Penalized Logistic Regression for handwritten digit recognition

Alexandre H.

Part 2: Penalized Logistic Regression Algorithm

This section introduces the core functions required for implementing penalized logistic regression, including functions to load data, evaluate model performance, make predictions, compute the gradient and loss for the model, and handle regularization.

```
library(glmnet)
library(ggplot2)
library(pROC)
library(readr)


Load_data <- function(data_dir) {
  # Function to read the data
  raw_data <- data.matrix(read_csv(data_dir, col_names = FALSE))
  x_data <- raw_data[,-1]
  y_data <- raw_data[,1]
  return(list(x = x_data, y = y_data))
}

Evaluate <- function(true_label, pred_label) {
  #  Compute the 0-1 loss between two vectors
  #
  #  @param true_label: A vector of true labels with length n
  #  @param pred_label: A vector of predicted labels with length n
  #  @return: fraction of points get misclassified

  error <- mean(true_label != pred_label)

  return(error)
}



Predict_logis <- function(data_feature, beta, beta0, type) {
  # Predict by the logistic classifier.
  #
  # Note: n is the number of examples
  #       p is the number of features per example
  #
  # @param data_feature: A matrix with dimension n x p, where each row corresponds to
  #   one data point.
  # @param beta: A vector of coefficients with length equal to p.
```

```r
  # @param beta0: the intercept.
  # @param type: a string value within {"logit", "prob", "class"}.
  # @return: A vector with length equal to n, consisting of
  #   predicted logits,         if type = "logit";
  #   predicted probabilities,  if type = "prob";
  #   predicted labels,         if type = "class".

  p <- exp(beta0 + data_feature %*% beta) / (1 + exp(beta0 + data_feature %*% beta))

  if (type=="logit") {
    return(log(p/(1-p)))
  } else if (type=="prob") {
    return(p)
  } else if (type=="class") {
    return(ifelse(p > 0.5, 1, 0))
  } else {
    stop("type not supported")
  }

}



Comp_gradient <- function(data_feature, data_label, beta, beta0, lbd) {
  # Compute and return the gradient of the penalized logistic regression
  #
  # Note: n is the number of examples
  #       p is the number of features per example
  #
  # @param data_feature: A matrix with dimension n x p, where each row corresponds to
  #   one data point.
  # @param data_label: A vector of labels with length equal to n.
  # @param beta: A vector of coefficients with length equal to p.
  # @param beta0: the intercept.
  # @param lbd: the regularization parameter
  #
  # @return: a (p+1) x 1 vector of gradients, the first coordinate is the gradient
  #   w.r.t. the intercept.

  n <- nrow(data_feature)
  p <- ncol(data_feature)

  proba <- Predict_logis(data_feature, beta, beta0, "prob")
  grad_intercept <- (1/n) * sum(proba - data_label)
  grad_beta <- (1/n) * t(data_feature) %*% (proba - data_label) + lbd * beta
  grad <- c(grad_intercept, grad_beta)

  return(grad)
}



Comp_loss <- function(data_feature, data_label, beta, beta0, lbd) {
```

```r
  # Compute and return the loss of the penalized logistic regression
  #
  # Note: n is the number of examples
  #       p is the number of features per example
  #
  # @param data_feature: A matrix with dimension n x p, where each row corresponds to
  #   one data point.
  # @param data_label: A vector of labels with with length equal to n.
  # @param beta: A vector of coefficients with length equal to p.
  # @param beta0: the intercept.
  # @param lbd: the regularization parameter
  #
  # @return: a value of the loss function

  n <- length(data_label)
  proba <- Predict_logis(data_feature, beta, beta0, "prob")

  log_loss <- (-1/n) * sum(data_label * log(proba) + (1 - data_label) * log(1 - proba))
  reg_penalty <- (lbd/2) * sum(beta[-1]^2)

  loss <- log_loss + reg_penalty

  return(loss)
}
```

Part 3: Penalized Logistic Regression Implementation

Part 3 focuses on the main function to fit the Penalized Logistic Regression model. This function iteratively adjusts model parameters (coefficients and intercept) using gradient descent to minimize the penalized logistic loss, and it records the loss and error at each iteration.

```r
Penalized_Logistic_Reg <- function(x_train, y_train, lbd, stepsize, max_iter) {
  # This is the main function to fit the Penalized Logistic Regression
  #
  # Note: n is the number of examples
  #       p is the number of features per example
  #
  # @param x_train: A matrix with dimension n x p, where each row corresponds to
  #   one training point.
  # @param y_train: A vector of labels with length equal to n.
  # @param lbd: the regularization parameter.
  # @param stepsize: the learning rate.
  # @param max_iter: a positive integer specifying the maximal number of
  #   iterations.
  #
  # @return: a list containing four components:
  #   loss: a vector of loss values at each iteration
  #   error: a vector of 0-1 errors at each iteration
  #   beta: the estimated p coefficient vectors
  #   beta0: the estimated intercept.

  p <- ncol(x_train)
```

