Using Text as Predictor for Regression

Promothesh Chatterjee[*]

[*]Copyright© by Promothesh Chatterjee. All rights reserved. No part of this note may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the author.

Text as a Predictor

Through out the semester so far we have looked at various ways in which to utilize text for business analytics. One important part of analytics is predictive analytics where we are trying to predict a variable of business interest using text as predictors. This variable if it is categorical (e.g. satisfied versus not satisfied), we utilize classification techniques as Naive Bayes and others that we have covered in the past two weeks. In this week, we will look at other techniques that can be used for predicting continuous variables (e.g. price). Please note that some techniques that we have talked about previously such as Support Vector Machines can be utilized for continuous variables with some modifications. Others, such as using sentiment scores or truncated dataset based on Latent Semantic Analysis as discussed previously can also be used. Before introducing other techniques, I want to do a quick recap of these focusing mostly on the R code.

Since we will using a continuous dependent variable in the predictive tasks, let's modify our hotel reviews dataset to create a ratings variable (which is not strictly continuous but let's assume it is).

Creating the Ratings Variable in the dataset

```r
library(tidyverse)
hotel_review<-read_csv("Data/hotel-reviews.csv")
hotel_review<-sample_n(hotel_review, 5000)

# Convert binary variable, Is_Response to a continuous ratings variable
convert_fn <- function(x) {
  sapply(x, function(y) {
    if (y == "not happy") {
      return(sample(1:2, size = 1, prob = c(0.5, 0.5)))
    } else { # y == "happy"
      return(sample(3:5, size = 1, prob = c(0.33, 0.33, 0.34)))
    }
  })
}

hotel_review$ratings <- convert_fn(hotel_review$Is_Response)
head(hotel_review)
```

```
## # A tibble: 6 x 6
##   User_ID Description                Browser_Used Device_Used Is_Response ratings
##   <chr>   <chr>                      <chr>        <chr>       <chr>         <int>
## 1 id19075 "I enjoy staying here wh~ Google Chro~ Mobile      happy             3
## 2 id40632 "Even if you pay for you~ Edge         Desktop     not happy         2
## 3 id16314 "With a large room on th~ InternetExp~ Desktop     happy             5
## 4 id25493 "Great hotel. Large room~ Safari       Tablet      happy             3
## 5 id19431 "Stayed at Blue Moon ove~ Mozilla      Mobile      happy             5
## 6 id44314 "The Standard Hotel room~ Firefox      Tablet      not happy         2
```

Create Train and Test Datasets

Let's use library caret to create a train and test datasets using createDataPartition function.

```r
library(caret)
library(magrittr)

# Set the seed for reproducibility
set.seed(123)

# Define the proportion of data you want to keep for training
trainIndex <- createDataPartition(hotel_review$ratings, p = 0.70, list = FALSE, times = 1)

# Create the training set
train_set <- hotel_review[trainIndex,]

# Create the testing set
test_set <- hotel_review[-trainIndex,]
```

1. Using Sentiment Scores as a Proxy for Text Data

We have seen in the sentiment analysis chapter, how to use sentiment scores as a predictor, so I am just focusing on the R code.

```r
library(sentimentr)

t1 <- train_set %>%
    mutate(review = get_sentences(Description)) %$%
    sentiment_by(review, User_ID)

train_set1 <- train_set %>% left_join(t1,by = "User_ID")

Train <- train_set1 %>% select(User_ID,ratings, word_count, ave_sentiment)

ctrl <- trainControl(method="cv", number=10)

# Create a linear regression model
model <- train(ratings ~ word_count+ ave_sentiment, data=Train, method="lm", trControl=ctrl)

# Print model results
print(model)
```

Training the model

```
## Linear Regression
##
## 3502 samples
##    2 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 3152, 3152, 3152, 3152, 3152, 3151, ...
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   1.199396  0.2393671  0.9927248
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
summary(model)
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.7466 -0.8365 -0.0802  0.9411  2.9903
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)    2.5336278  0.0432837  58.535  < 2e-16 ***
## word_count    -0.0005032  0.0001585  -3.175  0.00151 **
## ave_sentiment  3.4022184  0.1095758  31.049  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.2 on 3499 degrees of freedom
## Multiple R-squared:  0.2371, Adjusted R-squared:  0.2367
## F-statistic: 543.7 on 2 and 3499 DF,  p-value: < 2.2e-16
```

```r
t2 <- test_set %>%
    mutate(review = get_sentences(Description)) %$%
    sentiment_by(review, User_ID)

test_set1 <- test_set %>% left_join(t2,by = "User_ID")

Test <- test_set1 %>% select(User_ID,ratings, word_count, ave_sentiment)

# Generate predictions on test data
predictions <- predict(model, newdata = Test)

# Calculate R-squared and RMSE
rsq_rmse <- postResample(pred = predictions, obs = Test$ratings)

# Print the results
print(rsq_rmse)
```

Testing the model

```
##      RMSE  Rsquared       MAE
## 1.1918549 0.2541856 0.9920377
```

