×171) and all the convolutional filters, which are unidimensional also in the reference paper, are applied only along one axis.

We implemented the CNN by means of the API of the libraries Keras and Tensorflow, relying on the pre-built layers. The model was trained in Google Colab, using the same settings reported in Schreiber et al.

Althought the CNN gives us very good results and a low RMSE, it lacks in interpretability - due also to the great number of parameters - and needs more than 8 epochs (on average) to be trained well. Even if feasible to tackle, these challenges pushed us to decide to early stop the training - the elected model has been trained for 4 (?) epochs - and to make use of the CNN only for a data pre-processing step - synthesize the mel-frequencies spectrum -. Thus, the CNN's predictions were added to the data set as a replacement of the mel-spectrum, leading to a consistent dimensionality reduction.

```
cat("File_Name,TrainRMSE,ValRMSE\n", file="results.csv")
list_dir = list.files('/content')

# Loop over all the CNN models saved in list_dir
for (file in list_dir[5:length(list_dir)]) {

    # Get model identifier from the file name itself
    str.id = str_sub(file, 8, 10)
```

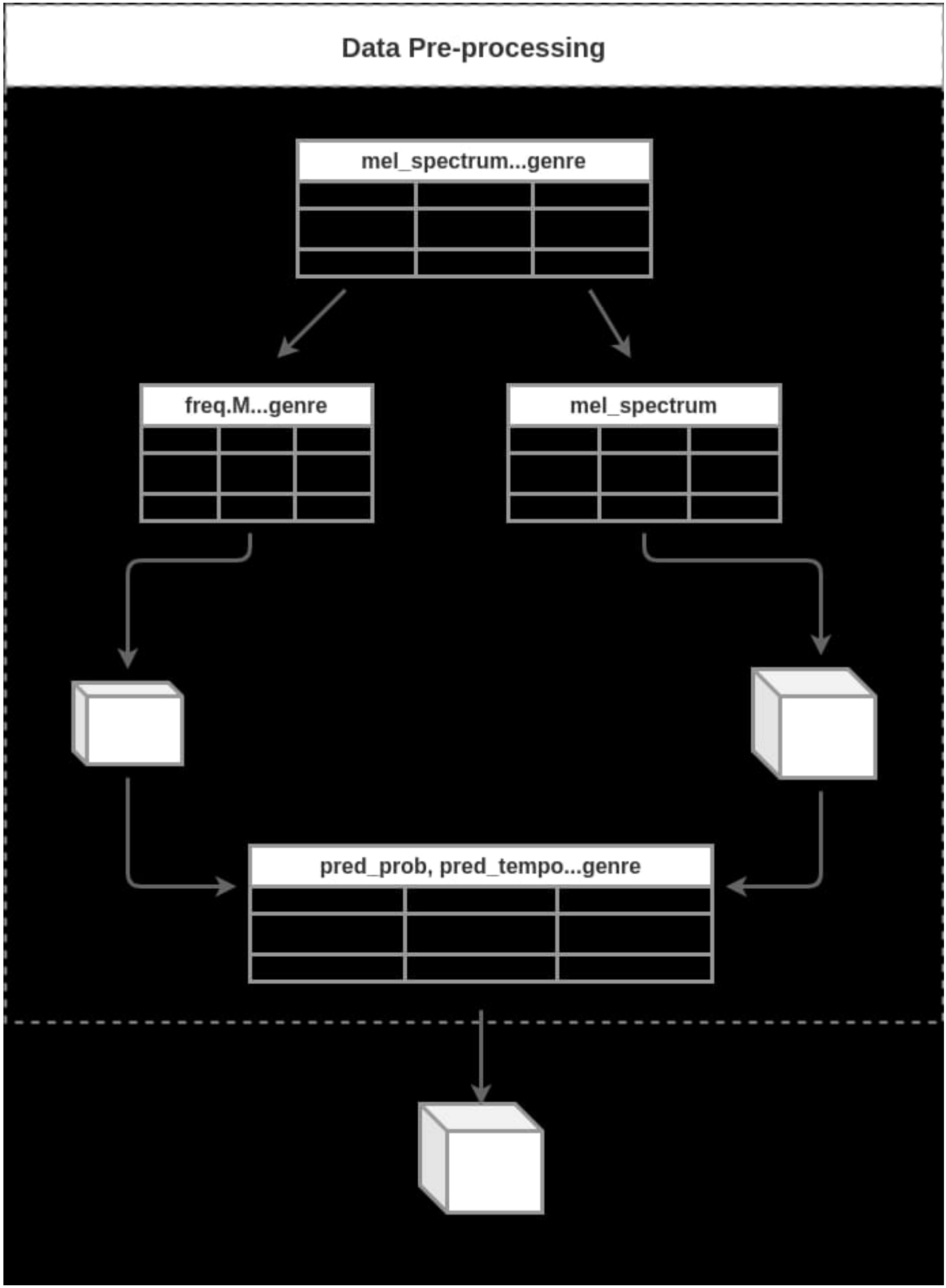


Figure 1: Pipeline.

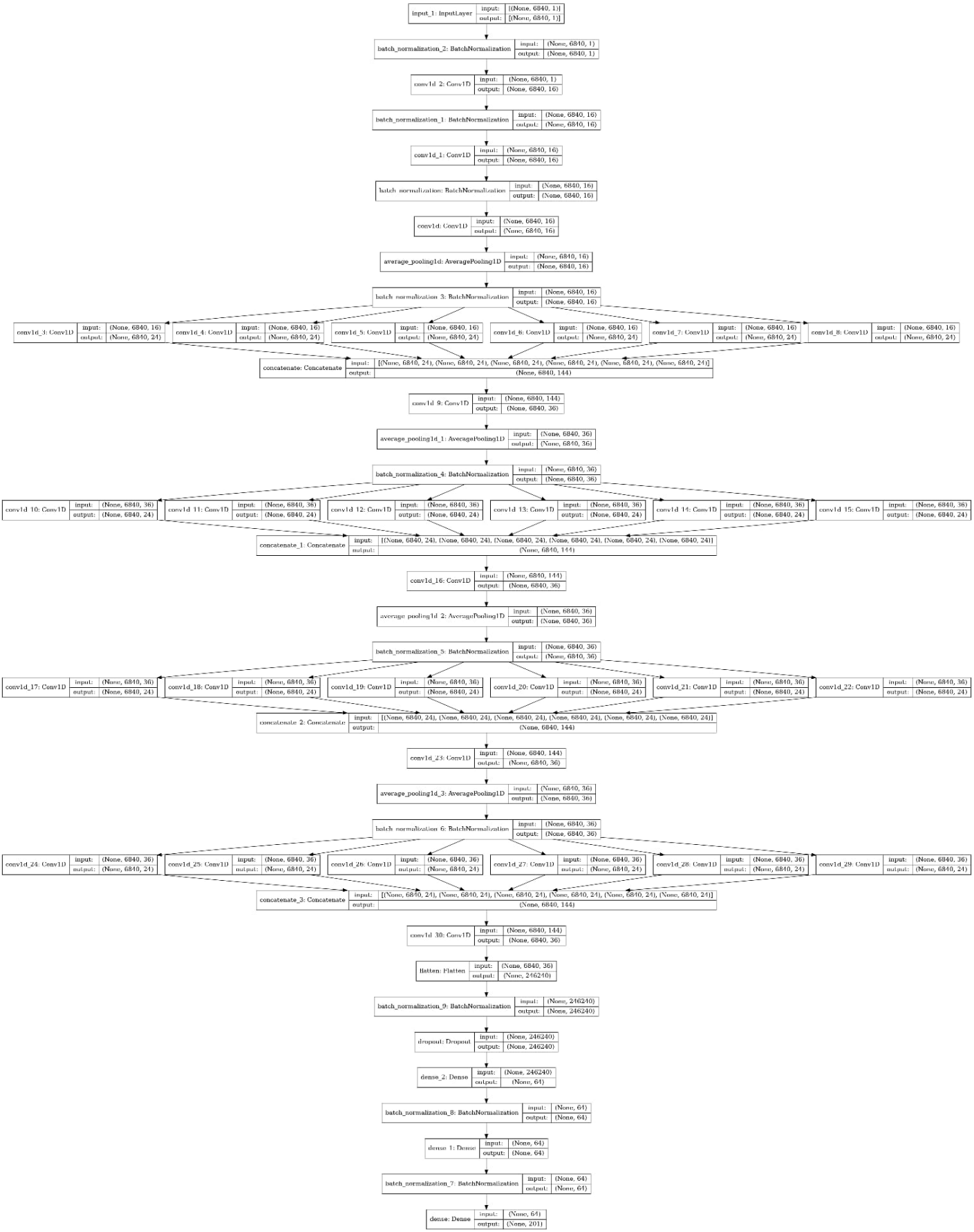


Figure 2: CNN architecture.

```
    val.idx = as.numeric(unlist(as.vector(log_list[str.id])))

    # Generate labels
    labels <- data$tempo
    cat_labels <- keras::to_categorical(labels)
    cat_labels = cat_labels[, (min(labels)+1):ncol(cat_labels)]
    seq_ = seq(min(labels), max(labels))
    colnames(cat_labels) <- seq_

    # Split data into validation and train, same split for all models to
    # evaluate on the same basis
    val.labels <- cat_labels[val.idx,]
    val.data <- data[val.idx,1:(40*171)]

    train.labels <- cat_labels[-val.idx,]
    train.data <- data[!val.idx,1:(40*171)]

    # Load model and predict on train and validation data
    new_model <- load_model_hdf5(file)
    res.train = new_model %>% keras_predict(as.matrix(train.data))
    res.val = new_model %>% keras_predict(as.matrix(val.data))

    # Estimate RMSE on training data
    tempo_predicted_train = apply(res.train,1,which.max)+62
    ground_truth_train = rep(NA, length(tempo_predicted_train))
    for (i in 1:length(ground_truth_train)) {
        ground_truth_train[i] = match(c(1), train.labels[i, ])+62
    }
    rmse.train <- sqrt(sum((tempo_predicted_train-ground_truth_train)^2) / length(tempo_predicted_tr

    # Estimate RMSE on validation data
    tempo_predicted_test = apply(res.val,1,which.max)+62
    ground_truth_test = rep(NA, length(tempo_predicted_test))
    for (i in 1:length(ground_truth_test)) {
        ground_truth_test[i] = match(c(1), val.labels[i, ])+62
    }
    rmse.val <- sqrt(sum((tempo_predicted_test-ground_truth_test)^2) /length(tempo_predicted_test))

    # Save RMSE values and file name into a csv file named results
    cat(paste0(file, ",", rmse.train, ",", rmse.val, "\n"),
        file="results.csv", append=TRUE)
}
```

```
generate_CNN_features <- function(data, model_path) {

  df = data[,1:(40*171)]

  print('Loading Model')
  new_model <- load_model_hdf5(model_path)
  print('Model Loaded')
  print('Doing Prediction')
  res = new_model %>% keras_predict(as.matrix(df))
```

Model Name	Train RMSE	Validation RMSE
weights312-128.01-0.63.hdf5	15.08	11.80
weights312-128.04-0.44.hdf5	9.41	7.97
weights312-128.07-0.23.hdf5	4.53	0.65