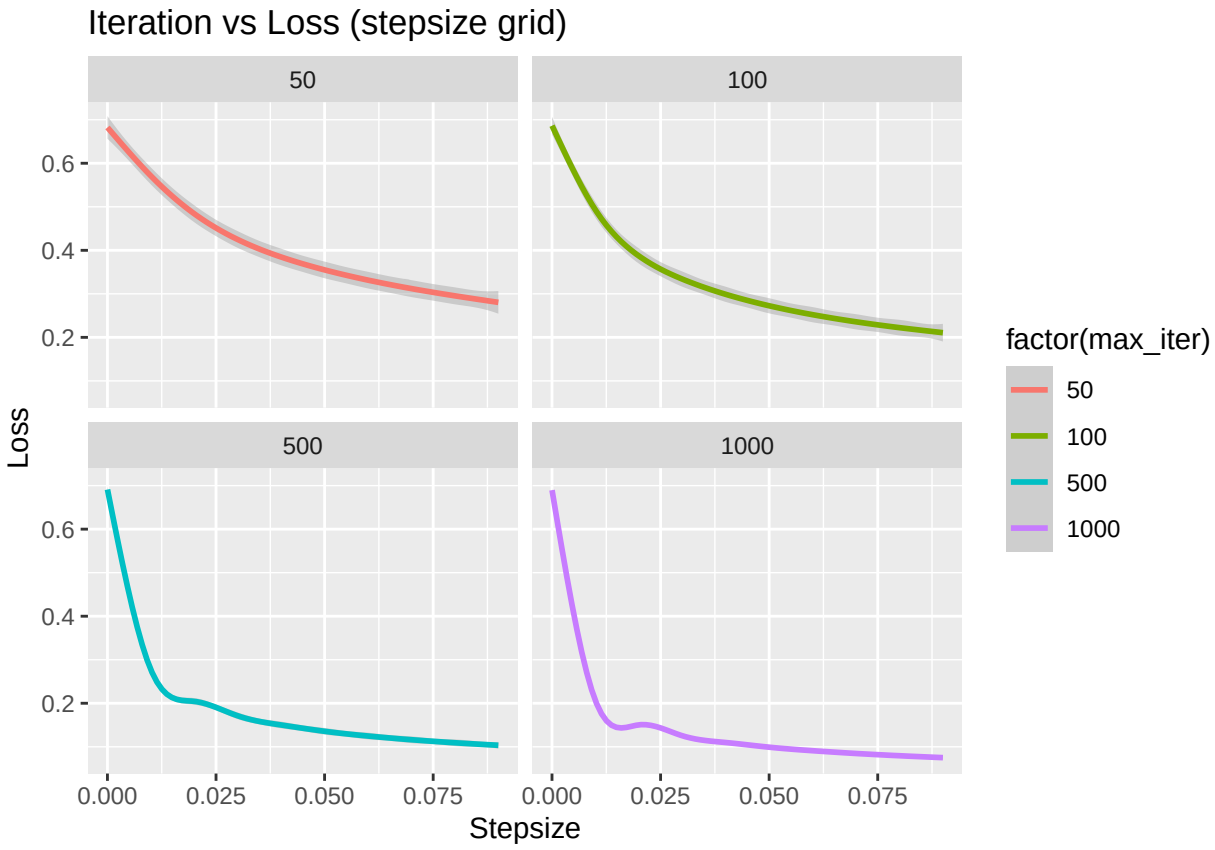
```
  # Initialize parameters to 0
  beta_cur <- rep(0, p)
  beta0_cur <- 0

  # Create the vectors for recording values of loss and 0-1 error during
  # the training procedure
  loss_vec <- rep(0, max_iter)
  error_vec <- rep(0, max_iter)

  # Modify this section to perform gradient descent and to compute    #
  # losses and 0-1 errors at each iterations.                         #

  tolerance <- 1e-6

  for (iter in 1:max_iter) {

    grad <- Comp_gradient(x_train, y_train, beta_cur, beta0_cur, lbd)

    beta_cur <- beta_cur - stepsize * grad[-1]
    beta0_cur <- beta0_cur - stepsize * grad[1]

    loss_vec[iter] <- Comp_loss(x_train, y_train, beta_cur, beta0_cur, lbd)
    error_vec[iter] <- Evaluate(y_train, Predict_logis(x_train, beta_cur, beta0_cur, "class"))

    if (iter > 1 && abs(loss_vec[iter] - loss_vec[iter - 1]) < tolerance) {
      loss_vec <- loss_vec[1:iter]
      error_vec <- error_vec[1:iter]
      break
    }
  }

  return(list("loss" = loss_vec, "error" = error_vec,
              "beta" = beta_cur, "beta0" = beta0_cur))
}
```