2. Using Truncated LSA data for Regression

Now we come to one of the most interesting uses of LSA. Recall, after the truncated Singular Value Decomposition, we have 3 matrices as the output: reduced term matrix, reduced document matrix and the singular matrix. We can use the reduced document matrix as a proxy for the original DTM. This would be very useful for downstream predictive analysis as huge dimensions and sparsity of matrix would have been taken care. In our previous chapter (Classification), we have used library IRLBA to perform LSA, in this one let's perform using linear regression with the reduced document matrix from library quanteda.textmodels to predict ratings.

```
library(caret)
library(quanteda)
review_corpus<-corpus(hotel_review, text_field = "Description")

hotel_review_token <- tokens(review_corpus, what = "word",
                             remove_numbers = TRUE, remove_punct = TRUE,
                             remove_symbols = TRUE)
#Create DFM
hotel_review_tfidf<-hotel_review_token %>%
    tokens_remove(stopwords(source = "smart")) %>%
    tokens_wordstem() %>%
    tokens_tolower() %>%
    dfm() %>%
```

```r
    dfm_trim( min_termfreq = 50, min_docfreq = 10) %>%
    dfm_tfidf()

# we don't need to transpose here as we will use other library

library("quanteda.textmodels")
mylsa <- textmodel_lsa(hotel_review_tfidf, nd = 100)

new.dataset <- data.frame(mylsa$docs)
new.dataset$y.var <- hotel_review$ratings
```

Fitting a Predictive Model

Let's use the truncated dataset in a predictive model. First we need to split the original dataset into train and test.

```r
test_index <- createDataPartition(new.dataset$y.var, p = .2, list = F)
Test_LSA<-new.dataset[test_index,]
Train_LSA<-new.dataset[-test_index,]

ctrl <- trainControl(method="cv", number=10)
#Training data
# Create a linear regression model
model <- train(y.var ~ ., data=Train_LSA, method="lm", trControl=ctrl)
summary(model)
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.2424 -0.8219 -0.0634  0.8736  3.3006
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)    3.095647   0.037457  82.645  < 2e-16 ***
## X1            -3.984075   2.446606  -1.628 0.103519
## X2           -34.793670   1.409382 -24.687  < 2e-16 ***
## X3            -5.144502   1.338083  -3.845 0.000123 ***
## X4           -22.718898   1.402345 -16.201  < 2e-16 ***
## X5            -6.101969   1.387680  -4.397 1.13e-05 ***
## X6            -3.757179   1.382433  -2.718 0.006601 **
## X7            -0.242805   1.387621  -0.175 0.861105
## X8            -2.927783   1.371324  -2.135 0.032823 *
## X9            -0.472329   1.332455  -0.354 0.722998
## X10            1.865042   1.307453   1.426 0.153813
## X11            0.523717   1.346393   0.389 0.697314
## X12            1.391946   1.416464   0.983 0.325821
## X13            6.159788   1.493938   4.123 3.81e-05 ***
## X14            2.987590   1.424121   2.098 0.035983 *
## X15           -5.405002   1.386490  -3.898 9.85e-05 ***
## X16          -11.638499   1.549501  -7.511 7.23e-14 ***
## X17           -4.065380   1.414017  -2.875 0.004061 **
## X18            1.437383   1.350152   1.065 0.287119
## X19            1.564207   1.471706   1.063 0.287915
## X20           -3.576902   1.355011  -2.640 0.008330 **
## X21           -5.128165   1.348604  -3.803 0.000145 ***
## X22           -7.926845   1.392530  -5.692 1.34e-08 ***
## X23            3.724242   1.337086   2.785 0.005373 **
## X24            1.734107   1.410521   1.229 0.218993
## X25           -7.071487   1.336862  -5.290 1.29e-07 ***
## X26            6.597335   1.333038   4.949 7.77e-07 ***
## X27            1.719884   1.436098   1.198 0.231142
## X28           -2.313952   1.365068  -1.695 0.090133 .
## X29           -1.129871   1.385415  -0.816 0.414809
## X30           -1.443635   1.363662  -1.059 0.289827
## X31            2.599508   1.317182   1.974 0.048505 *
## X32           -3.917510   1.384203  -2.830 0.004676 **
## X33           -2.001372   1.344575  -1.488 0.136705
## X34            3.716874   1.332503   2.789 0.005306 **
## X35            0.085904   1.349094   0.064 0.949232
## X36            0.220397   1.395765   0.158 0.874541
## X37            3.673673   1.351319   2.719 0.006585 **
## X38           -1.232843   1.382646  -0.892 0.372633
## X39            0.005291   1.348879   0.004 0.996870
## X40            2.678917   1.350883   1.983 0.047428 *
## X41            2.626879   1.334872   1.968 0.049152 *
## X42            1.728817   1.393674   1.240 0.214875
## X43           -0.052126   1.334586  -0.039 0.968846
## X44            4.089414   1.334451   3.064 0.002195 **
## X45           -0.044762   1.395568  -0.032 0.974414
## X46            2.210028   1.340053   1.649 0.099185 .
## X47           -3.891438   1.344445  -2.894 0.003819 **
## X48            0.299772   1.330550   0.225 0.821758
## X49            1.099339   1.338746   0.821 0.411599
## X50           1.496841   1.350330   1.109 0.267714
## X51          -0.201835   1.333092  -0.151 0.879665
## X52          -4.276362   1.325491  -3.226 0.001265 **
## X53           2.640953   1.367377   1.931 0.053506 .
## X54          -4.411999   1.372012  -3.216 0.001312 **
## X55           3.271765   1.358763   2.408 0.016091 *
## X56           2.616870   1.366773   1.915 0.055612 .
## X57          -2.457062   1.345576  -1.826 0.067922 .
## X58          -1.585513   1.344434  -1.179 0.238344
## X59           1.195821   1.383063   0.865 0.387302
## X60          -1.400851   1.339964  -1.045 0.295885
## X61          -4.043881   1.350672  -2.994 0.002771 **
## X62          -0.168039   1.351710  -0.124 0.901072
## X63           1.832280   1.360535   1.347 0.178144
## X64          -3.938088   1.383639  -2.846 0.004448 **
## X65           2.657402   1.363822   1.948 0.051427 .
## X66           3.060313   1.325296   2.309 0.020987 *
## X67           1.786440   1.348566   1.325 0.185350
## X68           2.517720   1.359830   1.851 0.064174 .
## X69          -2.142399   1.381425  -1.551 0.121016
## X70          -4.767752   1.343074  -3.550 0.000390 ***
## X71           1.388793   1.326120   1.047 0.295045
## X72          -2.427489   1.350499  -1.797 0.072337 .
## X73           0.186822   1.362887   0.137 0.890976
## X74          -4.209730   1.350392  -3.117 0.001838 **
## X75           0.743051   1.336408   0.556 0.578239
## X76          -1.896673   1.337644  -1.418 0.156294
## X77           1.439866   1.358651   1.060 0.289312
## X78          -0.722275   1.351904  -0.534 0.593189
## X79           2.522376   1.318013   1.914 0.055722 .
## X80           1.845215   1.334518   1.383 0.166841
## X81           4.506735   1.326527   3.397 0.000687 ***
## X82           3.840550   1.367994   2.807 0.005019 **
## X83          -1.558614   1.358557  -1.147 0.251346
## X84          -2.677249   1.336890  -2.003 0.045290 *
## X85           0.410252   1.373811   0.299 0.765244
## X86           2.314382   1.342818   1.724 0.084873 .
## X87          -1.225254   1.336701  -0.917 0.359396
## X88          -3.052734   1.349558  -2.262 0.023751 *
## X89          -1.328679   1.340434  -0.991 0.321635
## X90           1.150221   1.362289   0.844 0.398537
## X91           3.882218   1.344448   2.888 0.003903 **
## X92           0.495076   1.357785   0.365 0.715414
## X93           3.629610   1.358976   2.671 0.007598 **
## X94          -1.886634   1.389661  -1.358 0.174662
## X95          -0.469396   1.335973  -0.351 0.725344
## X96          -0.796851   1.348159  -0.591 0.554511
## X97          -1.539787   1.346480  -1.144 0.252874
## X98          -4.850199   1.372115  -3.535 0.000413 ***
## X99          -2.264107   1.355792  -1.670 0.095009 .
## X100         -0.817181   1.344285  -0.608 0.543294
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.185 on 3897 degrees of freedom
## Multiple R-squared:  0.278,  Adjusted R-squared:  0.2595
## F-statistic: 15.01 on 100 and 3897 DF,  p-value: < 2.2e-16
```

```r
# Generate predictions on test data
predictions1 <- predict(model, newdata = Test_LSA)

# Calculate R-squared and RMSE
rsq_rmse1 <- postResample(pred = predictions1, obs = Test_LSA$y.var)

# Print the results
print(rsq_rmse1)
```

```
##      RMSE  Rsquared       MAE
## 1.1790753 0.2574155 0.9770018
```