Table 1: Best CNN Models

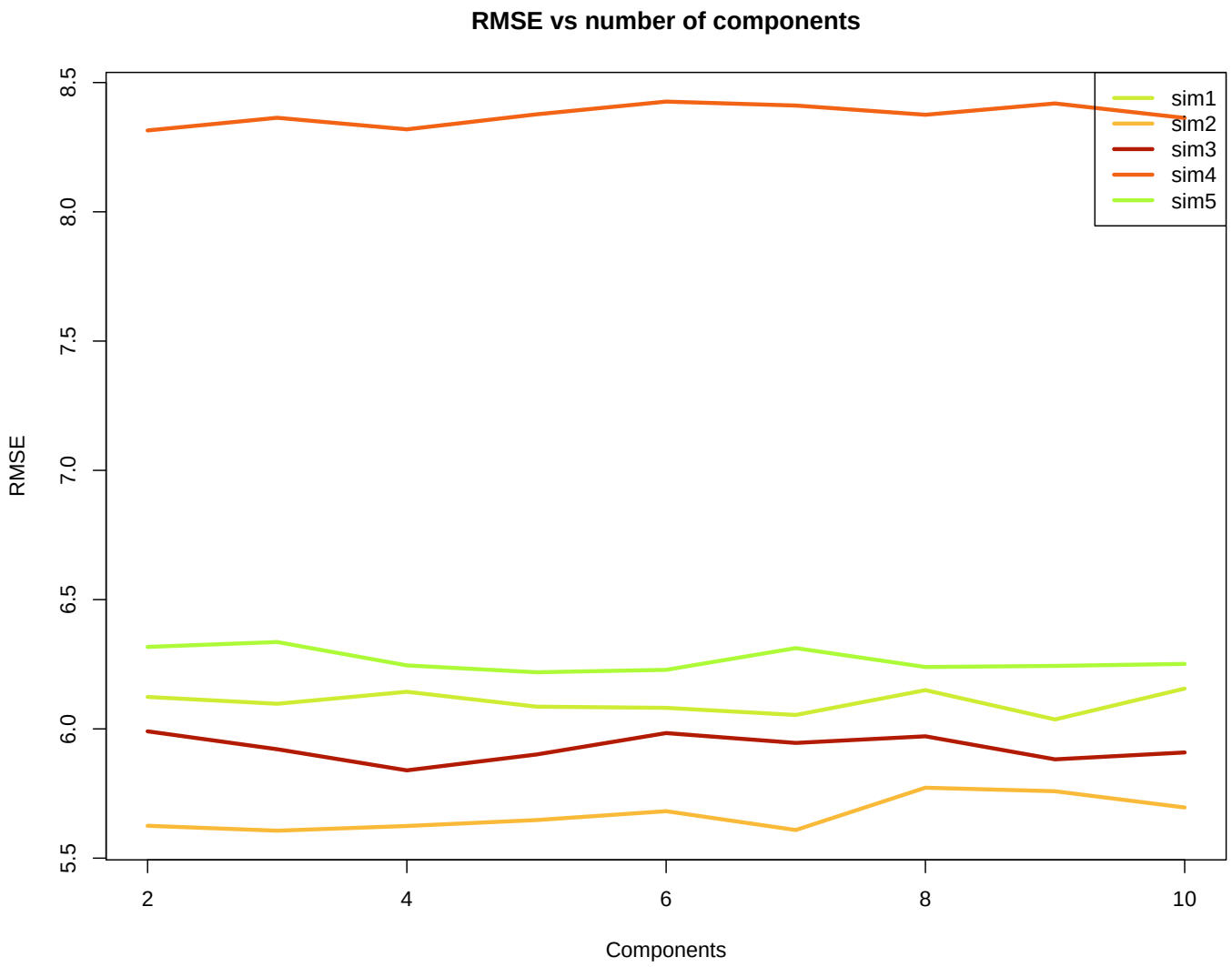
```
  print('Prediction Done, soon done')

  target = apply(res,1,which.max)+62
  prob.target = apply(res,1,max)

  output = list(target=target, prob.target=prob.target)
  return(output)
}

CNN.train = generate_CNN_features(data.train, '/content/weights312-128.04-0.44.hdf5')
CNN.test = generate_CNN_features(data.test, '/content/weights312-128.04-0.44.hdf5')
```

In order to choose which of the CNN models to use and what epoch we have stored the RMSE results that these CNN's return on the validation and test data. Our aim is to use a model for which the RMSE on the validation data is lower than that of the training data (implying the model is not overfitting) and such that the CNN can still be improved by the next classification model. The following table shows the three candidate models to generate the feature variables.

Feature Engineering

From the previous we have generated 2 features associated to the CNN model. However we still have multiple other features which need to be engineered before we can consider them in the final model. In particular we have 171 features that track the dominant frequency. We have considered that the best way to exploit these features was by reducing their dimensionality through SVD, seeking the optimal number of components (optimal is defined as that which returns the lowest RMSE on the validation data). The results of this analysis can be seen in the plot below.

```
cols <- sample(viridis::turbo(100), components)

matplot(2:components, sim_matrix, type="l", lty=1, lwd=3, col=cols,
        xlab="Components", ylab="RMSE", main="RMSE vs number of components")
legend("topright", legend=paste0("sim", 1:simulations), col=cols, lwd=3)
```



As we can observe, there is no predominant combination that generates the lowest RMSE, In this case, and in order to make the model more simple, we reduce the 171 features into 2 components.

Next we will tune the randomForest parameters in a similar fashion, by picking those parameters that give the lowest RMSE on validation:

```
sim_matrix_grid
```

```
##    ntree      mtry        s1       s2       s3        s4        s5      mean
## 1    100  5.571429 10.028392 9.677817 8.906068 10.667579 10.880979 10.032167
## 2    300  5.571429  9.817205 9.611951 8.807867 10.438085 10.557407  9.846503
## 3    500  5.571429  9.840768 9.624986 8.915104 10.290748 10.791968  9.892715
## 4    700  5.571429  9.698560 9.761601 9.133370 10.385162 10.594411  9.914621
## 5    900  5.571429  9.776410 9.628825 8.994359 10.506207 10.604045  9.901969
## 6    100  7.800000  8.224315 8.934801 8.005990  8.715350  9.399541  8.655999
## 7    300  7.800000  8.119618 8.901061 7.753182  9.186210  9.374013  8.666817
## 8    500  7.800000  7.989786 8.793139 7.668184  8.938107  9.421334  8.562110
## 9    700  7.800000  7.944830 8.783002 7.687069  8.904948  9.212805  8.506531
## 10   900  7.800000  8.170173 8.883879 7.744964  8.956751  9.360613  8.623276
## 11   100 13.000000  6.524690 8.055774 6.816748  7.613138  7.944812  7.391032
## 12   300 13.000000  6.434109 7.953348 6.587487  7.583623  7.940370  7.299787
```

```
## 13   500 13.000000  6.566572 8.050643 6.537070  7.489883  7.916178  7.312069
## 14   700 13.000000  6.430656 8.068352 6.504464  7.438239  7.853839  7.259110
## 15   900 13.000000  6.440725 8.074174 6.534180  7.415545  7.854018  7.263728
## 16   100 19.500000  6.208452 7.961732 6.304956  7.228191  7.580140  7.056694
## 17   300 19.500000  5.951064 7.904465 6.295080  7.147937  7.504090  6.960527
## 18   500 19.500000  5.860041 7.836875 6.283237  7.187208  7.523821  6.938236
## 19   700 19.500000  5.889718 7.811242 6.267295  7.182439  7.498287  6.929796
## 20   900 19.500000  5.958446 7.759833 6.245188  7.163443  7.524100  6.930202
## 21   100 37.000000  6.040342 7.922214 6.292383  7.611185  7.672044  7.107634
## 22   300 37.000000  6.154831 7.967800 6.399337  7.532372  7.649306  7.140729
## 23   500 37.000000  6.051242 7.933734 6.450882  7.513973  7.613633  7.112693
## 24   700 37.000000  6.108532 7.894283 6.383613  7.563235  7.606183  7.111169
## 25   900 37.000000  6.110018 7.947132 6.385381  7.524011  7.619947  7.117298
```

From this grid search we can obtain the best parametrization of the Random Forest, which will be the model used to do predictions on the test data.

B) Predicting with confidence

1.

In the following section, we implement the split conformal prediction for regression algorithm to compute confidence intervals for our estimations.

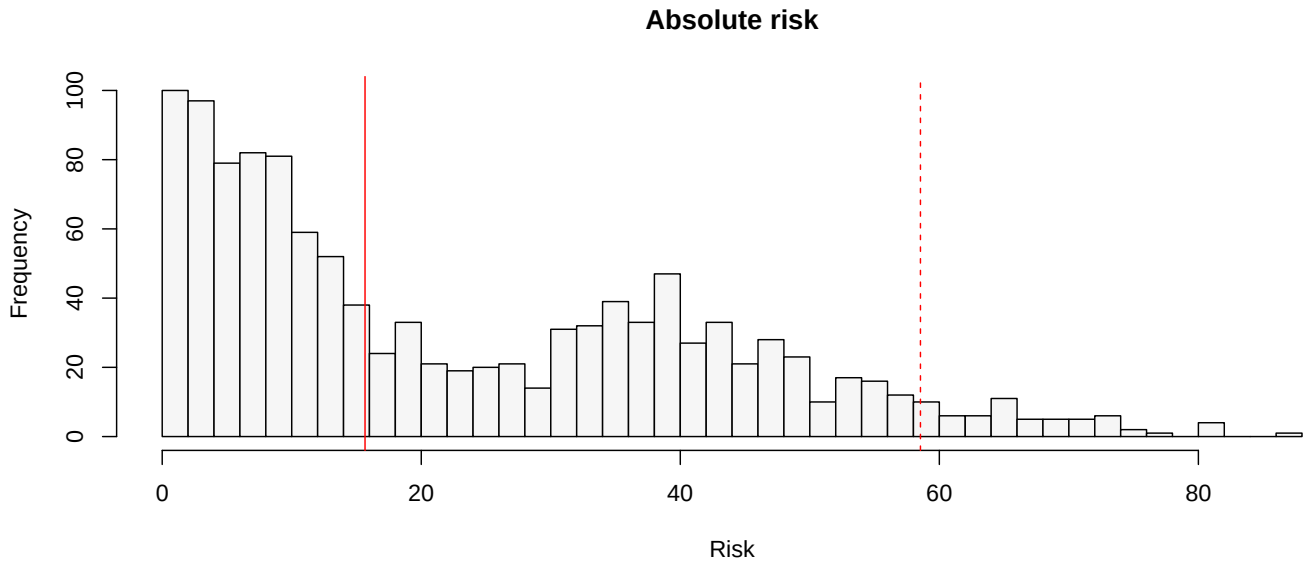
The implementation of this algorithm can be subdivided into five steps, the first of which is to randomly split the provided training set into two equal subsets. We call these subsets D1 and D2 and they act as the training and validating datasets.

```
x = rf_model_final$dat
y = as.vector(df.train$tempo)
n = nrow(x)/2
n_obs = n*2 # size of D1
ind <- sample(1:n_obs, n, replace=F)
# split 50 %

D1 = x[sort(ind),]
D2 = x[-sort(ind),]

D1_y = y[sort(ind)]
D2_y = y[-sort(ind)]
```

Secondly, we use D1 dataset to train the random forest model and then in the third step, we apply the model to the D2 dataset to obtain the corresponding predictions. Since the variable of interest is known in this dataset, we can evaluate its residuals as the absolute difference between the predicted value and the true one.

```
model = randomForest(x = D1, y = D1_y, xtest = D2, mtry=best_mtry,
                     ntree=best_ntree)

y_D2_pred = model$test$predicted
R = abs(D2_y - y_D2_pred)
```

```
## Warning in D2_y - y_D2_pred: longer object length is not a multiple of shorter
## object length
```



From the histogram above we can see how the lowest 50% of residuals are all under 20, while only the top 5% are above 62.