Part 4: Hyperparameter Tuning

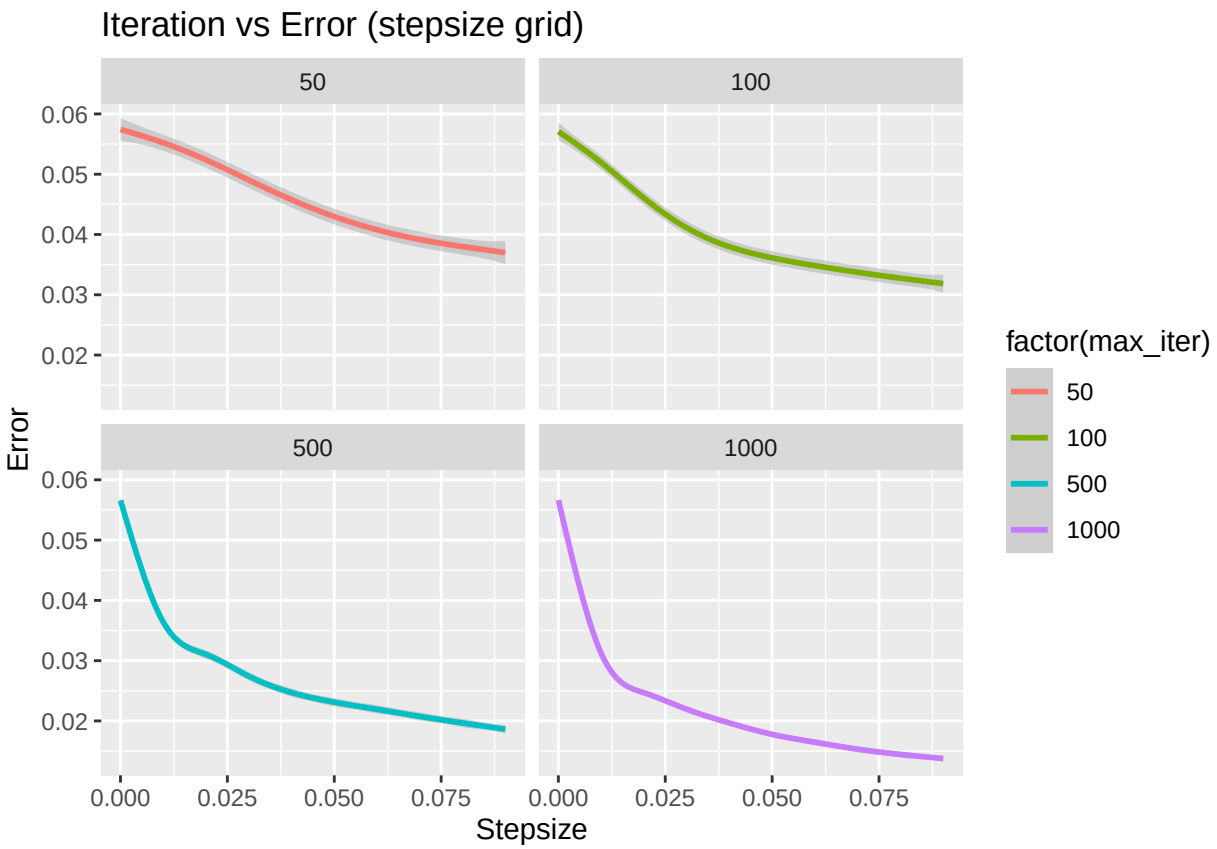
In this part, hyperparameters such as maximum iterations and step size for the logistic regression algorithm are tuned. This involves exploring various combinations of these parameters, evaluating their impact on model performance, and identifying the optimal settings.

```
train <- Load_data(train_data_path)
valid <- Load_data(valid_data_path)
test <- Load_data(test_data_path)

x_train <- train$x
y_train <- train$y

x_valid <- valid$x
y_valid <- valid$y
```

```r
x_test <- test$x
y_test <- test$y

lbd = 0
max_iter_grid <- c(50, 100, 500, 1000)
stepsize_grid <- seq(0.00001, 0.1, 0.01)

results <- data.frame(max_iter= numeric(),
                      stepsize = numeric(),
                      loss = numeric(), error = numeric())

for (max_iter in max_iter_grid) {
for (stepsize in stepsize_grid) {
result <- Penalized_Logistic_Reg(x_train, y_train, lbd, stepsize, max_iter)
results <- rbind(results, data.frame(max_iter = max_iter,
                                     stepsize = stepsize,
                                     loss = result$loss,
                                     error = result$error))
}
}

ggplot(results, aes(x = stepsize, y = loss, color = factor(max_iter))) +
geom_smooth() +
facet_wrap(~max_iter, ncol = 2) +
labs(title = "Iteration vs Loss (stepsize grid)", x = "Stepsize", y = "Loss")
```



```
ggplot(results, aes(x = stepsize, y = error, color = factor(max_iter))) +
geom_smooth() +
facet_wrap(~max_iter, ncol = 2) +
labs(title = "Iteration vs Error (stepsize grid)", x = "Stepsize", y = "Error")
```




```
best_stepsize <- results[which.min(results$error), "stepsize"]
best_max_iter <- results[which.min(results$error), "max_iter"]

paste0("Best stepsize:", best_stepsize)
```

```
## [1] "Best stepsize:0.07001"
```

```
paste0("Best max_iter:", best_max_iter)
```

```
## [1] "Best max_iter:1000"
```