Ok results on the test data, perhaps something to do with the way we created the ratings data.

3. Using Support Vector Regression with Text as Predictors

In the classification chapter, we looked at Support Vector Machine (SVM). Support Vector Regression (SVR) is a type of Support Vector Machine (SVM) that is used for regression problems, as opposed to classification problems which are more commonly associated with SVMs.

The main idea behind SVR is to find a function that can predict the continuous dependent variable with a given amount of error. The objective is to fit the error within a certain threshold, meaning the model doesn't care about errors as long as they fall within a predefined range, often called the "epsilon-insensitive tube". This tube is a range within which deviations are tolerated, i.e., errors within this range do not contribute to the model's loss function.

The methodology of SVR is based on the computation of a hyperplane which can predict continuous values, rather than classifying data into to classes. The objective of SVR is to minimize the coefficients — more specifically, the l2-norm of the coefficient vector — not the squared error. The smaller the coefficients, the flatter (and therefore more robust) the resulting model will be.

In SVR, a margin of tolerance (epsilon) is set in approximation to the SVM. The main idea is to find an epsilon-soft margin hyperplane in the high dimensional feature space that the amount of data points residing outside the margin is minimized.

Key parameters in SVR include:

- **C**: Also known as the "cost parameter", this determines the trade-off between allowing training errors and forcing rigid margins. A smaller value of C creates a wider margin but may allow more misclassifications. A larger value of C creates a narrower margin and fewer misclassifications.
- **Epsilon**: This defines the epsilon-insensitive tube within which no penalty is assigned to errors.
- **Kernel**: The function used to map the data into a higher dimensional space. Common choices include linear, polynomial, and radial basis function (RBF) kernels. The choice of kernel can significantly affect the performance of the model.