In the fourth step, we compute d, which is the value that determines the width of the confidence interval. In order to compute it, it is necessary to firstly sort the residuals in ascending order and then calculate the index of the residual that should be considered. Such index is k, obtained with the following formula: $[(n/2+1)(1-\alpha)]$, where α is the probability that our parameter of interest will actually be included in the interval. In this case, having chosen $\alpha = 0.05$ we expect that 95% of the D2_y values are included in the interval.

```
alpha = 0.05
k = (n + 1)*(1-alpha)
d = sort(R)[round(k)]
CI_validation_df = data.frame(matrix(c(D2_y,y_D2_pred - d,y_D2_pred + d),ncol=3))
```

```
## Warning in matrix(c(D2_y, y_D2_pred - d, y_D2_pred + d), ncol = 3): data length
## [1951] is not a sub-multiple or multiple of the number of rows [651]
```

```
colnames(CI_validation_df)=c('Y','low_pred','up_pred')
head(CI_validation_df)
```

```
##     Y low_pred  up_pred
## 1 119      140 123.8641
## 2 140      144 131.7257
## 3 126      124 123.1977
## 4  84      140 124.1023
## 5 140      126 119.3435
## 6 126      140 140.9614
```

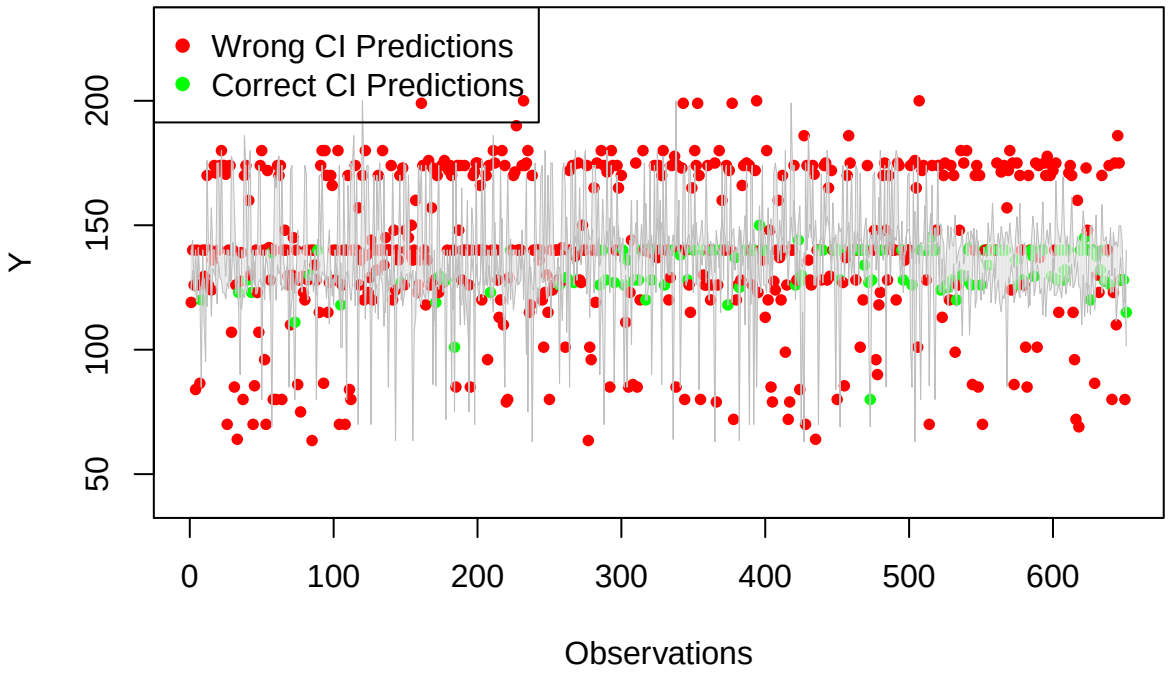
The table above provides some examples of the actual values of the variable of interest and the corresponding confidence interval. It can be observed how the values do actually fall within the lower and upper bounds.

Lastly, we return the confidence intervals for each value of our validation set and plot them.

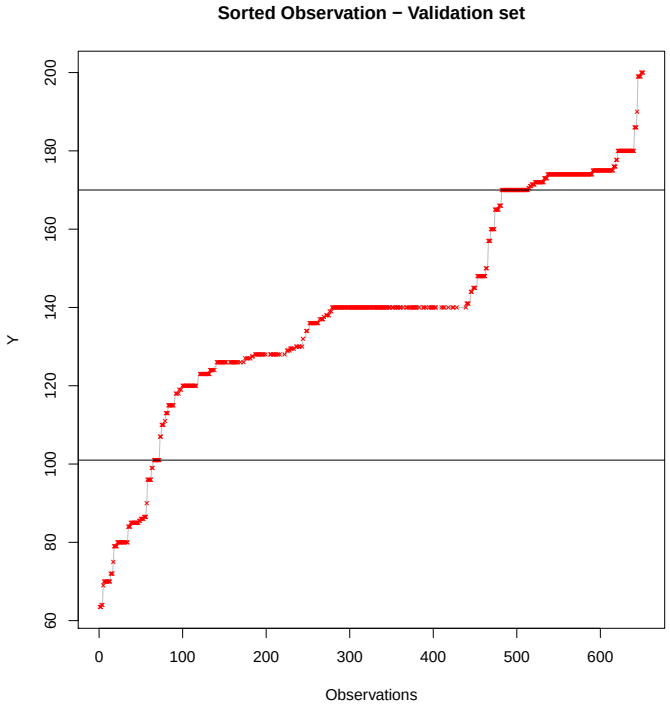
```
## [1] "Percentage of predictions that are not included in the CI: 1.32820512820513"
```

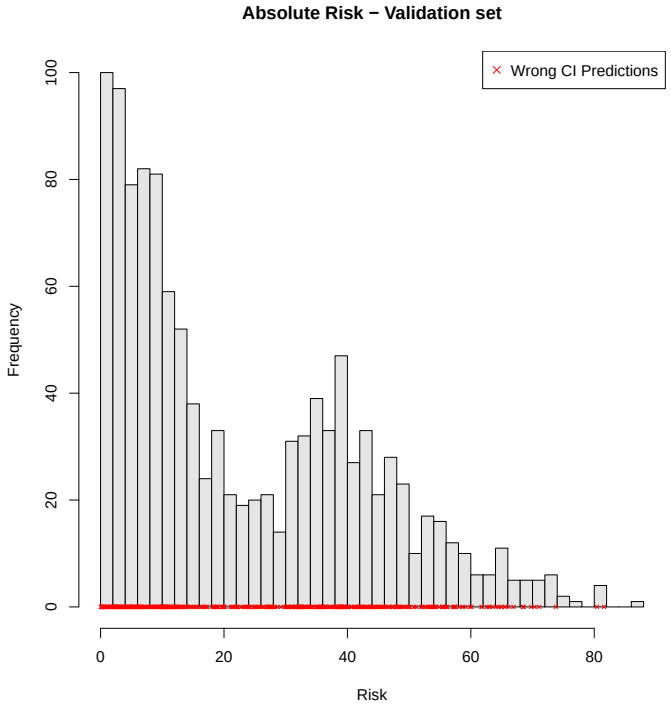
The plot below shows the prediction intervals for each data point in the D2 set, along with the true values of the target variable. The true values are green when they fall within the interval and red when they do not. Only 4.92% of the predictions here are inaccurate.

```r
ax <- 1:length(predictions_y)
plot(predictions_y, type="p", col=names(predictions_y),cex=0.8,ylim=c(40,230),pch=16,ylab='Y',xlab='
segments(x0=ax,x1=ax,y0=CI_validation_df$low_pred,y1=CI_validation_df$up_pred,col='#e5e5e5',lwd=0.6,
points(ax, CI_validation_df$up_pred, type="l", col="gray",lwd=0.1,ylim=c(40,230))
points(ax, CI_validation_df$low_pred, type="l", col="gray", lwd=0.1,ylim=c(40,230),
       main="Prediction interval", ylab="interval", xlab="obs")
legend("topleft",legend=c('Wrong CI Predictions','Correct CI Predictions'),col=c("red","green"),pch=
```



Checking on wrong predictions By investigating further on the nature of the inaccurate predictions, we find that they all fall on the more extreme sides of the distribution (first decile or fourth quartile) and they all have a high absolute risk.





2.

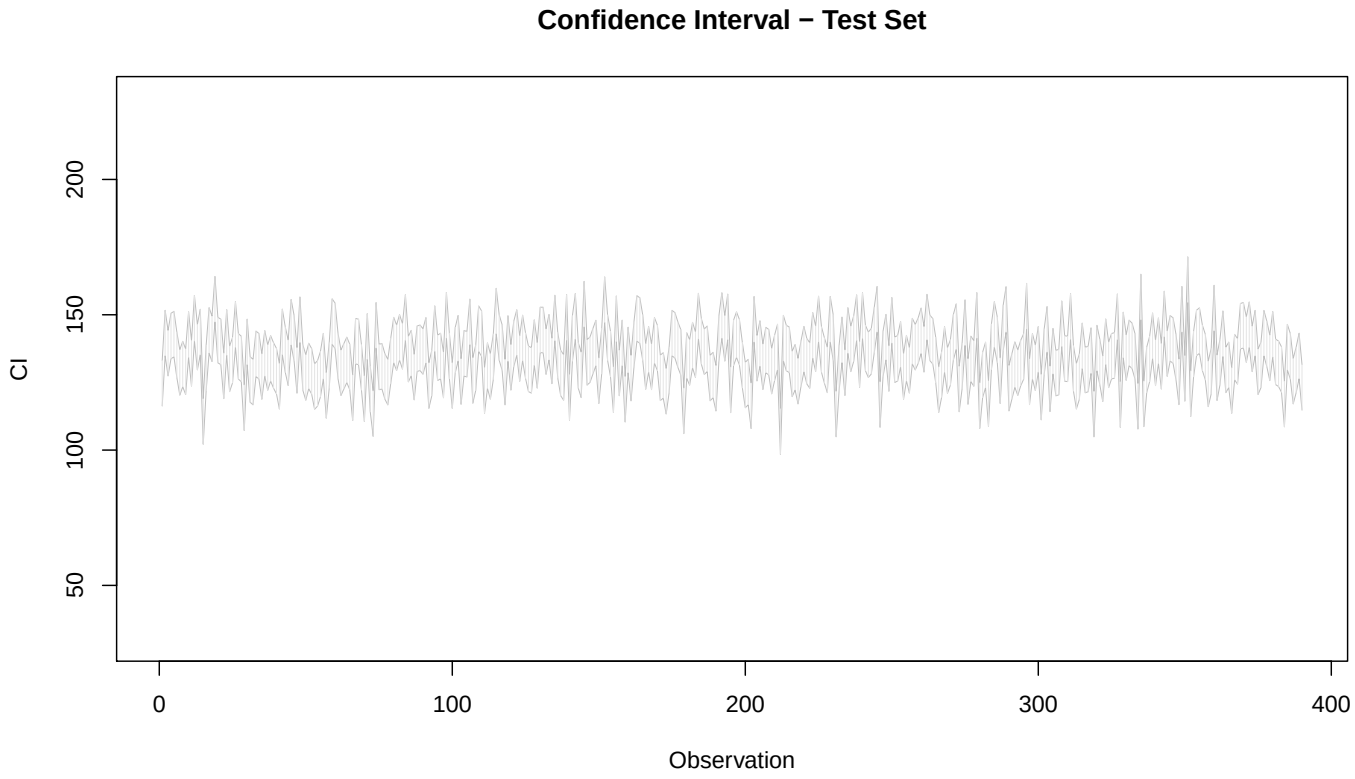
Now that we have a model and the value of d needed for the split conformal prediction for regression algorithm, we can apply it to the actual testing set to make our predictions. Of course, for this set the real values are not provided, but we can train the model on the D1 set as we did before and compute the confidence intervals using the d value found in the step before.

```
predictions = randomForest(x = D1, y = D1_y, xtest = df.test.svd,
                           mtry=best_mtry,
                           ntree=best_ntree)$predicted
colnames(df.test.svd)
```

```
##  [1] "feature_1"          "feature_2"          "time.P1"
##  [4] "time.M"             "time.P2"            "time.IPR"
##  [7] "freq.P1"            "freq.M"             "freq.P2"
## [10] "freq.IPR"           "mean"               "sd"
## [13] "median"             "sem"                "mode"
## [16] "Q25"                "Q75"                "IQR"
## [19] "cent"               "skewness"           "kurtosis"
## [22] "sfm"                "sh"                 "prec.x"
## [25] "roughness"          "rugo"               "sfm.1"
## [28] "shannon"            "simpson"            "renyi"
## [31] "genre"              "CNN_Prediction_3"   "CNN_Prediction_Prob_3"
```

