# Block 1 Lab 2 report

Group B13: Aron(aroen488), Sergey(servo519), Shahin(mdpa888)

2025-11-26

### Statement of Contribution

Assignment 1 was mainly contributed by Aron. Assignment 2 was mainly contributed by Sergey. Assignment 3 was mainly contributed by Shahin. Theory questions: Q1: Aron, Q2: Sergey, Q3: Shahin.

### Assignment 1.

## Q1

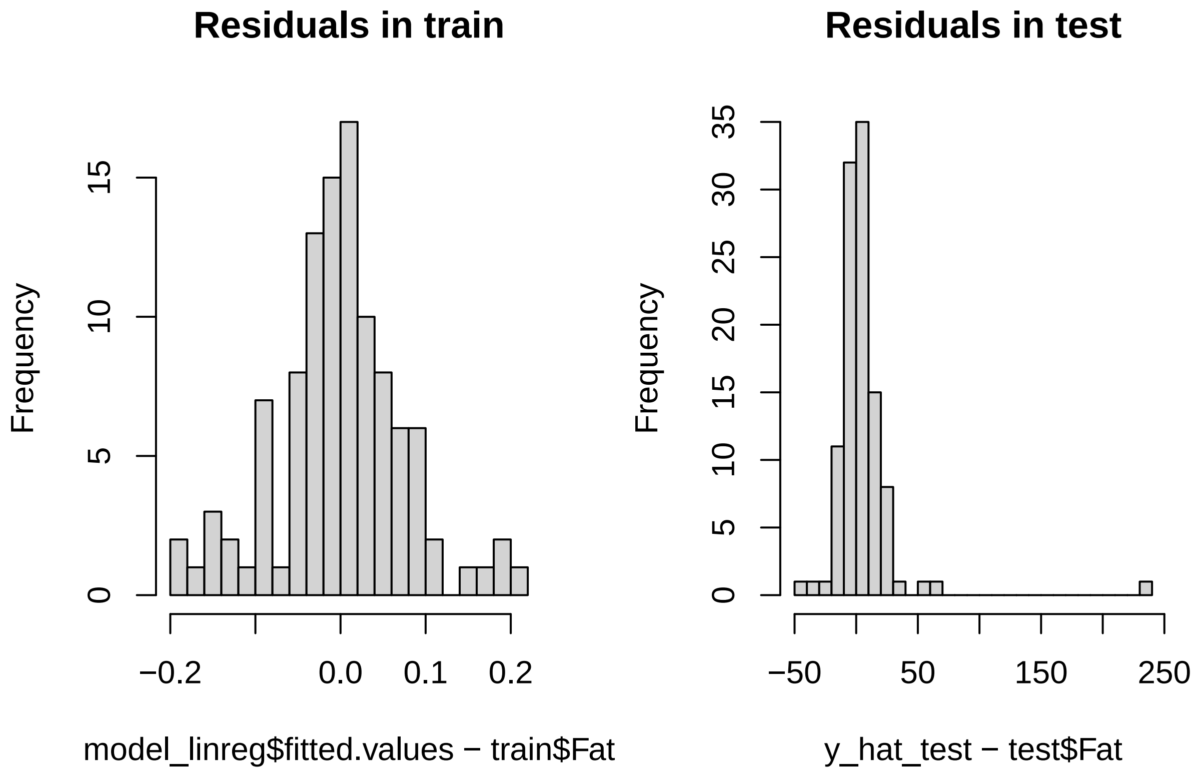
The statistical assumption is that we assume normal residuals:

$$\varepsilon = y - \hat{y} = y - X\beta$$

$$\varepsilon \sim N(0, \sigma^2)$$

then we simply perform MLE on f(y-XB) where f is the pdf for the normal distribution.

```
##
## Call:
## lm(formula = Fat ~ ., data = train[, index])
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.201500 -0.041315 -0.001041  0.037636  0.187860
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.815e+01  5.488e+00  -3.306  0.01628 *
## Channel1     2.653e+04  1.126e+04   2.357  0.05649 .
## Channel2    -5.871e+04  3.493e+04  -1.681  0.14385
## Channel3     1.154e+05  7.373e+04   1.565  0.16852
## Channel4    -2.432e+05  1.175e+05  -2.070  0.08387 .
## Channel5     3.026e+05  1.193e+05   2.536  0.04430 *
## Channel6    -2.365e+05  8.160e+04  -2.898  0.02741 *
## Channel7     1.090e+05  3.169e+04   3.440  0.01380 *
## Channel8    -6.054e+04  1.508e+04  -4.015  0.00700 **
## Channel9     7.871e+04  2.160e+04   3.643  0.01079 *
## Channel10   -1.730e+04  1.640e+04  -1.055  0.33215
## Channel11    9.562e+04  3.529e+04   2.710  0.03512 *
## Channel12   -2.114e+05  6.198e+04  -3.410  0.01431 *
## Channel13    9.725e+04  4.424e+04   2.198  0.07026 .
## Channel14    5.296e+04  4.666e+04   1.135  0.29968
## Channel15   -7.855e+04  5.245e+04  -1.498  0.18491
## Channel16   -8.209e+03  1.893e+04  -0.434  0.67969
## Channel17    3.769e+04  1.987e+04   1.897  0.10666
## Channel18    3.306e+04  7.934e+03   4.167  0.00590 **
## Channel19   -8.405e+04  1.929e+04  -4.358  0.00478 **
## Channel20    1.510e+05  3.361e+04   4.492  0.00414 **
## Channel21   -2.069e+05  4.256e+04  -4.862  0.00282 **
## Channel22    1.348e+05  3.824e+04   3.526  0.01243 *
## Channel23   -4.094e+04  3.546e+04  -1.154  0.29222
## Channel24    2.023e+04  2.761e+04   0.733  0.49134
## Channel25    3.269e+03  1.071e+04   0.305  0.77045
## Channel26   -1.297e+04  7.636e+03  -1.699  0.14028
## Channel27    4.131e+03  1.422e+04   0.291  0.78120
## Channel28   -4.548e+03  2.988e+04  -0.152  0.88402
## Channel29    1.089e+04  1.768e+04   0.616  0.56072
## Channel30   -7.985e+04  2.653e+04  -3.010  0.02371 *
## Channel31    1.756e+05  5.279e+04   3.326  0.01589 *
## Channel32   -1.107e+05  2.904e+04  -3.813  0.00883 **
## Channel33   -6.525e+04  5.407e+04  -1.207  0.27294
## Channel34    1.007e+05  6.589e+04   1.528  0.17738
## Channel35   -2.841e+03  1.214e+04  -0.234  0.82266
## Channel36   -2.268e+04  2.295e+04  -0.988  0.36127
## Channel37   -4.479e+04  1.292e+04  -3.468  0.01334 *
## Channel38    3.209e+04  1.843e+04   1.742  0.13221
## Channel39    1.992e+04  2.067e+04   0.964  0.37246
## Channel40   -9.833e+03  2.431e+04  -0.404  0.69988
## Channel41    1.659e+04  3.648e+04   0.455  0.66531
## Channel42   -1.829e+04  3.528e+04  -0.519  0.62260
## Channel43   -2.423e+04  2.427e+04  -0.998  0.35669
## Channel44    3.246e+04  2.013e+04   1.613  0.15793
## Channel45   -8.089e+03  4.023e+04  -0.201  0.84728
## Channel46    7.065e+03  2.810e+04   0.251  0.80990
## Channel47   -4.062e+04  1.007e+04  -4.034  0.00685 **
## Channel48    9.080e+04  2.618e+04   3.469  0.01332 *
## Channel49   -6.647e+04  2.372e+04  -2.803  0.03105 *
## Channel50   -4.196e+04  2.856e+04  -1.469  0.19213
## Channel51    1.097e+05  5.572e+04   1.968  0.09661 .
## Channel52   -1.148e+05  6.376e+04  -1.800  0.12196
## Channel53    9.525e+04  7.450e+04   1.278  0.24830
## Channel54   -4.534e+04  7.363e+04  -0.616  0.56067
## Channel55   -1.535e+03  4.933e+04  -0.031  0.97618
## Channel56   -2.377e+03  2.109e+04  -0.113  0.91394
## Channel57    3.174e+04  1.005e+04   3.158  0.01961 *
## Channel58    2.221e+03  1.048e+04   0.212  0.83915
## Channel59   -8.504e+04  2.574e+04  -3.304  0.01634 *
## Channel60    6.382e+04  1.607e+04   3.972  0.00735 **
## Channel61    2.151e+04  1.234e+04   1.742  0.13211
## Channel62   -2.859e+04  1.065e+04  -2.685  0.03631 *
## Channel63    1.796e+04  9.187e+03   1.955  0.09838 .
## Channel64    5.759e+04  3.526e+04   1.633  0.15354
## Channel65   -1.470e+05  6.911e+04  -2.127  0.07752 .
## Channel66    9.121e+04  4.461e+04   2.045  0.08688 .
## Channel67   -5.733e+03  2.197e+04  -0.261  0.80288
## Channel68   -6.290e+04  2.192e+04  -2.870  0.02843 *
## Channel69    6.421e+04  2.074e+04   3.096  0.02121 *
## Channel70   -1.749e+04  1.581e+04  -1.106  0.31111
## Channel71   -7.248e+03  1.934e+04  -0.375  0.72075
## Channel72    3.406e+04  1.185e+04   2.873  0.02830 *
## Channel73   -2.100e+04  1.132e+04  -1.855  0.11308
## Channel74   -3.314e+04  1.220e+04  -2.717  0.03480 *
## Channel75    7.039e+04  2.054e+04   3.427  0.01402 *
## Channel76   -3.187e+04  1.736e+04  -1.836  0.11597
## Channel77    2.061e+04  1.810e+04   1.138  0.29832
## Channel78   -1.180e+04  2.273e+04  -0.519  0.62225
## Channel79    2.669e+04  2.997e+04   0.890  0.40750
## Channel80   -6.051e+04  1.483e+04  -4.080  0.00650 **
## Channel81    1.386e+03  2.628e+04   0.053  0.95966
## Channel82    1.020e+05  4.694e+04   2.173  0.07275 .
## Channel83   -1.706e+05  4.688e+04  -3.640  0.01083 *
## Channel84    1.097e+05  2.892e+04   3.792  0.00905 **
## Channel85   -1.294e+05  3.600e+04  -3.594  0.01145 *
## Channel86    2.130e+05  4.345e+04   4.903  0.00270 **
## Channel87   -1.198e+05  3.818e+04  -3.139  0.02011 *
## Channel88   -2.199e+04  6.085e+04  -0.361  0.73021
## Channel89    7.974e+04  5.077e+04   1.571  0.16733
## Channel90   -1.711e+05  5.499e+04  -3.112  0.02079 *
## Channel91    2.107e+05  6.406e+04   3.289  0.01663 *
## Channel92   -1.959e+05  7.171e+04  -2.733  0.03407 *
## Channel93    2.874e+05  9.937e+04   2.892  0.02762 *
## Channel94   -3.064e+05  9.601e+04  -3.191  0.01881 *
## Channel95    2.048e+05  6.220e+04   3.292  0.01656 *
## Channel96   -5.600e+04  2.929e+04  -1.912  0.10441
## Channel97   -1.318e+04  3.050e+04  -0.432  0.68065
## Channel98   -2.724e+04  2.107e+04  -1.292  0.24375
## Channel99    3.556e+04  1.382e+04   2.573  0.04218 *
## Channel100  -1.206e+04  4.264e+03  -2.828  0.03006 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3191 on 6 degrees of freedom
## Multiple R-squared:      1,  Adjusted R-squared:  0.9994
## F-statistic:  1651 on 100 and 6 DF,  p-value: 1.058e-09

## [1] "Train MSE"

## [1] 0.005709117

## [1] "Test MSE"

## [1] 722.4294
```