SVR has been found to be useful in many real-world applications, including time series prediction, handwriting recognition, and image analysis, among others.

Let's run SVR on our dataset. We will take the previously created DFM called the hotel_review_tfidf and use test_index to split the DFM into train and test DFMs.

```
hotel_review_tfidf1 <- convert(hotel_review_tfidf, to ="data.frame")
hotel_review_tfidf1 <- data.frame(y.var=hotel_review$ratings,hotel_review_tfidf1 )

hotel_review_tfidf1 <- hotel_review_tfidf1 %>% select(-doc_id)

test_index <- createDataPartition(hotel_review_tfidf1$y.var, p = .2, list = F)

test_text <- hotel_review_tfidf1[test_index, ]
train_text <- hotel_review_tfidf1[-test_index, ]

## Set up repeated k-fold cross-validation
train_control <- trainControl(method = "cv", number = 2)

# Fit the SVR model
set.seed(12345)
SVM_Linear <- train(
  y.var ~., data = train_text, method = "svmLinear",
  trControl = train_control,
  preProcess = c("center","scale"),
  tuneLength = 7
  )

# Predict on the test set
predictions <- predict(SVM_Linear, newdata = test_text)
```

```r
# Evaluate the model
postResample(pred = predictions, obs = test_text$y.var)
```

```
##      RMSE  Rsquared       MAE
## 1.6518176 0.1109201 1.2891751
```

3. **Using Topic Proportions as Predictors**

Recall that topic modeling is a technique used to discover abstract topics within a collection of documents. The most common method for this is Latent Dirichlet Allocation (LDA), which represents each document as a mixture of a fixed number of topics, and each topic as a mixture of words.

To use topic modeling as a predictor in a regression model, we need to first perform topic modeling on the corpus of text data. Each document is then represented as a vector of topic proportions, which can be used as predictor variables in a regression model.

Here's a general step-by-step guide of how you can do this:

1. **Preprocess the Text**: Clean your text data by removing stop words, punctuation, and applying other preprocessing steps such as stemming or lemmatization.
2. **Build the Topic Model**: Use a topic modeling method (like LDA) to extract topics from your text data.
3. **Extract Topic Proportions**: For each document, get the proportion of each topic. This will be a matrix where each row represents a document and each column represents a topic.
4. **Build the Regression Model**: Use the topic proportions as predictor variables in the regression model. The response variable would be whatever outcome we are interested in predicting. One can use any regression method for this, such as linear regression, logistic regression, or more complex models.

Remember that the number of topics is a hyperparameter that you will need to choose. Perplexity scores, or topic coherence measures can help inform this choice.

Also, remember that the topics are derived from the data and may not always correspond to intuitive or easily interpretable themes. It's often a good idea to examine the most representative words for each topic to get a sense of what they might represent.

Let's see how we can do this in R. We will use the previously created hotel_review_tfidf d fm f or topic modeling. Let's take number of topics as 8.

```r
set.seed(1234)
library(seededlda)
# Train a topic model with 5 topics
lda <- seededlda::textmodel_lda(hotel_review_tfidf, k = 8)
```

Now, let's create the final dataset from the LDA output for regression. lda$theta contains the topic proportions for each document. We will use the topic proportions as predictors.

```r
df <- data.frame(y.var=hotel_review$ratings,lda$theta )
```

Let's run the linear regression model

```r
model <- lm(y.var ~ ., data = df)
# Print the summary of the model
summary(model)
```

```
##
## Call:
## lm(formula = y.var ~ ., data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.2093 -0.8614 -0.0402  0.9611  3.4842
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    7.473      1.204   6.207 5.84e-10 ***
## topic1        -6.317      1.215  -5.199 2.08e-07 ***
## topic2        -6.641      1.220  -5.444 5.46e-08 ***
## topic3        -3.203      1.220  -2.625 0.008688 **
## topic4        -3.681      1.224  -3.007 0.002651 **
## topic5        -4.126      1.233  -3.345 0.000828 ***
## topic6        -2.781      1.231  -2.260 0.023857 *
## topic7        -4.572      1.227  -3.725 0.000197 ***
## topic8        -3.375      1.232  -2.740 0.006169 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.203 on 4991 degrees of freedom
## Multiple R-squared:  0.2354, Adjusted R-squared:  0.2342
## F-statistic: 192.1 on 8 and 4991 DF,  p-value: < 2.2e-16
```

I haven't shown the cross-validation process here but you can follow the usual procedure of splitting the DFM into train and test, and then adding the dependent variable to create the final dataset to train the model before testing.

4. **Using Ridge, Lasso and Elastic Net Regression with Text as Predictors**

Let's understand some background information about regression framework before we get into the details of these regression techniques.

Loss Function: In regression analysis, a loss function is a mathematical function that quantifies how well the predictions of a model match the actual values. It measures the discrepancy between the predicted and actual outcome values, with the goal of minimizing this difference.

In the context of regression, one of the most common loss functions is the Mean Squared Error (MSE) loss function. It calculates the average of the squared differences between the predicted and actual values. Mathematically, it is represented as:

`MSE = 1/n Σ(y_i - ŷ_i)^2`

where:

- `n` is the number of observations,
- `y_i` is the actual value of the outcome variable for observation `i`, and
- `ŷ_i` is the predicted value of the outcome variable for observation `i`.