3.

Finally, we can visualise these results and provide final comments.



Given the results in the training stage we expect these predictions to have an accuracy of at least 95%.

C) Variable Importance

In this last section the LOCO (leave-one-covariate-out) inference will be applied on our model to infer quantitavely the impact that the top 5 variables of our model have on our final model. In our particular case, since we are using a Random Forest as our final model, it is easy to obtain the top 5 most important variables. These variables are:

1. CNN_Prediction_3
2. genre
3. CNN_Prediction_Prob_3
4. freq.M
5. mode

This variables are the variables we which hoped the model would consider the most relevant as well, in particular the top 3. CNN_Prediction_3 gives a good approximation of the target variable while genre and CNN_Prediction_Prob_3 try to correct in cases in which CNN_Prediction_3 returns an incorrect result.

The following steps have been followed for the LOCO inference, each of which can be illustrated in the code following this description:

1. **Split Data:** We will split the data in two. D_{n_1} will consist of 70% of the data, while D_{n_2} will be composed by the other observations not included in D_{n_1}.
2. **Apply model on D_{n_1}:** We will run our random forest on the first data set. This model will generate our estimate $\hat{f}_{n_1}(\cdot)$. We can consider this as the base estimate and we will see what impact removing variables from the model will have on this estimate.

3. **Remove one feature:** Next step is to recompute the previous estimate but in this case we will remove one of the top 5 features, leading to the estimate defined as $\hat{f}_{n_1}^{-j}(\cdot)$. We will generate 5 of these estimates, removing one feature every time.

4. **Compute the error prediction:** Given the previously mentioned ingredients the next step is to calculate the prediction error by the equation:

$$|Y - \hat{f}_{n_1}^{-j}(X)| - |Y - \hat{f}_{n_1}(X)| \tag{1}$$

which we will summarize using the median. This quantity is random since it depends on the dataset D_{n_1}, therefore in order to get a better representation we will use a non-parametric bootstrap to obtain its confidence interval. Bonferroni will be used to adjust for multiplicity ($\alpha/5$).

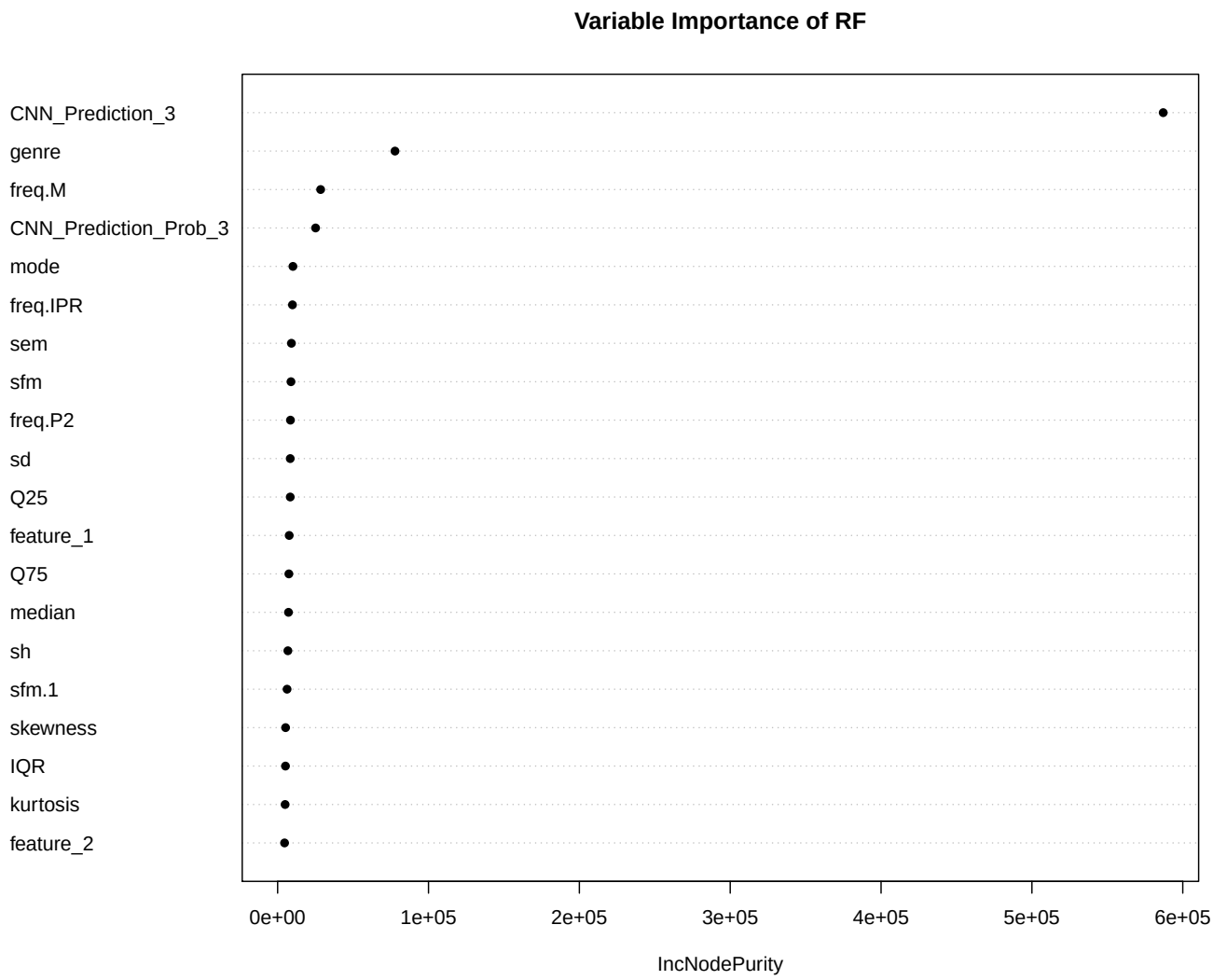
```
#== 1. Split the data into 2 parts --> Dn1 and Dn2

df.train <- apply_svd(df.train, n_components=2, plot=F)

Dn <- select_features(df.train, features=c('domfr', 'mel',
                                           'CNN_Prediction_1',
                                           'CNN_Prediction_2',
                                           'CNN_Prediction_Prob_1',
                                           'CNN_Prediction_Prob_2'),
                      set.difference = T)



# Split the data set into Dn1 and Dn2
Dn1 <- split_xy(Dn, proportion=0.7)
Dn2 <- split_xy(Dn, indexes=-Dn1$idx)


#== 2. Run any algorithm on Dn1 using all the feature variables used in the champion model --> fn1

# Train the model on Dn1
fn1 <- randomForest(x = Dn1$x, y = Dn1$y)


#== 3. From here we will take the 5 most relevant variables (these will be the j
# variables we will use in the algorithm)

n_top_var <- 5

variables <- colnames(Dn1$x)
top_var <- variables[sort(importance(fn1), index.return=T, decreasing = T)$ix][1:n_top_var]



#== 4. Pick a variable (one of the 5 most important variables) and rerun the algorithm on Dn1 but w

# Define the list for holding the models fn1-j
model.list <- lapply(rep(NA, n_top_var), function(x) {NA})
names(model.list) <- top_var
```

```r
# Run the model leaving the variable "var" out
for (var in top_var) {
  model.list[[var]] <- randomForest(x = select_features(Dn1$x, features=c(var),
                                                        set.difference=T),
                                    y = Dn1$y)
}



#== 5. Using a bootstrap approach (to  get the confidence interval around the estimate which is act

# Predict on Dn2
y.Dn2.pred <- predict(fn1, Dn2$x)


# Setup the non-parameteric bootstrap procedure
B = 1000 # number of simulations

median.boot <- as.data.frame(matrix(NA, nrow=B, ncol=length(top_var)))
colnames(median.boot) <- top_var

median.true <- rep(NA, length(top_var))
names(median.true) <- top_var

error.dis <- matrix(NA, nrow=length(Dn2$y), ncol=length(top_var))
colnames(error.dis) <- top_var

n2 <- length(Dn2$y)

for(var in top_var) {

  # Take the data without the variable "var" and predict
  x.Dn2.minus.j <- select_features(Dn2$x, features=c(var), set.difference=T)
  y.Dn2.minus.j.pred <- predict(model.list[[var]], x.Dn2.minus.j)

  # Compute the delta error
  delta.error <- abs(Dn2$y - y.Dn2.minus.j.pred) - abs(Dn2$y - y.Dn2.pred)

  # Compute the median and save the delta error
  median.true[[var]] <- median(delta.error)
  error.dis[,var] <- abs(delta.error)

  for (b in 1:B) {

    # Sample the indexes
    idx.Dn2.boot <- sample(1:n2, size=n2, replace=T)

    # Get the bootstrap sample
    x.Dn2.minus.j.boot <- x.Dn2.minus.j[idx.Dn2.boot,]
    y.Dn2.boot <- Dn2$y[idx.Dn2.boot]
```

```
    # Predict on the bootstrap sample with and without the variable "var"
    y.Dn2.minus.j <- predict(model.list[[var]], x.Dn2.minus.j.boot)
    y.Dn2.pred.boot <- y.Dn2.pred[idx.Dn2.boot]

    # Compute the median
    median.boot[b, var] <- median(abs(y.Dn2.boot - y.Dn2.minus.j) -
                                  abs(y.Dn2.boot - y.Dn2.pred.boot))

  }
}
```

```
#== 6. Compute the confidence intervals

conf.int <- function(median.boot, median.true, alpha, n_correction) {

  alpha <- alpha / n_correction
  se.boot <- apply(median.boot, 2, sd)
  CI.boot.vec.lower <- median.true - qnorm(1-alpha/2)*se.boot
  CI.boot.vec.upper <- median.true + qnorm(1-alpha/2)*se.boot
  CI.boot.list <- list(lower=CI.boot.vec.lower, upper=CI.boot.vec.upper)
  return(CI.boot.list)
}
```