We see that the MSE is very low in the training set, model is extremely overfit as the MSE spikes in the validation set. In the validation set the model makes alot of very wrong predictions, considering that the median for Fat is 15.9.

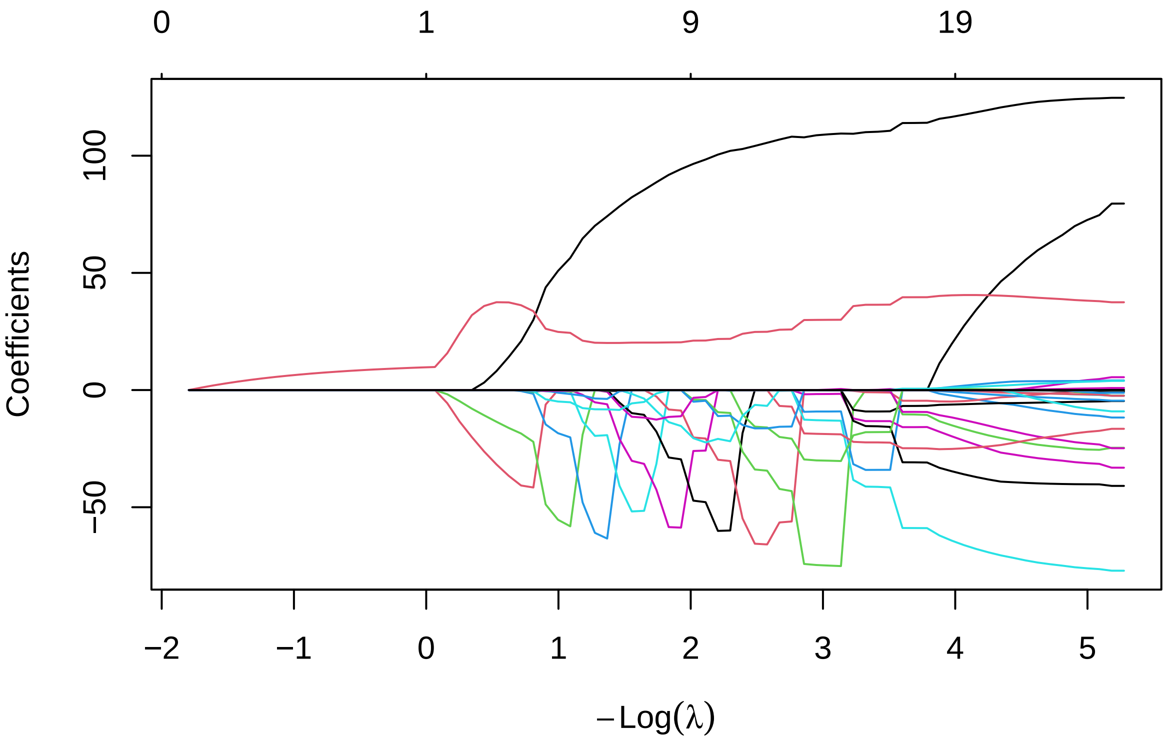
## Q2

For lasso regression we force the less important parameters to 0, this is because we add the L1 norm to the cost function.

$$Cost = \sum_{i=1}^{n}(\vec{y_i} - x_i^T\vec{\beta}) + \lambda\sum_{j=1}^{p}|\beta_j|$$