The goal of a regression model is to minimize this MSE loss function - in other words, to find the model parameters that make the predicted values as close as possible to the actual values, according to the MSE measure.

Another common loss function used in regression is the Mean Absolute Error (MAE), which calculates the average of the absolute differences between the predicted and actual values.

`MAE = 1/n Σ|y_i - ŷ_i|`

Each of these loss functions has different properties and may be preferred in different situations, depending on the specific requirements of the analysis. For example, MSE is more sensitive to large errors due to the squaring operation, whereas MAE is more robust to outliers.

Regularization: Regularization in regression is a technique used to prevent overfitting by adding a penalty term to the loss function. Overfitting occurs when a model learns from the noise and outliers in the training data to the extent that it negatively impacts the performance of the model on new, unseen data. Regularization discourages learning a more complex or flexible model, thus reducing overfitting.

In the linear regression framework, we want to find an equation Y = Intercept + a*X1 + b*X2 + c*X3, where "a", "b", and "c" are our coefficients. Thus, the goal of linear regression is to find the values of intercept, a, b, and c that minimize MSE (or sums of squares).

There are two common types of regularization in regression:

1. **Ridge Regression (L2 Regularization)**:

   In Ridge Regression, we add the penalty term $\lambda(a^2 + b^2 + c^2)$ to the loss function. Thus, instead of minimizing MSE/SSR, we are aiming to find the intercept, a, b, and c that minimize MSE + $\lambda(a^2 + b^2 + c^2)$.

   In other words, we add the squared magnitude of coefficients as a penalty term to the loss function. The formula is:

   Loss function + λ * (sum of square of coefficients)

2. **Lasso Regression (L1 Regularization)**: In Lasso Regression, we add the penalty term $\lambda(\|a\|+\|b\|+\|c\|)$ to the loss function. Thus, instead of minimizing MSE/SSR, we are aiming to find the intercept, a, b, and c that minimize MSE + $\lambda(\|a\|+\|b\|+\|c\|)$.

   In other words, we add λ times the sum of the absolute values of the coefficients as a penalty term to the loss function. The formula is:

   Loss function + λ * (sum of absolute value of coefficients)‘

In both formulas, λ is a tuning hyperparameter. A larger λ provides a larger penalty, and thus a more regularized model. In both cases, the value of λ is typically chosen via cross-validation: choose the λ that minimizes prediction error on a held-out set of data.

These two types of regularization have slightly different e ffects: Ridge Regression can shrink the coefficients close to, but not exactly, zero. This can be useful if we believe many features should have non-zero influence on the outcome. On the other hand, Lasso Regression can shrink some coefficients to ze ro, effectively performing variable selection.

3. **Elastic Net regression.** Third type of regression is Elastic Net which is a type of linear regression that combines the penalties of Ridge regression and Lasso regression to overcome some of their limitations and to create a balance between them.

The Elastic Net adds both L1 (used in Lasso) and L2 (used in Ridge) penalties to the loss function:

Loss function + λ_1 * (sum of absolute value of coefficients) + λ_2 * (sum of square of coefficients)

Here, λ_1 and λ_2 are tuning parameters that control the strength of the L1 and L2 penalties, respectively. In practice, instead of tuning two parameters, it’s common to fix a ratio α between the L1 and L2 penalty terms, and then tune a single λ parameter.

The effect of Elastic Net is a blend of Lasso and Ridge:

- Like Lasso, it can shrink some coefficients to zero, effectively performing feature selection and resulting in a sparse model.
- Like Ridge, it can also shrink coefficients towards each other, thus handling multicollinearity better.

This makes Elastic Net a flexible tool that can be particularly useful when dealing with datasets where you have a large number of features, and especially when these features are correlated.

It’s also worth noting that, as with Lasso and Ridge, the value of the tuning parameters (λ_1, λ_2 or λ and α) is typically chosen via cross-validation: choose the parameters that minimize prediction error on a held-out set of data.

Let’s see how we can run these in R:

```r
# Load required libraries
library(glmnet)
library(caret)
library(quanteda)

set.seed(12345)

hotel_record <- createDataPartition(hotel_review$ratings, times = 1,
                                    p = 0.7, list = FALSE)

hotel_train <- hotel_review[hotel_record,] # using hotel_record for indexing
hotel_test <- hotel_review[-hotel_record,]

df_training <- tokens(hotel_train$Description) %>%
    dfm() %>%
    dfm_remove(stopwords("english"))%>%
    dfm_wordstem ()

df_test <- tokens(hotel_test$Description) %>%
    dfm() %>%
    dfm_remove(stopwords("english"))%>%
    dfm_wordstem ()


### Train the model

# Set up the outcome variable
y <- hotel_train$ratings

# Create a sparse matrix from the dfm
x <- as(df_training, "sparseMatrix")

# Ridge regression with cross-validation (alpha = 0)
cv_ridge <- cv.glmnet(x, y, alpha = 0)

# Lasso regression with cross-validation (alpha = 1)
cv_lasso <- cv.glmnet(x, y, alpha = 1)

# Elastic Net regression with cross-validation (alpha = 0.5)
cv_elastic_net <- cv.glmnet(x, y, alpha = 0.5)
```

The **`cv.glmnet()`** function performs k-fold cross-validation by default, where k = 10. You can change the number of folds by specifying the **`nfolds`** parameter (e.g., **`nfolds = 5`** for 5-fold cross-validation).