- After testing various settings, the combination of a step size of 0.07001 with 1000 iterations came out as the most effective. This balances the convergence speed and the precision of the penalized logistic regression model.
- The training loss plot indicates a consistent downward trend when the step size is set to 0.07001, suggesting that the model is successfully minimizing the loss function and learning from the data with an increased number of iterations.

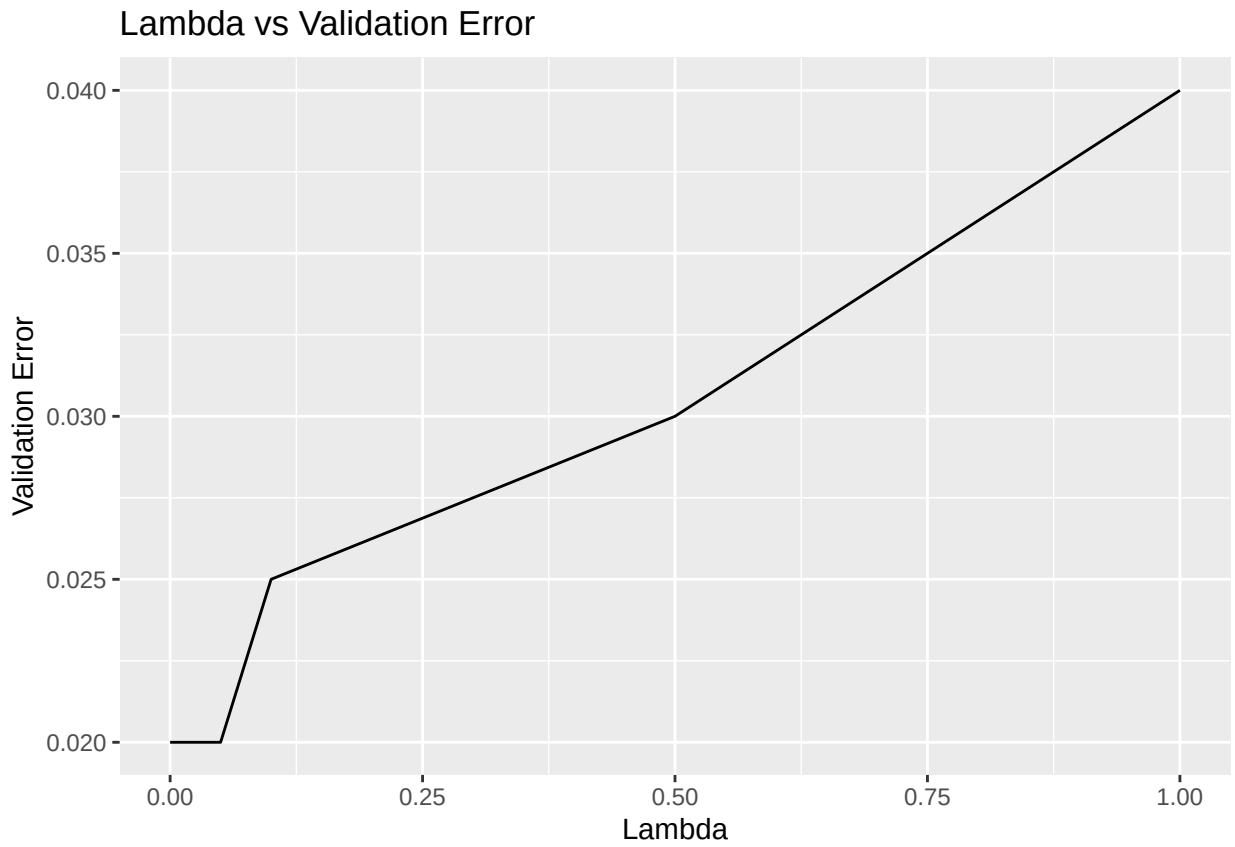
- Corresponding to the loss plot, the training 0-1 error decreases as the number of iterations grows to 1000. This step size appears to allow for sufficient model adjustment without overshooting the minimum error.
- Similar patterns observed in both the training loss and 0-1 error plots are expected. As the model reduces its prediction error, the overall loss decreases. This correlation confirms that the model's predictions are becoming more accurate over iterations. The results seem reasonable, although with more computational power and time, a more exhaustive search in the hyperparameter space could provide a model with even lower error rates and loss.