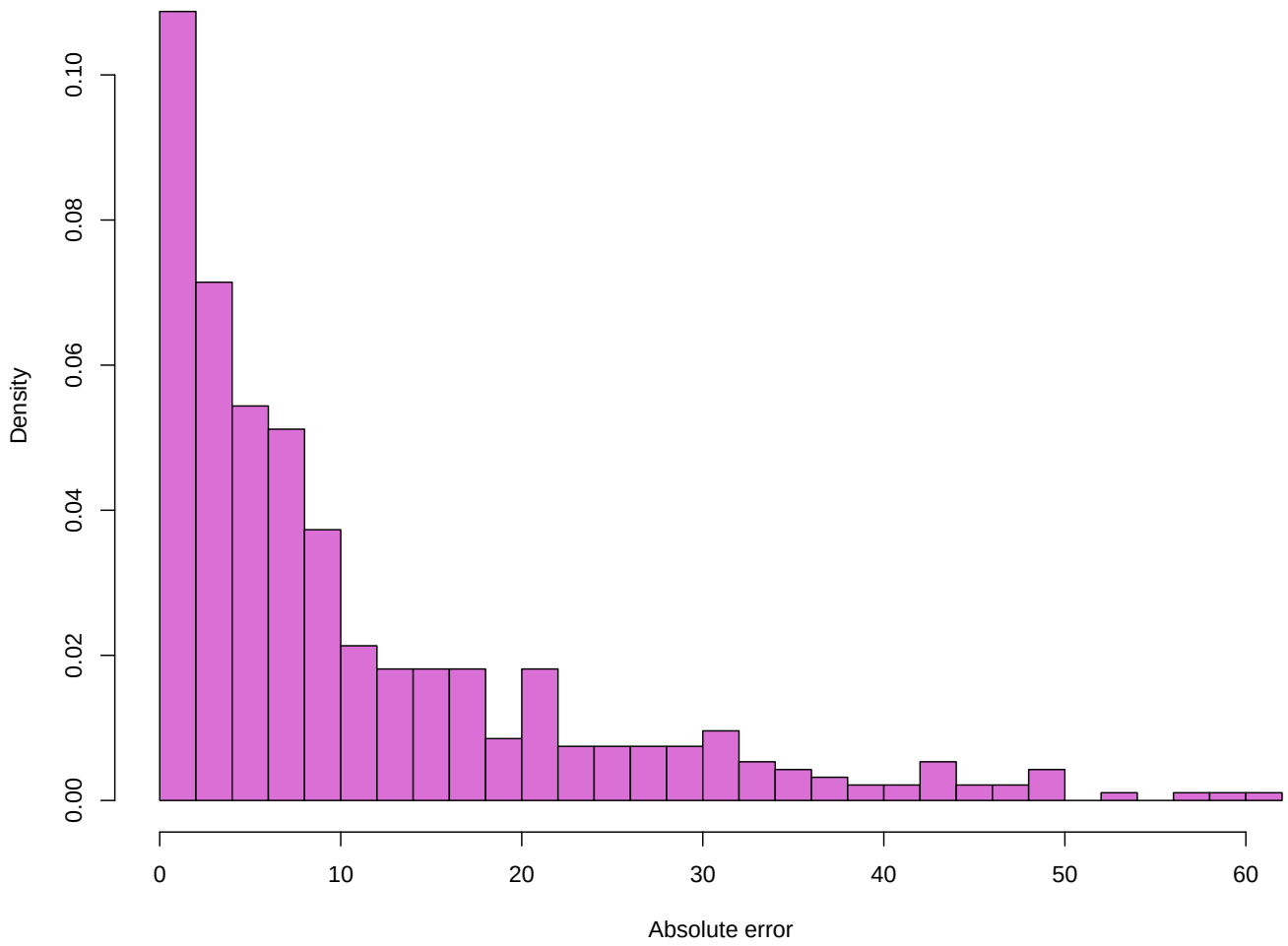
The next plots will allow us to visualize the most relevant findings of LOCO:

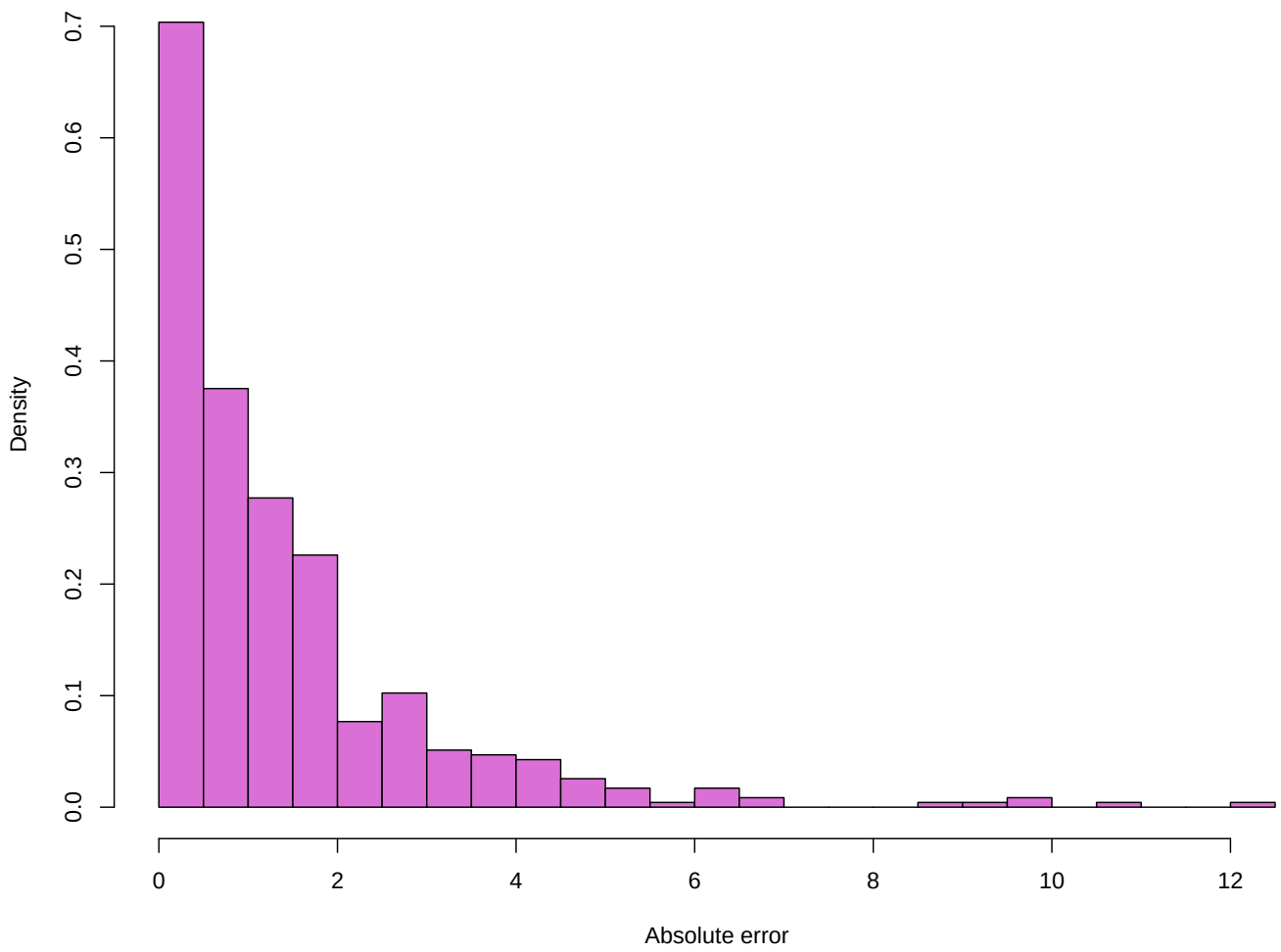


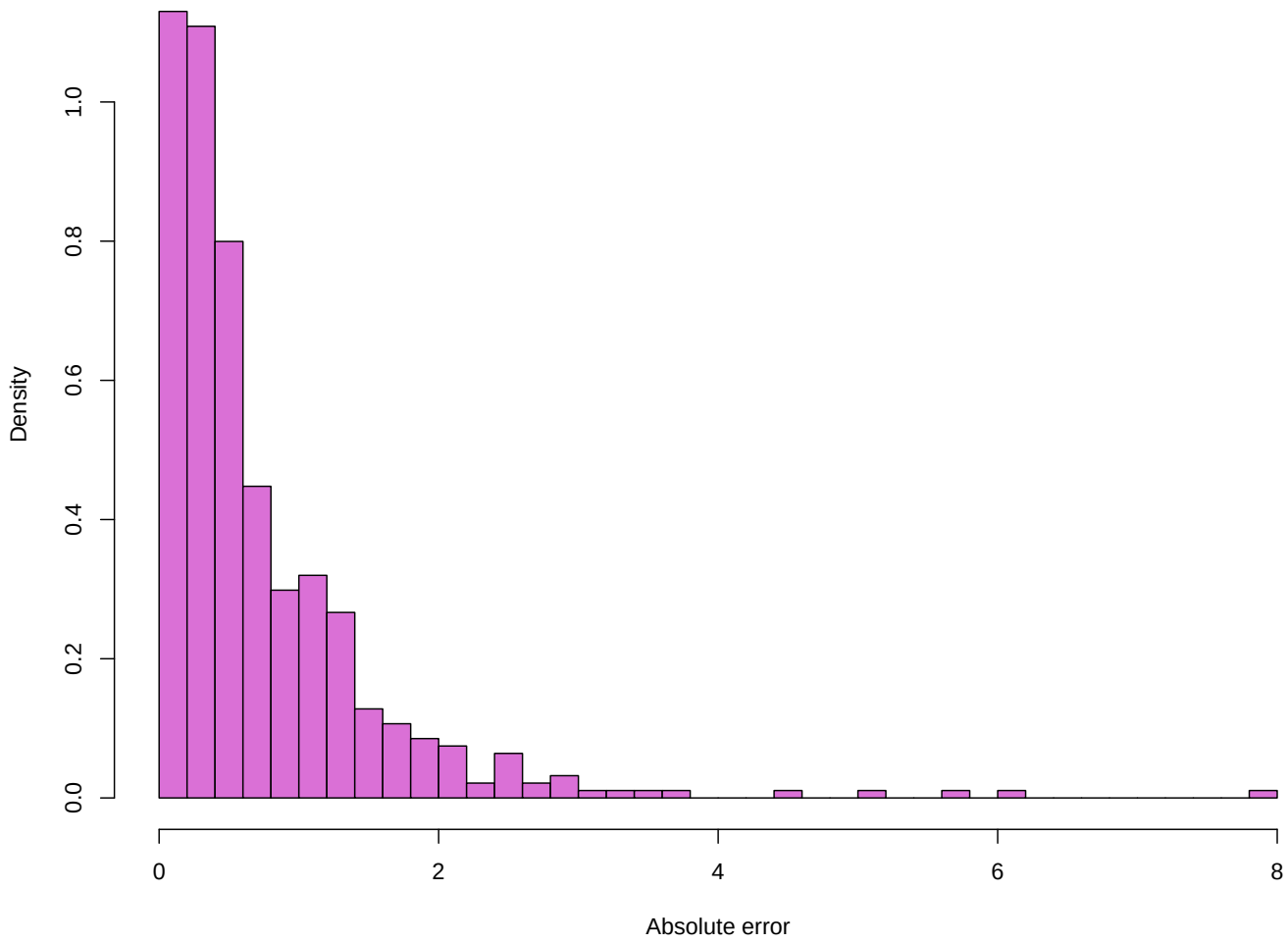
As we had already anticipated at the beginning of the section, the most relevant feature by far is CNN_Prediction_3, which is the prediction generated by our CNN model.The following plots will quantify this importance more accurately and in a more intuitive manner.