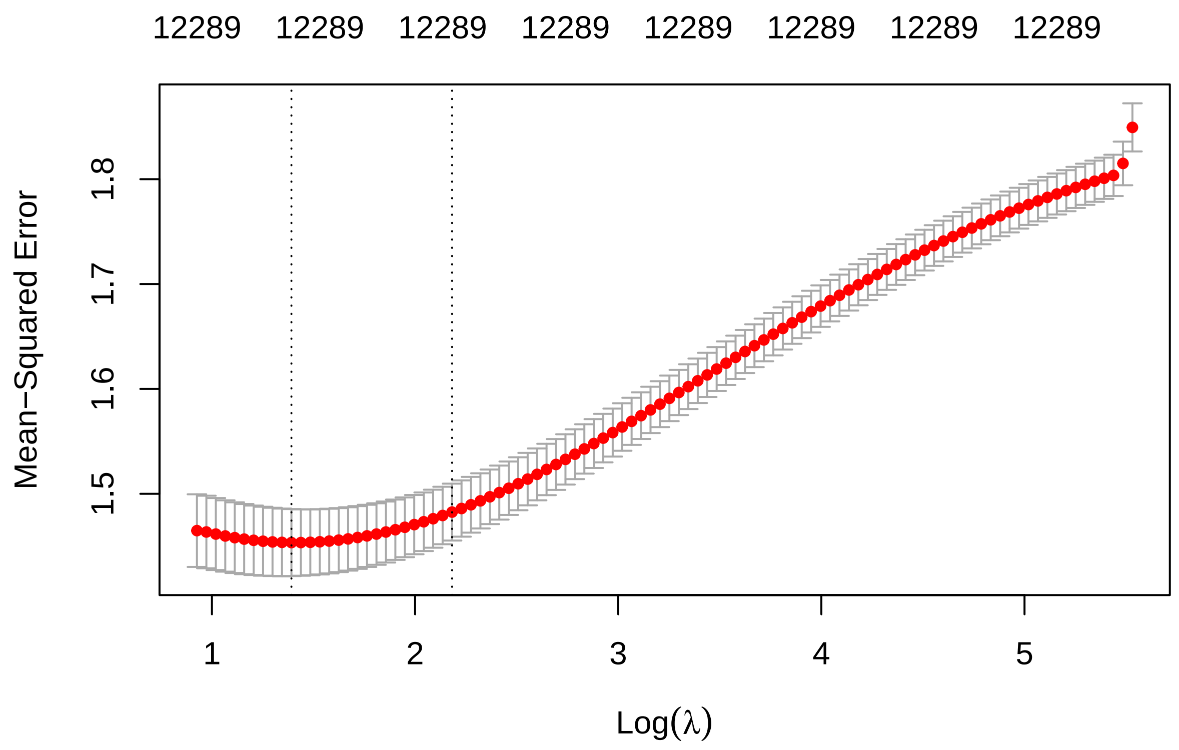
After running this code, **`cv_ridge`**, **`cv_lasso`**, and **`cv_elastic_net`** will contain the results of the cross-validation for the Ridge, Lasso, and Elastic Net models, respectively. You can use the **`coef()`** function to get the coefficients at the optimal value of lambda (the regularization parameter), e.g., **`coef(cv_ridge, s`**

`= "lambda.min")`. You can also plot the cross-validation results with **`plot(cv_ridge)`**.

Here's how you can get the coefficients at the optimal lambda and plot the cross-validation results.