Part 5: Regularization

Part 5 deals with the regularization aspect of the logistic regression model. It explores the impact of different regularization strengths (lambda values) on the model's performance on training and validation data, aiming to find the optimal balance between model complexity and prediction accuracy.

```
lbd_grid <- c(0, 0.01, 0.05, 0.1, 0.5, 1)

results <- data.frame(lbd = numeric(), stepsize = numeric(), max_iter = numeric(),
                      train_error = numeric(), valid_error = numeric())

for (lbd in lbd_grid) {
result <- Penalized_Logistic_Reg(x_train, y_train, lbd, best_stepsize, best_max_iter)
train_error <- tail(result$error, n=1)
valid_error <- Evaluate(y_valid, Predict_logis(x_valid, result$beta, result$beta0, "class"))
results <- rbind(results, data.frame(lbd = lbd, stepsize = best_stepsize,
                                     max_iter = best_max_iter, train_error = train_error,
                                     valid_error = valid_error))
}

ggplot(results, aes(x = lbd, y = train_error)) +
geom_line() +
labs(title = "Lambda vs Training Error", x = "Lambda", y = "Training Error")
```


```
ggplot(results, aes(x = lbd, y = valid_error)) +
geom_line() +
labs(title = "Lambda vs Validation Error", x = "Lambda", y = "Validation Error")
```




```
best_lambda <- results[which.min(results$valid_error), "lbd"]

paste0("Best lambda:", best_lambda)
```

```
## [1] "Best lambda:0"
```

- The hyperparameters ensured convergence for all values of λ. The steady increase in both training and validation errors across λ values suggests consistent behavior without the need for adjustment.
- The graph displays a clear upward trend in training error as λ increases, indicating that too much regularization may be detrimental to the model's ability to fit the data.
- Similarly, the validation error rises with larger λ values. Initially, a small increase in λ does not significantly affect validation error, but as λ continues to increase, the error rate climbs more sharply.
- The increase in both training and validation errors with higher λ values suggests that the best λ in this context is 0. This is supported by the lowest error rates observed at $\lambda = 0$, indicating that introducing regularization did not benefit this particular model, potentially due to a sufficient amount of training data or the simplicity of the model that does not justify penalization.

Part 6: Comparison with glmnet

This section compares the performance of the custom implementation of Penalized Logistic Regression with the glmnet package's implementation. It evaluates both models on a test dataset and compares their test errors.

```r
base_result <- Penalized_Logistic_Reg(x_train, y_train, best_lambda, best_stepsize, best_max_iter)
test_error <- Evaluate(y_test, Predict_logis(x_test, base_result$beta,
                                             base_result$beta0, "class"))

glmnet_result <- glmnet(x_train, y_train, family = "binomial", alpha = 0, lambda = best_lambda)

predictions <- predict(glmnet_result, newx = x_test, s = best_lambda, type = "response")
predicted_probabilities <- as.vector(predictions)
predicted_classes <- ifelse(predicted_probabilities > 0.5, 1, 0)
glm_net_test_error <- mean(predicted_classes != y_test)

paste0("Test Error:", test_error)
```

```
## [1] "Test Error:0.025"
```

```r
paste0("Test Error:", glm_net_test_error)
```

```
## [1] "Test Error:0.0525"
```

The model fitted using glmnet with the same lambda yielded a higher test error of 0.0525 compared to the base implementation, which had a test error of 0.025.

Part 7: LDA and Naive Bayes

In Part 7, the code implements and evaluates Linear Discriminant Analysis (LDA) and Naive Bayes classifiers on the same dataset. It involves computing class priors, conditional means, and covariance matrices specific to each method, followed by predicting labels and calculating misclassification rates.

```r
# did include=False on the cell with functions from problem 2 containing
# LDA and NB to avoid repeating code and save space

lda_priors <- Comp_priors(train_labels = y_train, K = 2)
lda_means <- Comp_cond_means(train_data = x_train, train_labels = y_train, K = 2)
lda_covs <- Comp_cond_covs(train_data = x_train, train_labels = y_train, K = 2, method = "LDA")
lda_posterior_test <- Predict_posterior(test_data = x_test, priors = lda_priors,
                                        means = lda_means, covs = lda_covs, method = "LDA")
lda_predicted_test <- Predict_labels(lda_posterior_test)
lda_miss_rate_test <- mean(lda_predicted_test != y_test)

nb_priors <- Comp_priors(train_labels = y_train, K = 10)
nb_means <- Comp_cond_means(train_data = x_train, train_labels = y_train, K = 10)
nb_covs <- Comp_cond_covs(train_data = x_train, train_labels = y_train, K = 10, method = "NB")
nb_posterior_test <- Predict_posterior(test_data = x_test, priors = nb_priors,
                                       means = nb_means, covs = nb_covs, method = "NB")
nb_predicted_test <- Predict_labels(nb_posterior_test)
nb_miss_rate_test <- mean(nb_predicted_test != y_test)

paste0("Test Error LDA:", lda_miss_rate_test)
```

```
## [1] "Test Error LDA:0.055"
```

```r
paste0("Test Error NB:", nb_miss_rate_test)
```

```
## [1] "Test Error NB:0.065"
```