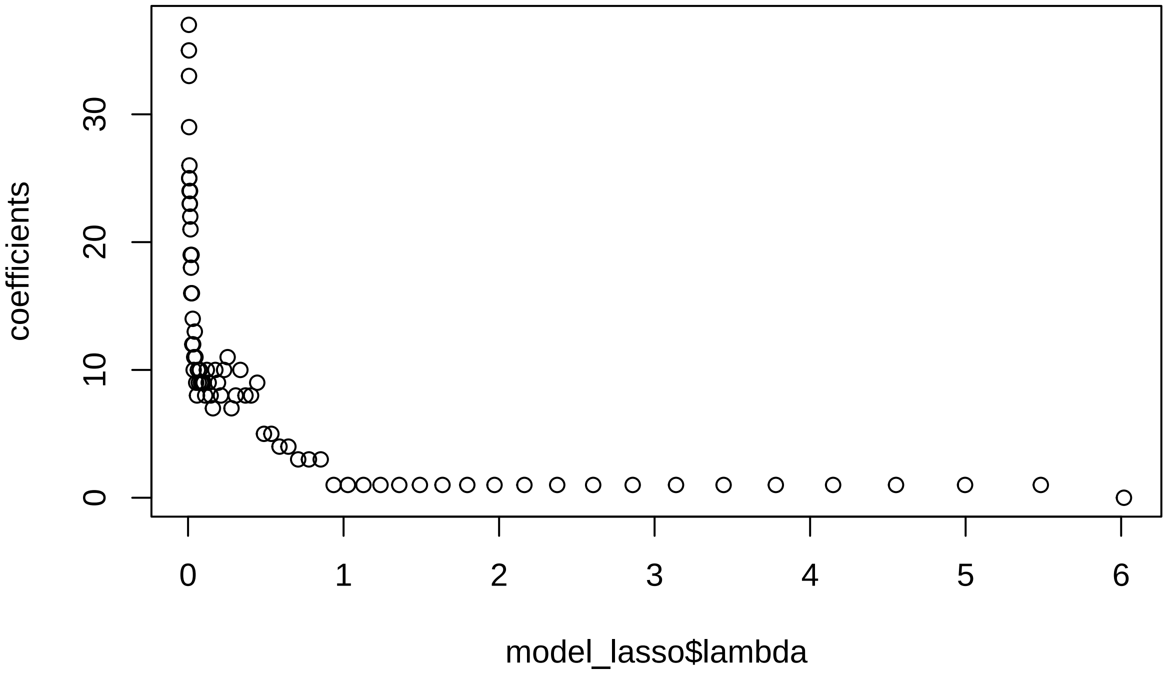
Where $x_i^T$ is the covariates for observation i. $\lambda$ is a hyperparameter controlling the strength of the lasso regularization.

## Q3



```
## [1] "Channels are almost 1 to 1 correlated"

##          Channel1 Channel2 Channel3 Channel4 Channel5
## Channel1 1.000000 0.999992 0.999969 0.999933 0.999885
## Channel2 0.999992 1.000000 0.999993 0.999971 0.999938
## Channel3 0.999969 0.999993 1.000000 0.999993 0.999974
## Channel4 0.999933 0.999971 0.999993 1.000000 0.999994
## Channel5 0.999885 0.999938 0.999974 0.999994 1.000000
```

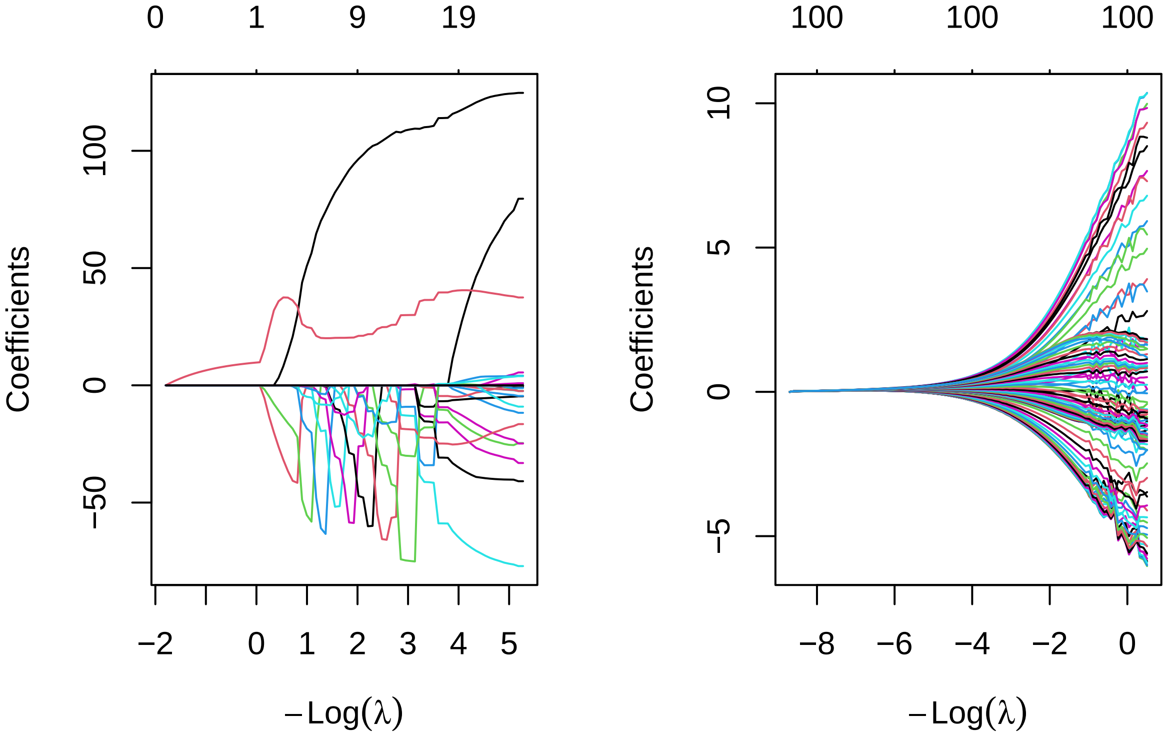
$\lambda = e^{-5} \leftrightarrow -log(\lambda) = 5$, for small $\lambda$ the regularization is weak and the coefficients are large, for larger $\lambda$ we see that all coefficients approach 0. An interesting pattern is that coefficients spike back up from 0. This is most probably because the correlation is so high. This means the model can't know if the signal is comming from Channel1 or Channel2 or a mix of them, if one coefficient drops the other one rises as the MSE drops from the signal being in the model.



```
## [1] "Smallest lambda tested that results in 3 or fewer nonzero coefficients:"
```

```
## [1] 0.7082131
```