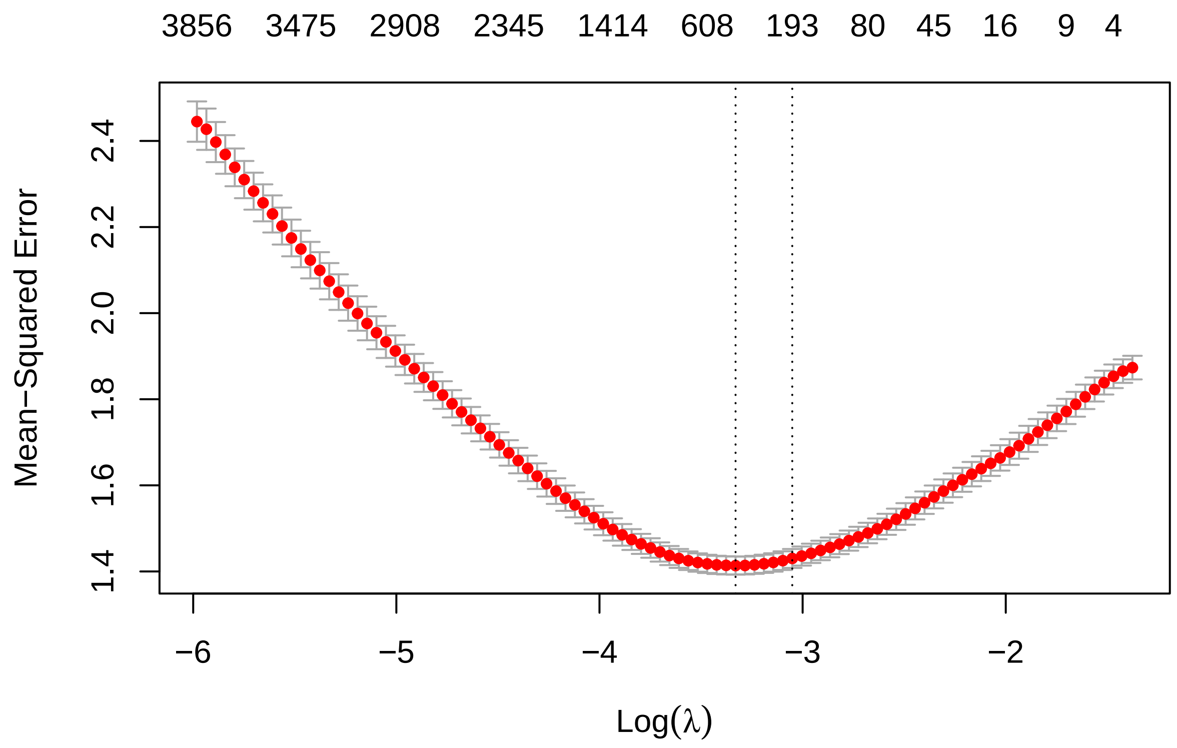
For Ridge regression:

```r
# Get coefficients at the optimal lambda for Ridge regression
best_coefs_ridge <- coef(cv_ridge, s = "lambda.min")
#print(best_coefs_ridge)

# Plot cross-validation results for Ridge regression
plot(cv_ridge)
```



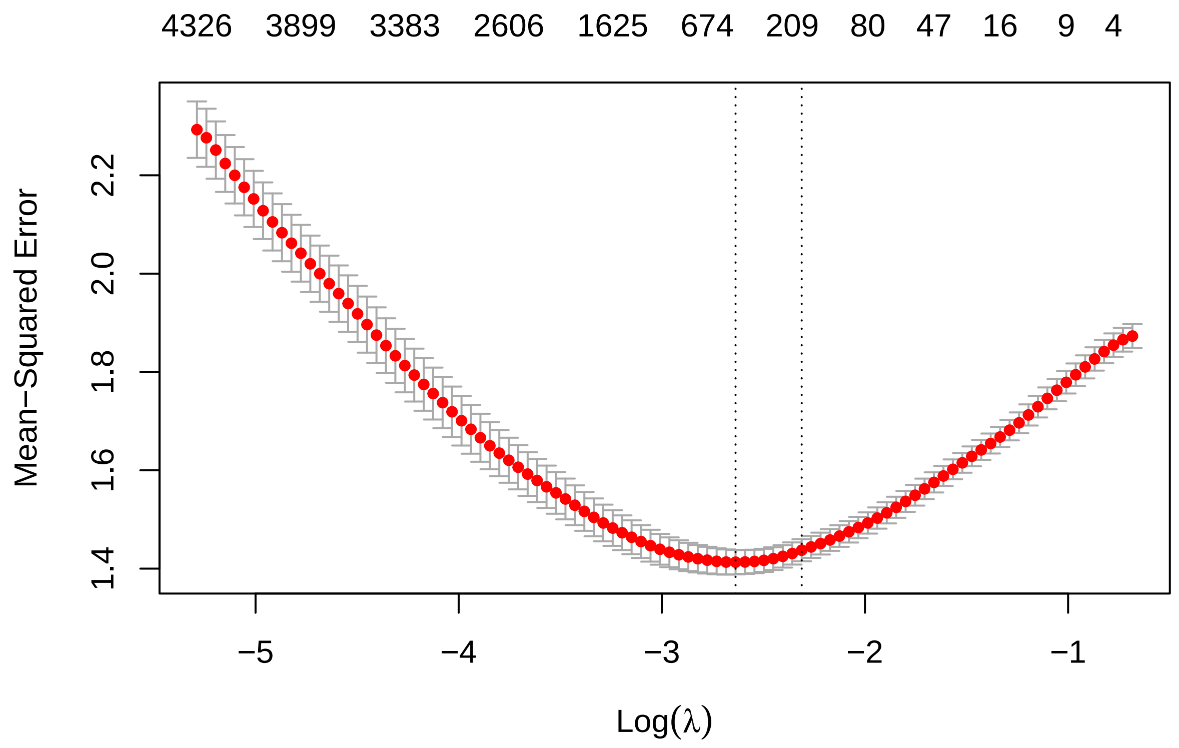
For Lasso regression:

```r
# Get coefficients at the optimal lambda for Lasso regression
best_coefs_lasso <- coef(cv_lasso, s = "lambda.min")
#print(best_coefs_lasso)

# Plot cross-validation results for Lasso regression
plot(cv_lasso)
```



For Elastic Net regression:

```
# Get coefficients at the optimal lambda for Elastic Net regression
best_coefs_elastic_net <- coef(cv_elastic_net, s = "lambda.min")
#print(best_coefs_elastic_net)

# Plot cross-validation results for Elastic Net regression
plot(cv_elastic_net)
```



In these code snippets, **`s = "lambda.min"`** is used to get the coefficients at the value of lambda that gives the minimum mean cross-validated error. You can change this to **`s = "lambda.1se"`** to get the coefficients at the most regularized model within one standard error of the minimum.