Part 8: ROC Curve

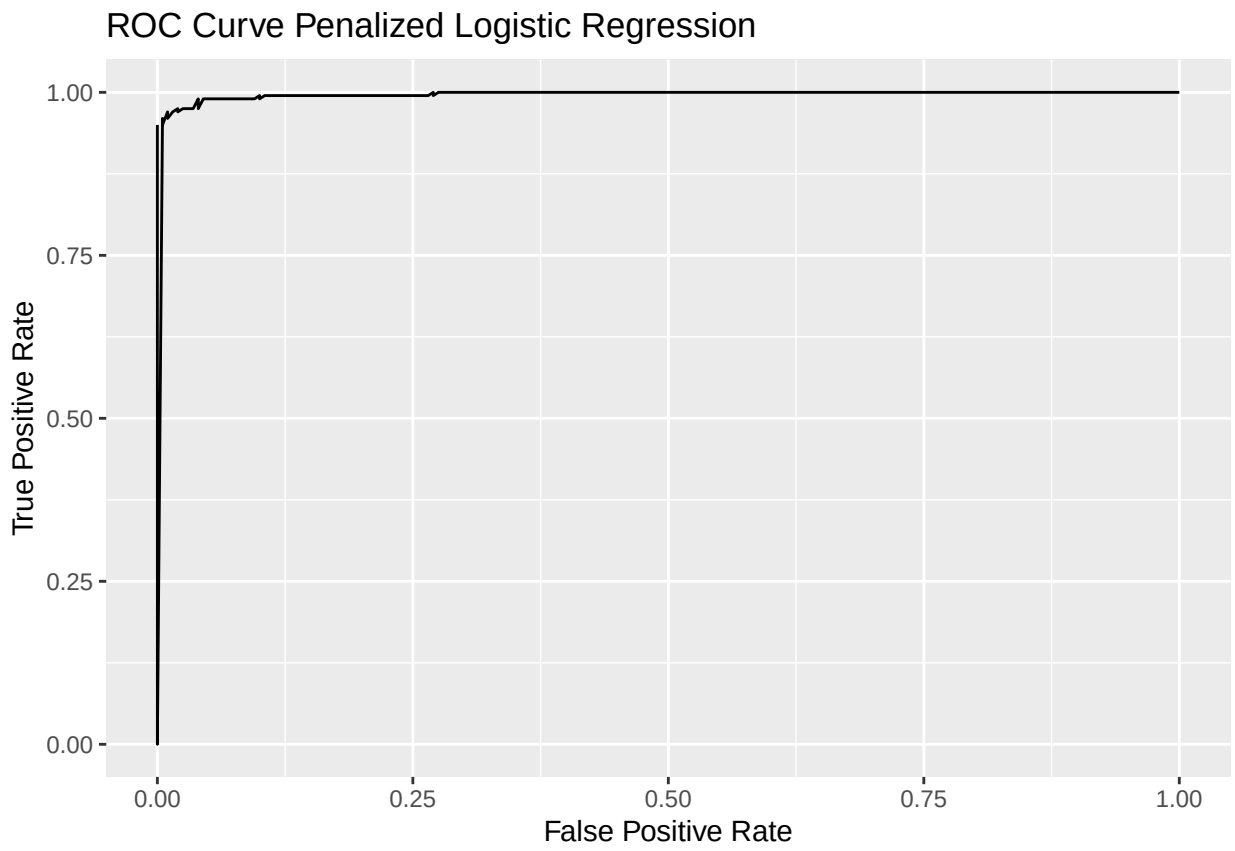
The final part involves plotting ROC curves for the Penalized Logistic Regression, LDA, and Naive Bayes models. It calculates and visualizes the trade-off between sensitivity and specificity for different threshold settings, concluding with a comparison of the area under the curve (AUC) for each model.

```r
base_predictions <- Predict_logis(x_test, base_result$beta, base_result$beta0, "prob")