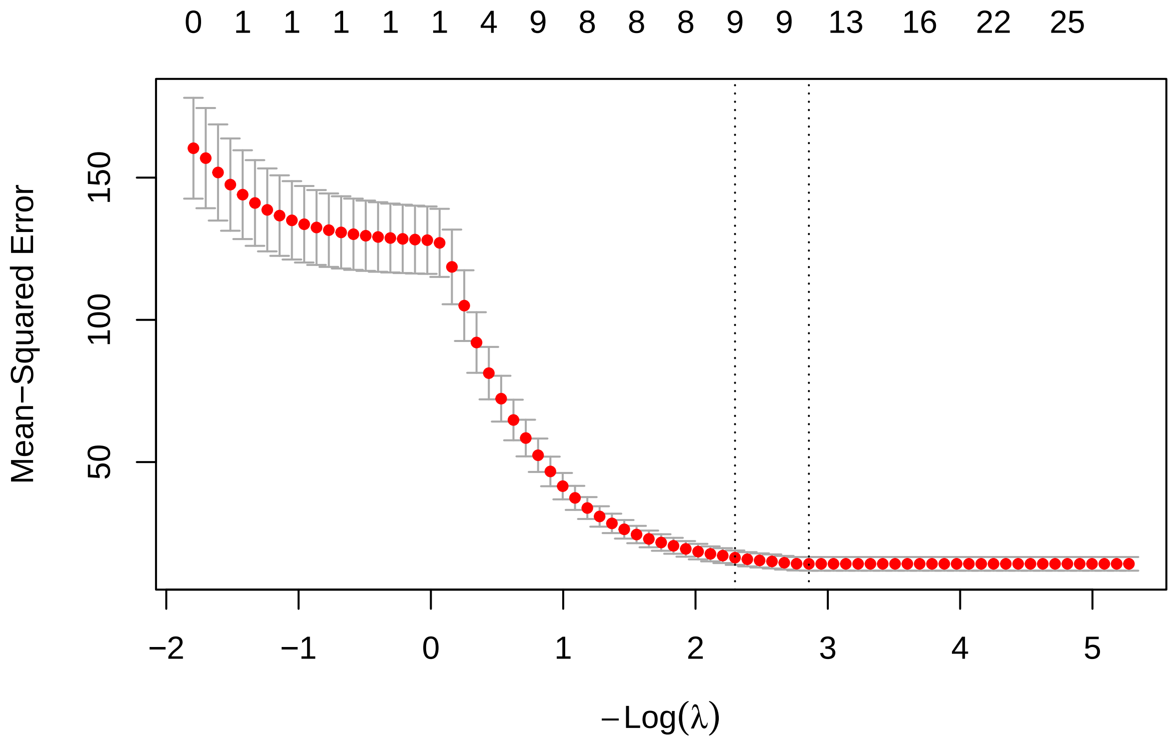
## Q4



Ridge shrinks all parameters at the same time while ridge tends to drop them to zero. This is because a reduction in a small coefficient, doesn't reduce the cost anything for L2 loss. For lasso, a reduction in a small coefficient is the same as reducing a large coefficient. $(10-\varepsilon)^2 >> (0.1-\varepsilon)^2$ while $(10-\varepsilon) = (0.1-\varepsilon)$.