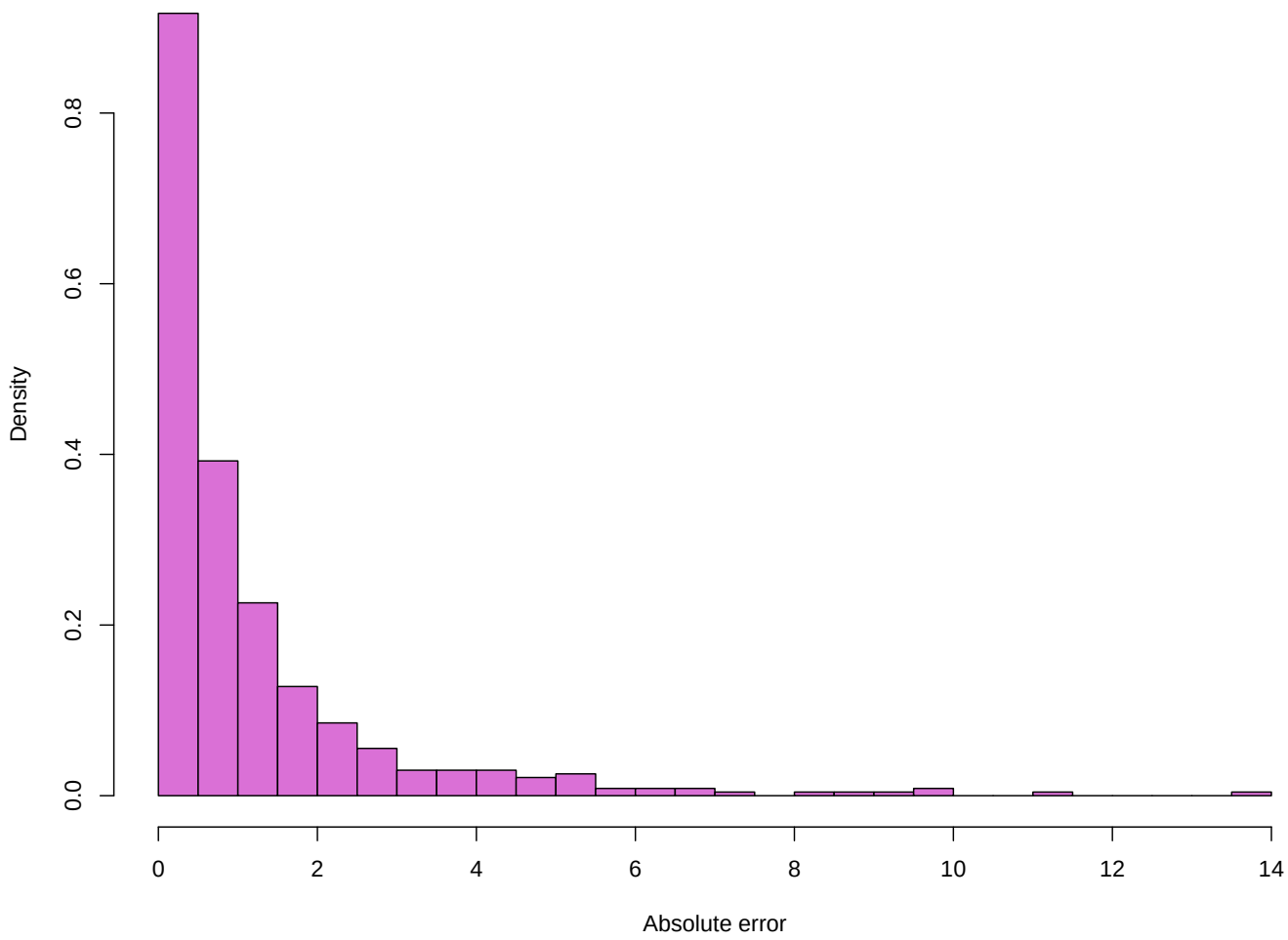
The following plot gives us the distribution (generated from the bootstrap) of the absolute error when removing one of the feature variables. It can be seen that the variable for which this error can aquire largest values is the most important variable, i.e. CNN_Prediction_3 (we could reach a maximum of 80 in absolute error).

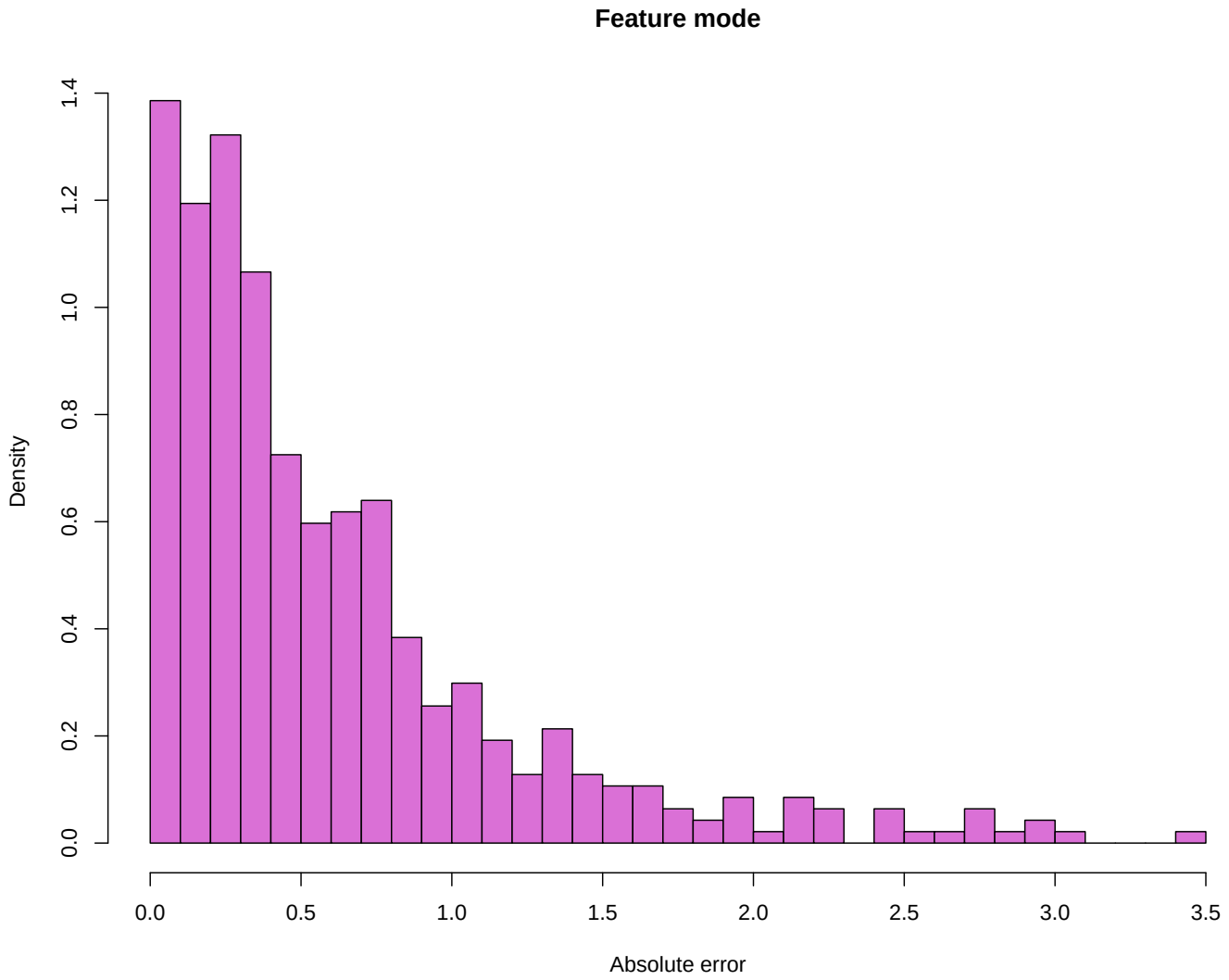
```
## NULL
```


Feature CNN_Prediction_3
Density
0.00
0.02
0.04
0.06
0.08
0.10
0
10
20
30
40
50
60
Absolute error