The **`plot()`** function will produce a plot showing the mean cross-validated error as a function of lambda. The vertical dotted lines show the optimal lambda (either **`lambda.min`** or **`lambda.1se`**).

Once you've determined the optimal value of lambda using cross-validation, you can fit the final model using this value. The **`glmnet()`** function can be used to fit the model, specifying the optimal lambda as the value of the **`lambda`** parameter. Here's how you could do this for each of the Ridge, Lasso, and Elastic Net models:

```r
# Fit the final Ridge model
final_ridge <- glmnet(x, y, alpha = 0, lambda = cv_ridge$lambda.min)

# Fit the final Lasso model
final_lasso <- glmnet(x, y, alpha = 1, lambda = cv_lasso$lambda.min)

# Fit the final Elastic Net model
final_elastic_net <- glmnet(x, y, alpha = 0.5, lambda = cv_elastic_net$lambda.min)
```

These models are now ready to make predictions on new data. You can use the **`predict()`** function to do this:

```r
new_data <- dfm_match(df_test, features = featnames(df_training))

# Predict with the final Ridge model
predictions_ridge <- predict(final_ridge, newx = new_data)

# Predict with the final Lasso model
predictions_lasso <- predict(final_lasso, newx = new_data)

# Predict with the final Elastic Net model
predictions_elastic_net <- predict(final_elastic_net, newx = new_data)
```

Once you have the predictions from your model, there are a number of things you can do with them, depending on your specific use case:

1. **Model Evaluation:** If you have the actual values for the outcome variable (y) for your `new_data`, you can compare these actual values with the predicted values to evaluate the performance of your model. Common metrics for regression models include Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared.
2. **Interpretation:** If your model is interpretable (as is often the case with Lasso and Ridge regression), you can examine the coefficients to understand the relationship between the predictors and the outcome variable. In the context of text analysis, this might mean understanding which words or phrases are most strongly associated with the outcome variable.
3. **Making decisions or taking actions:** The ultimate goal of most predictive modeling is to inform decision-making. What this looks like will depend on your specific context. For example, if you're using text analysis to predict customer sentiment, you might use the predictions to prioritize customer service resources. If you're predicting future sales based on product descriptions, you might use the predictions to inform inventory management decisions.

```r
library(caret)

# Calculate R-squared for the Ridge model
r_squared_ridge <- postResample(pred = predictions_ridge, obs = hotel_test$ratings)[2]
r_squared_ridge
```

```
## Rsquared
## 0.245648
```

```r
# Calculate RMSE for the Ridge model
rmse_ridge <- postResample(pred = predictions_ridge, obs = hotel_test$ratings)[1]
rmse_ridge
```

```
##     RMSE
## 1.210507
```

```r
# Calculate R-squared for the Lasso model
r_squared_lasso <- postResample(pred = predictions_lasso, obs = hotel_test$ratings)[2]
r_squared_lasso
```

```
##  Rsquared
## 0.2804121
```

```r
# Calculate RMSE for the Lasso model
rmse_lasso <- postResample(pred = predictions_lasso, obs = hotel_test$ratings)[1]
rmse_lasso
```

```
##     RMSE
## 1.182162
```

```r
# Calculate R-squared for the Elastic Net model
r_squared_elastic_net <- postResample(pred = predictions_elastic_net, obs = hotel_test$ratings)[2]
r_squared_elastic_net
```

```
##  Rsquared
## 0.2819712
```

```r
# Calculate RMSE for the Elastic Net model
rmse_elastic_net <- postResample(pred = predictions_elastic_net, obs = hotel_test$ratings)[1]
rmse_elastic_net
```

```
##     RMSE
## 1.181728
```

The R-squared value, also known as the coefficient of determination, is a statistical measure that represents the proportion of the variance for a dependent variable that's explained by an independent variable or variables in a regression model. If the R-squared value is 0.70, for example, then 70% of the variance in the dependent/output variable is explained by the independent/input variables. So, in general, the higher the R-squared, the better the model fits your data.

Root Mean Squared Error (RMSE) is a commonly used metric to evaluate the performance of a regression model. It provides a measure of the difference between the predicted values from the model and the actual values. Specifically, it is the square root of the average of squared differences between the predicted and actual values.

Interpreting the RMSE depends on the context and the scale of your outcome variable:

1. **Scale-dependent:** RMSE is in the same units as the outcome variable, which makes it somewhat interpretable. For example, if you are predicting house prices and your RMSE is 50,000, this means that on average, your predictions are $50,000 away from the actual prices.

2. **Lower is better:** A lower RMSE indicates a better fit to the data, because it means that the model's predictions are closer on average to the actual values.

3. **Zero is perfect:** An RMSE of zero means that the model's predictions are exactly correct all the time, which is unlikely in most real-world situations.

4. **Comparison between models:** RMSE can be useful for comparing different models on the same dataset. A model with a lower RMSE is performing better than a model with a higher RMSE, assuming that they are predicting the same outcome variable.

5. **No upper limit:** RMSE does not have an upper limit, and can be any positive number. The maximum value depends on the variability of the data set. If the data set has a large number of very large differences, the RMSE will be large.