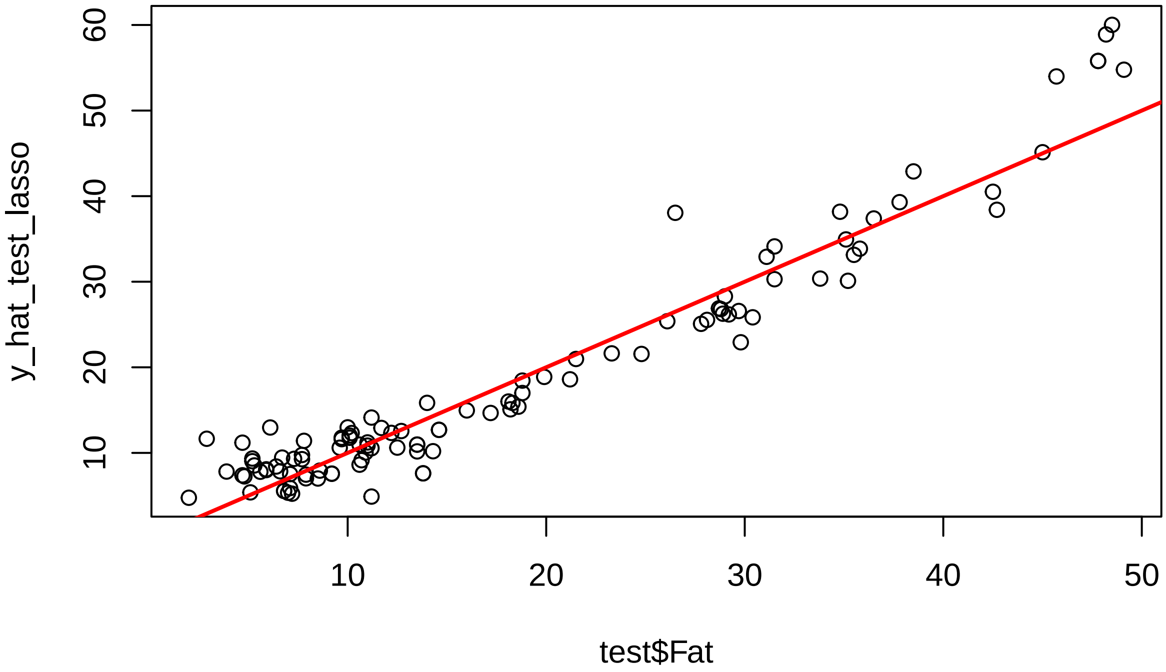
It doesn't make sense to select features with ridge, they do not drop to 0, on the top of the right plot we see the number of coefficients stays at 100 (this is not 101 because the intercept is 0, the intercept is included in the parametrization of the model).

## Q5



```
## [1] "Best lambda according to lowest mean CV-loss:"

## [1] 0.05744535

## [1] "log(lambda_optimal)"

## [1] -2.856921

## [1] "Amount of nonzero coefficients:"

## s50
##   8
```

The CIs for -2.8 and -4 (2.8 and 4 in the plot) are overlapping so there is no statistically significant improvement.



```
## [1] "Test MSE"
```

```
## [1] 13.2998
```

The model seems much better than the original non-regularized model. The predictions look roughly evenly distributed around the x=y line, no systematic error. The test MSE is also way lower than before.

## Assignment 2.

### Task 1

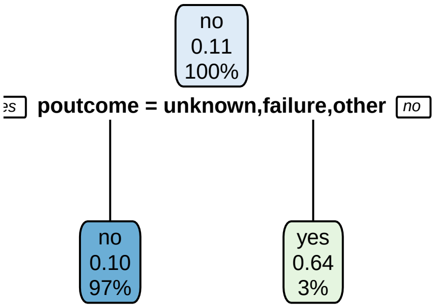
After splitting the data into training/test/validation at the 40/30/30 ratio, we have 18084/13563/13564 rows in each respective dataset.

### Task 2

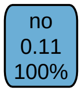
```
##            Model Train_Misclass Valid_Misclass
## 1        Default     0.10484406      0.1092679
## 2 Minbucket 7000     0.11424464      0.1207697
## 3      CP 0.0005     0.07951781      0.1262995
```

```
##       Default Minbucket7000     CP_0.0005
##             1             0           229
```