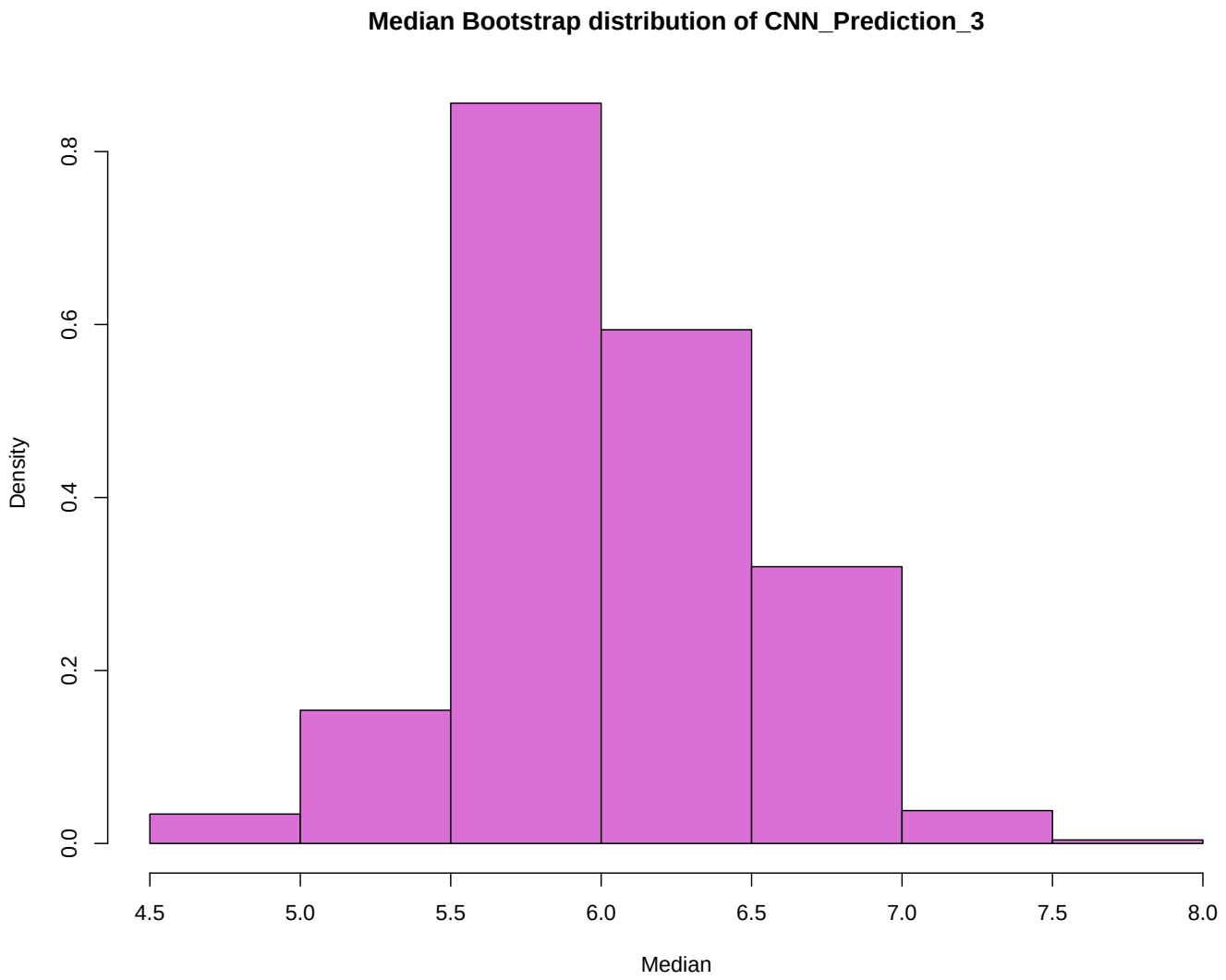

Feature genre
Density
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
2
4
6
8
10
12
Absolute error

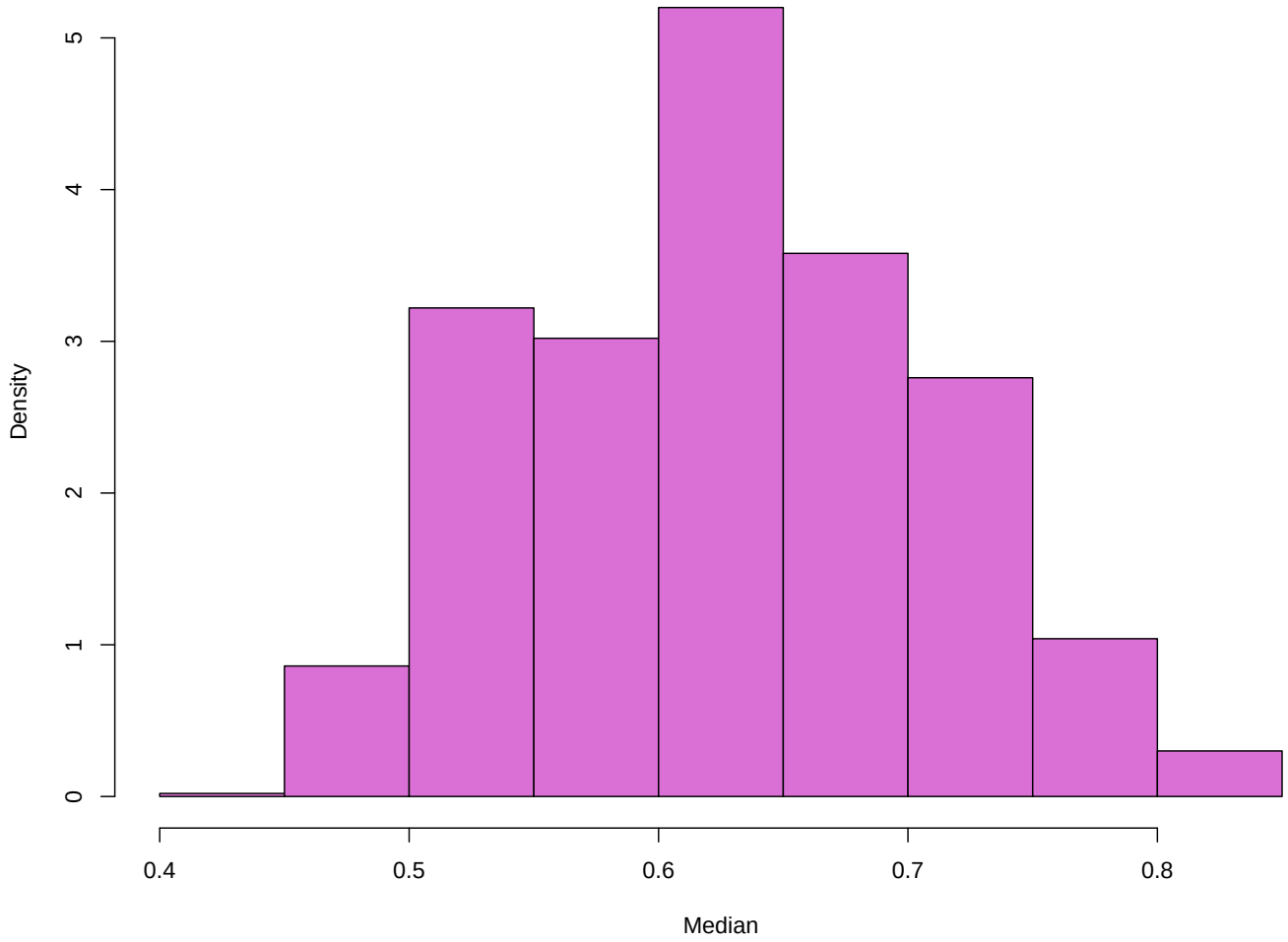

Feature freq.M
Density
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
Absolute error

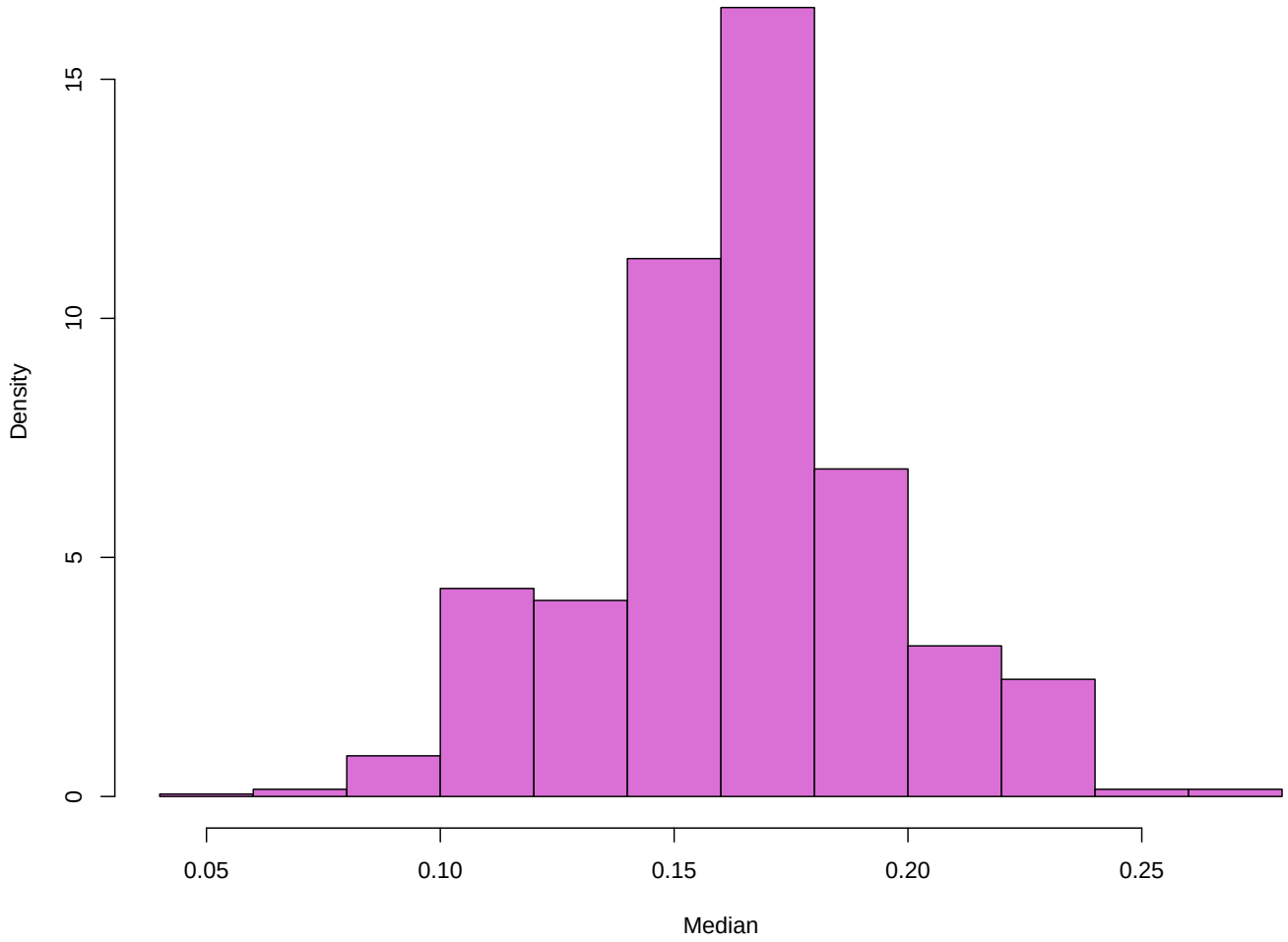

Feature CNN_Prediction_Prob_3
Density
0.0
0.2
0.4
0.6
0.8
0
2
4
6
8
10
12
14
Absolute error