base_predicted_probabilities <- as.vector(base_predictions)
base_predicted_classes <- ifelse(base_predicted_probabilities > 0.5, 1, 0)
base_roc <- roc(y_test, base_predicted_probabilities)
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

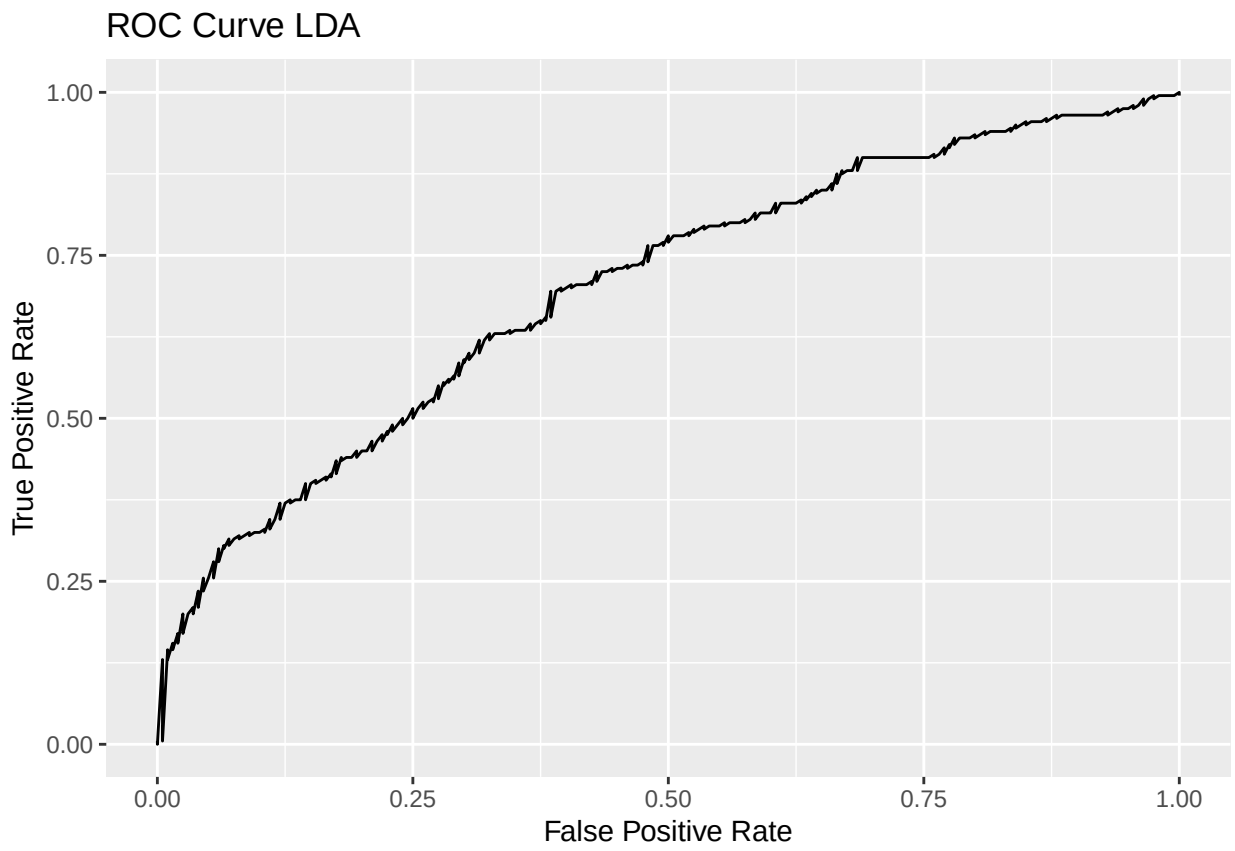
```r
base_roc_df <- data.frame(base_roc$thresholds, base_roc$sensitivities, base_roc$specificities)
colnames(base_roc_df) <- c("thresholds", "sensitivities", "specificities")