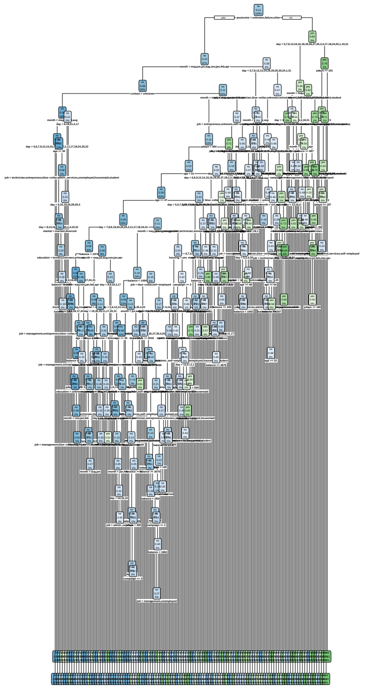
Default Settings



Minbucket = 7000

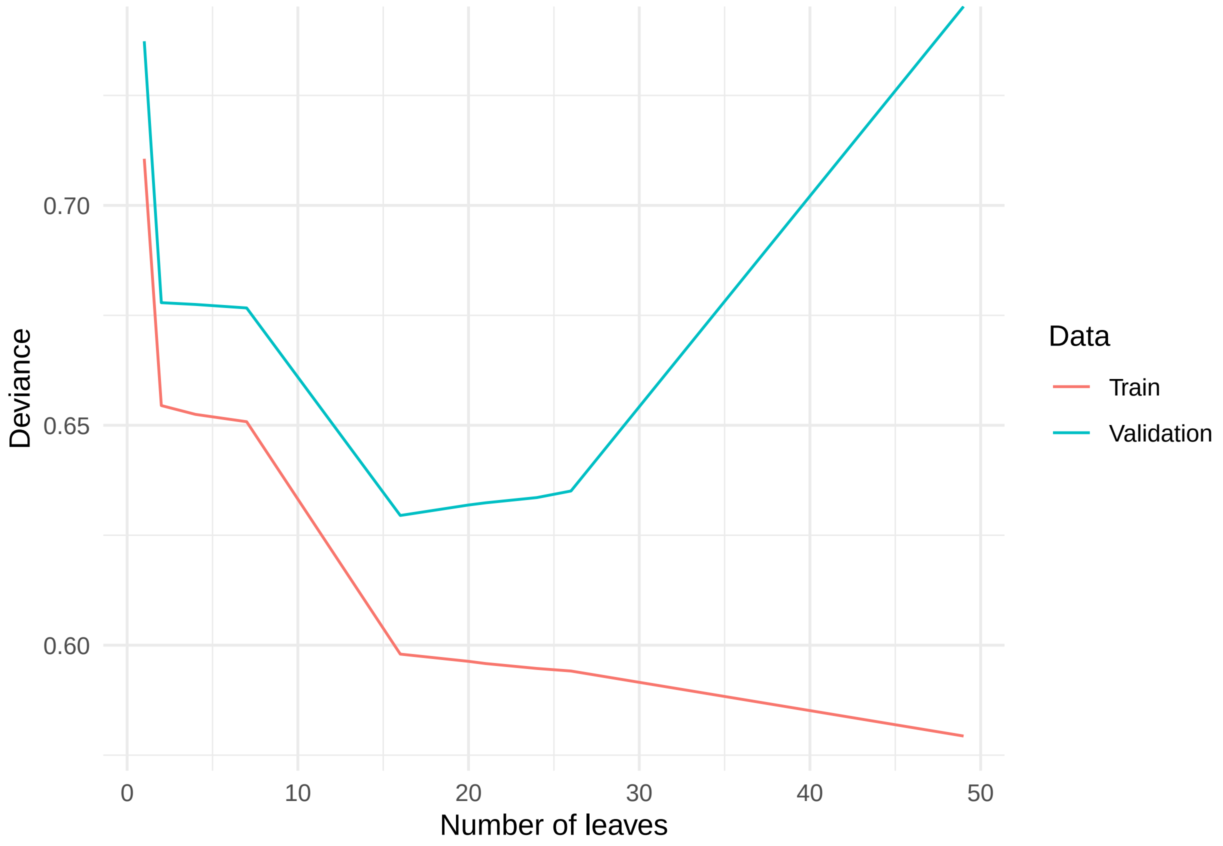


cp = 0.0005



- Default: uses reasonable default parameters, only 1 split, relatively low bias and low variance
- Min bucket 7000: no split at all as child nodes are all smaller than 7000. This is just assigning all instances to the dominant class
- Min dev = 0.0005: Very many leaves, low bias but the highest variance, clear signs of overfitting. We need to prune this tree.

**Task 3**



```
## [1] "Optimal number of leaves: 16"
```

For a classification model that predicts class probabilities $\hat{p}_{i,k}$ for each observation $i = 1, \dots, n$ and class $k = 1, \dots, K$, the (mean) deviance is

$$D = -\frac{2}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} y_{i,k} \log\big(\hat{p}_{i,k}\big),$$

where

- $n$ is the number of observations in the dataset.
- $K$ is the number of classes.
- $y_{i,k}$ is an indicator variable: $y_{i,k} = 1$ if observation $i$ belongs to class $k$, and 0 otherwise.
- $\hat{p}_{i,k}$ is the predicted probability (from the model) that observation $i$ belongs to class $k$.
- $D$ is the (average) deviance: lower values of $D$ indicate a better fit.

In the special case of binary classification with response $y_i \in \{0, 1\}$ and predicted probability $\hat{p}_i = P(y_i = 1)$, this simplifies to

$$D = -\frac{2}{n} \sum_{i=1}^{n} \Big[ y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i) \Big].$$

Here

- $y_i$ is the observed class for observation $i$ (1 for “success”, 0 for “failure”),

- $\hat{p}_i$ is the predicted probability of class 1 for observation $i$.

As the number of leaves increases, the training deviance decreases monotonically, while the validation deviance initially decreases and then increases, forming a U-shaped curve. For very small trees (few leaves), both training and validation deviances are high, indicating underfitting and high bias. As we allow more leaves, the model becomes more flexible, bias decreases, and validation deviance improves, reaching its minimum at 16 leaves. Beyond 16 leaves, further increases in tree size continue to reduce training deviance but lead to higher validation deviance, which indicates that additional complexity mainly increases variance and causes overfitting. Thus a tree with 16 leaves provides the best bias–variance tradeoff.

```
##     poutcome        month          day        pdays      contact          job 
## 350.05837209 171.48544945  41.08651139  36.43811307  35.05629601  10.43249346 
##      balance     campaign          age     previous    education      default 
##   5.42365245   3.44750580   1.97530146   0.38556166   0.36602273   0.06965385 
```

Variable importance in rpart is based on how much each variable reduces the node loss (impurity / deviance) over all the splits where it is used.