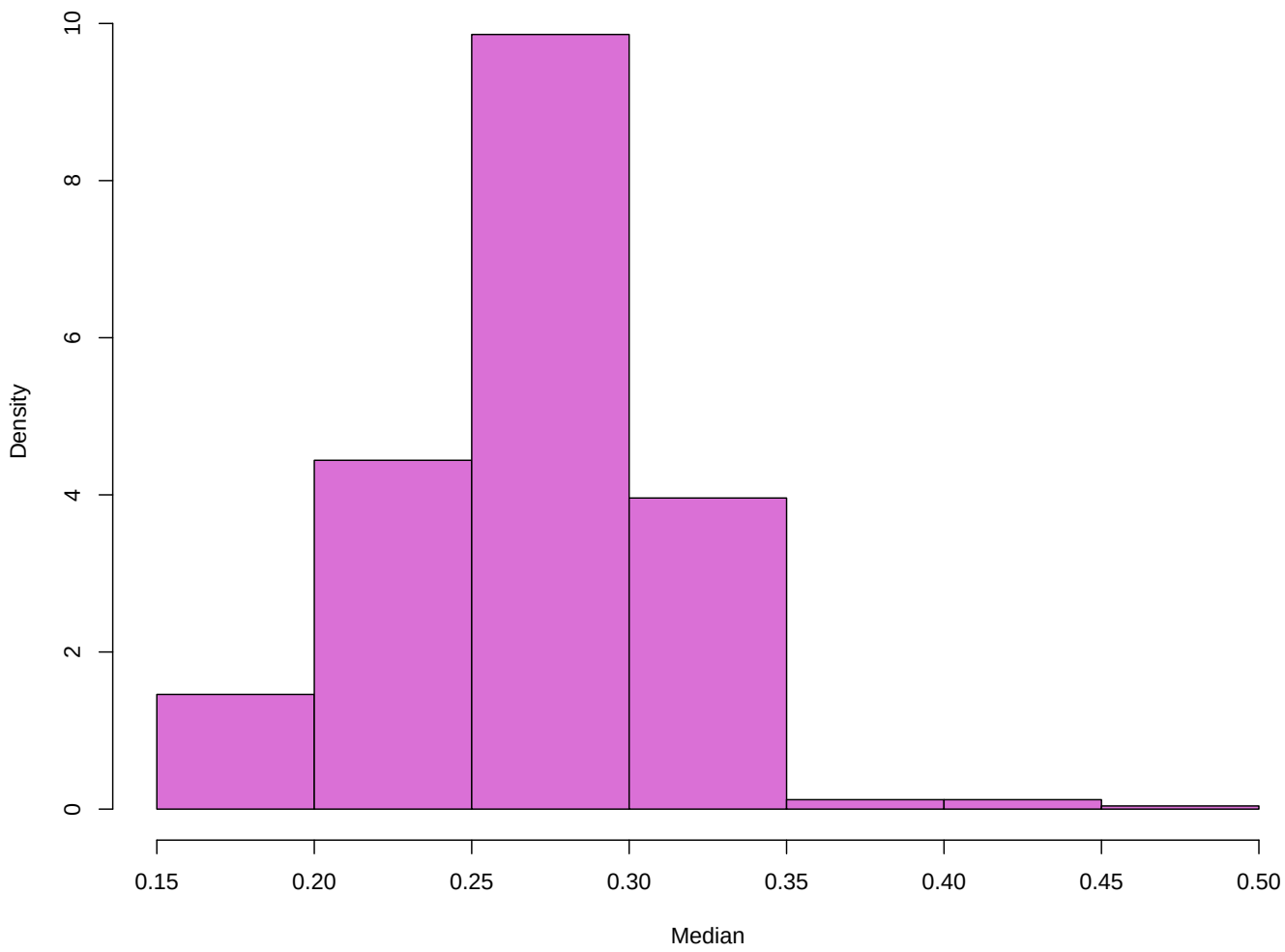


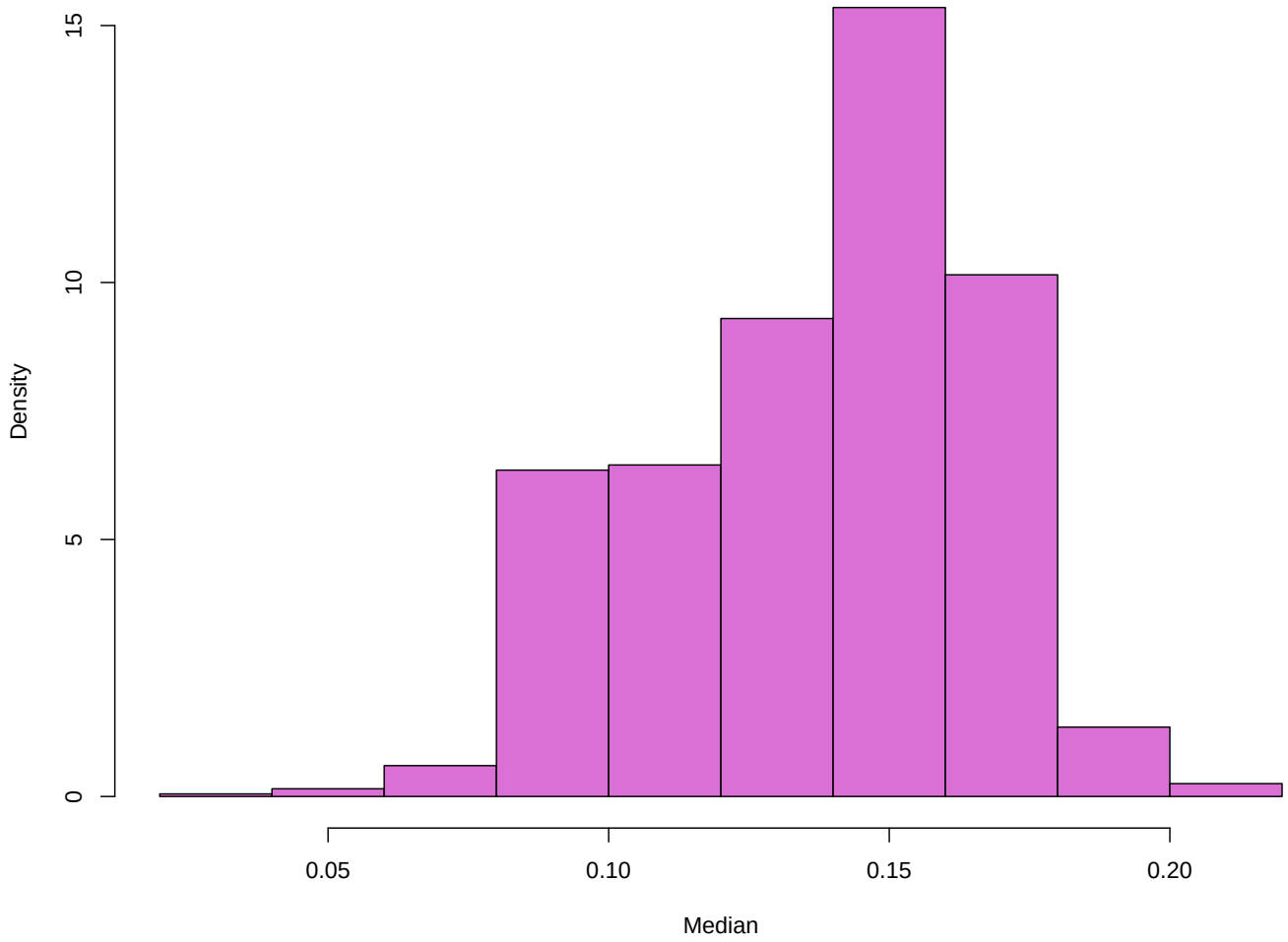
We will also plot the distribution of the median of the prediction error. We can observe that the average value of this median is very close to the true median value (LLN). It can also be appreciated that the distribution of this median is Gaussian, which justifies the choice made to compute the confidence intervals for this quantity.


Median Bootstrap distribution of CNN_Prediction_3
Density
0.0
0.2
0.4
0.6
0.8
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
Median

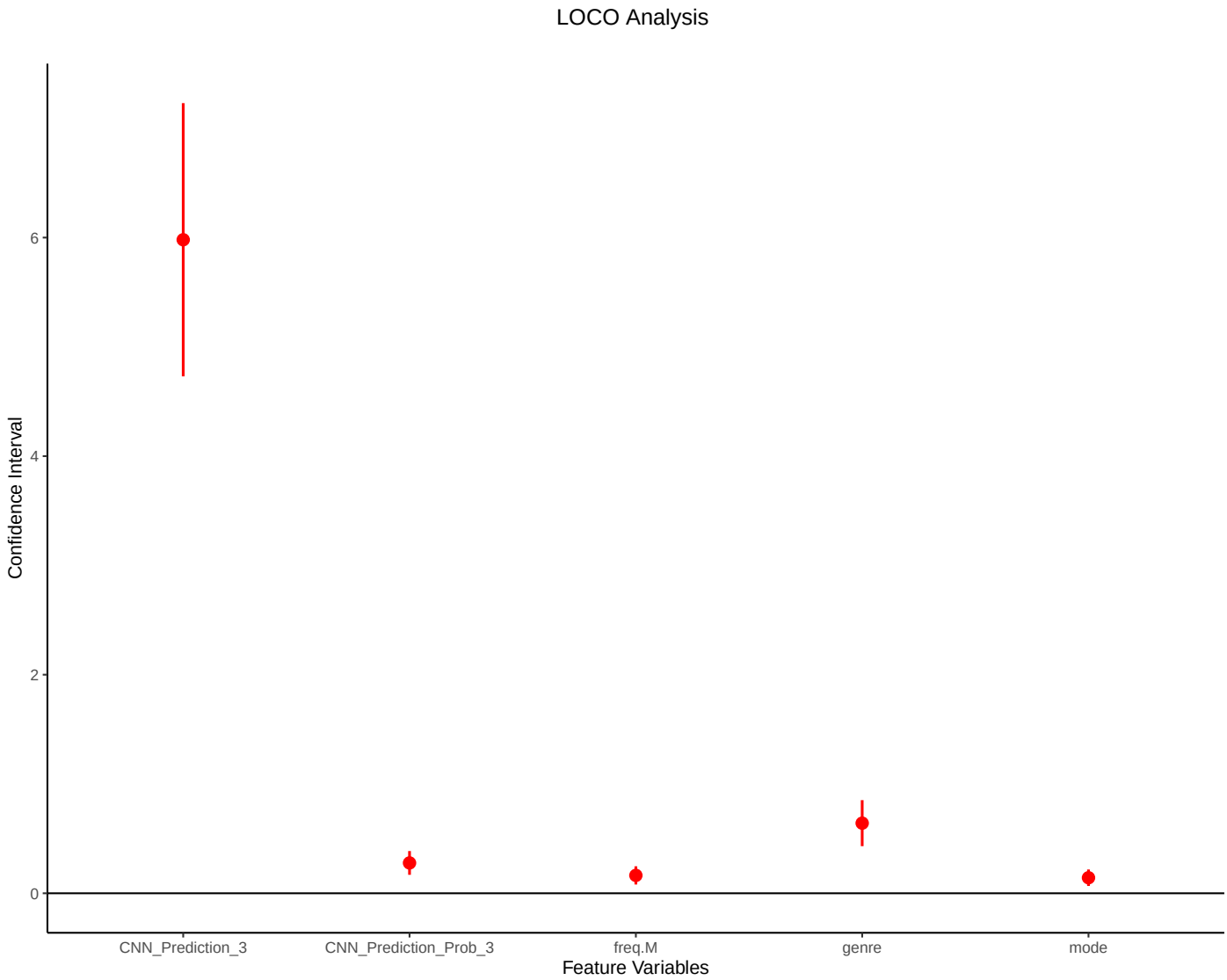

Median Bootstrap distribution of genre
Density
0
1
2
3
4
5
0.4
0.5
0.6
0.7
0.8
Median


Median Bootstrap distribution of freq.M
Density
0
5
10
15
0.05
0.10
0.15
0.20
0.25
Median


Median Bootstrap distribution of CNN_Prediction_Prob_3
Density
0
2
4
6
8
10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
0.50
Median



Finally, another representation of how much more impact the feature CNN_Prediction_3 has over the other variables is the following plot, in which we have also included the confidence interval around the true median value.



In conclusion, we can say that the variable that has most impact on the prediction error is CNN_Prediction_3, which is in line with our initial expectation. The random forest is able to identify that this variable already acts as a very good predictor and simply tries to correct it (mainly using the other top 4 feature variables).