ggplot(base_roc_df, aes(x = 1 - specificities, y = sensitivities)) +
geom_line() +
labs(title = "ROC Curve Penalized Logistic Regression", x = "False Positive Rate", y = "True Positive Ra
```




```
lda_predictions <- Predict_posterior(test_data = x_test, priors = lda_priors, means = lda_means, covs =
lda_predicted_probabilities <- as.vector(lda_predictions[, 2])
lda_predicted_classes <- ifelse(lda_predicted_probabilities > 0.5, 1, 0)
lda_roc <- roc(y_test, lda_predicted_probabilities)
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

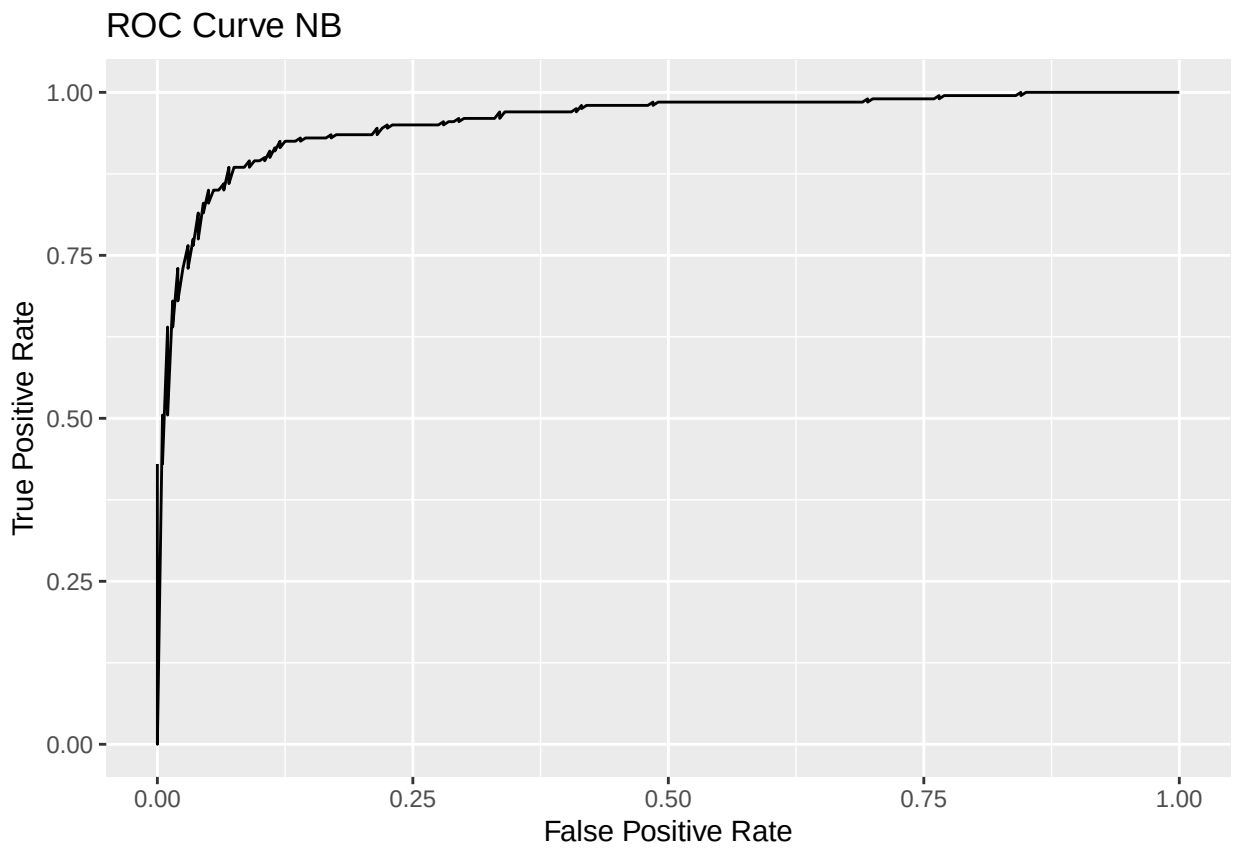
```
lda_auc <- auc(lda_roc)

lda_roc_df <- data.frame(lda_roc$thresholds, lda_roc$sensitivities, lda_roc$specificities)
colnames(lda_roc_df) <- c("thresholds", "sensitivities", "specificities")

ggplot(lda_roc_df, aes(x = 1 - specificities, y = sensitivities)) +
geom_line() +
labs(title = "ROC Curve LDA", x = "False Positive Rate", y = "True Positive Rate")
```




```
nb_predictions <- Predict_posterior(test_data = x_test, priors = nb_priors,
                                    means = nb_means, covs = nb_covs, method = "NB")
nb_predicted_probabilities <- as.vector(nb_predictions[, 2])
nb_predicted_classes <- ifelse(nb_predicted_probabilities > 0.5, 1, 0)
nb_roc <- roc(y_test, nb_predicted_probabilities)
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

```
nb_auc <- auc(nb_roc)

nb_roc_df <- data.frame(nb_roc$thresholds, nb_roc$sensitivities, nb_roc$specificities)
colnames(nb_roc_df) <- c("thresholds", "sensitivities", "specificities")

ggplot(nb_roc_df, aes(x = 1 - specificities, y = sensitivities)) +
geom_line() +
labs(title = "ROC Curve NB", x = "False Positive Rate", y = "True Positive Rate")
```




```
auc_table <- data.frame(model = c("Penalized Logistic Regression", "LDA", "NB"),
                        auc = c(base_roc$auc, lda_auc, nb_auc))
pander::pander(auc_table)
```

model	auc
Penalized Logistic Regression	0.9973
LDA	0.6998
NB	0.9562