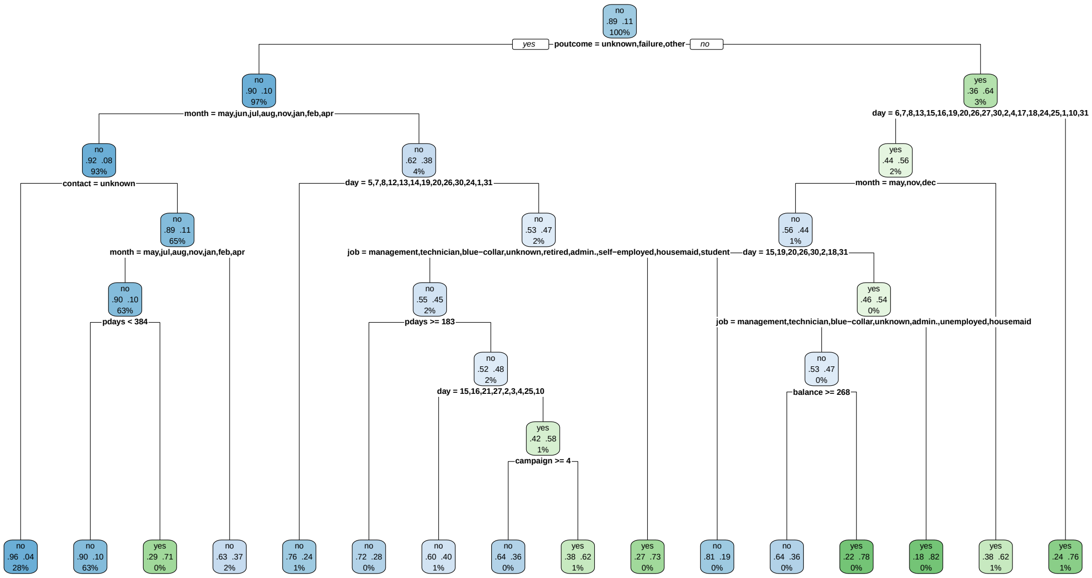
- Most important: poutcome (outcome of previous marketing campaign) and month (month of contact).
- Next tier: day of the month, pdays (days since last contact), and contact type.
- Less important but still used: job, balance, campaign.
- Barely used: age, previous, education, default (almost no contribution).

The tree mainly bases its decisions on past campaign outcome and timing of the contact, with customer/job characteristics playing a smaller role.

Table 1: Train and validation misclassification rates for tree models

| Model | Train_Misclass | Valid_Misclass |
|---|---:|---:|
| Default | 0.105 | 0.109 |
| Minbucket 7000 | 0.114 | 0.121 |
| CP 0.0005 | 0.080 | 0.126 |
| Optimal 16-leaf | 0.099 | 0.109 |

The 16-leaf tree chosen by minimum validation deviance has essentially the same validation misclassification rate as the default tree that only splits on *poutcome*. However, the 16-leaf tree achieves a lower validation deviance, indicating that its predicted probabilities are better calibrated and capture more nuanced structure in the data. In terms of the bias–variance tradeoff, it reduces bias (better fit) without noticeably reducing 0–1 error on the validation set.



Key findings:

- Previous campaign success is the single most informative variable, essentially splitting the population into a low-propensity group and a high-propensity group.
- Timing matters: both within the non-success and success groups, the specific month and day of contact have substantial influence, suggesting seasonality or within-month effects in customer responsiveness.
- Customer profile variables such as job and balance play a secondary, fine-tuning role: they help pick out especially promising or unpromising subsegments but do not overturn the primary pattern driven by past outcome and timing.

**Task 4**

Table 2: Confusion matrix for tree_opt on test data (TP = true positives, FP = false positives, FN = false negatives, TN = true negatives)

| | Predicted yes | Predicted no |
|---|---|---|
| Observed yes | 328 (TP) | 1257 (FN) |
| Observed no | 207 (FP) | 11772 (TN) |

```
## [1] "Accuracy: 0.892067236803303"
## [1] "Precision: 0.613084112149533"
## [1] "Recall: 0.206940063091483"
## [1] "F1: 0.309433962264151"
```

From the numbers above we can see that the accuracy is about 0.89, but the majority class ("no") already accounts for about 90% of the data, so a trivial classifier that always predicts "no" would get almost the same accuracy. At the same time, precision for "yes" is 0.61 and recall is 0.21, giving F1 around 0.31. That means that the model:

- Misses most actual "yes" cases (low recall).
- Among predicted "yes", a substantial fraction are wrong (precision moderate, not high).

So in terms of identifying the minority class, the predictive power is modest: the model finds only about one fifth of the true positives while still making a fair number of false alarms.

The F1 appears to be more informative here as the class distribution is highly skewed toward "no", so accuracy is dominated by correct "no" predictions and hides the poor detection of "yes". F1 focuses on the minority/positive class by combining precision and recall, so it better reflects the model's usefulness for finding subscribers.

Therefore despite a high overall accuracy driven by the majority "no" class, the model's performance on the important "yes" class is relatively weak, and F1 (approx 0.31) is a more appropriate performance summary than accuracy given the strong class imbalance.

**Task 5**

```
##         Predicted
## Observed no yes
##      no   0   1
##      yes  5   0
```

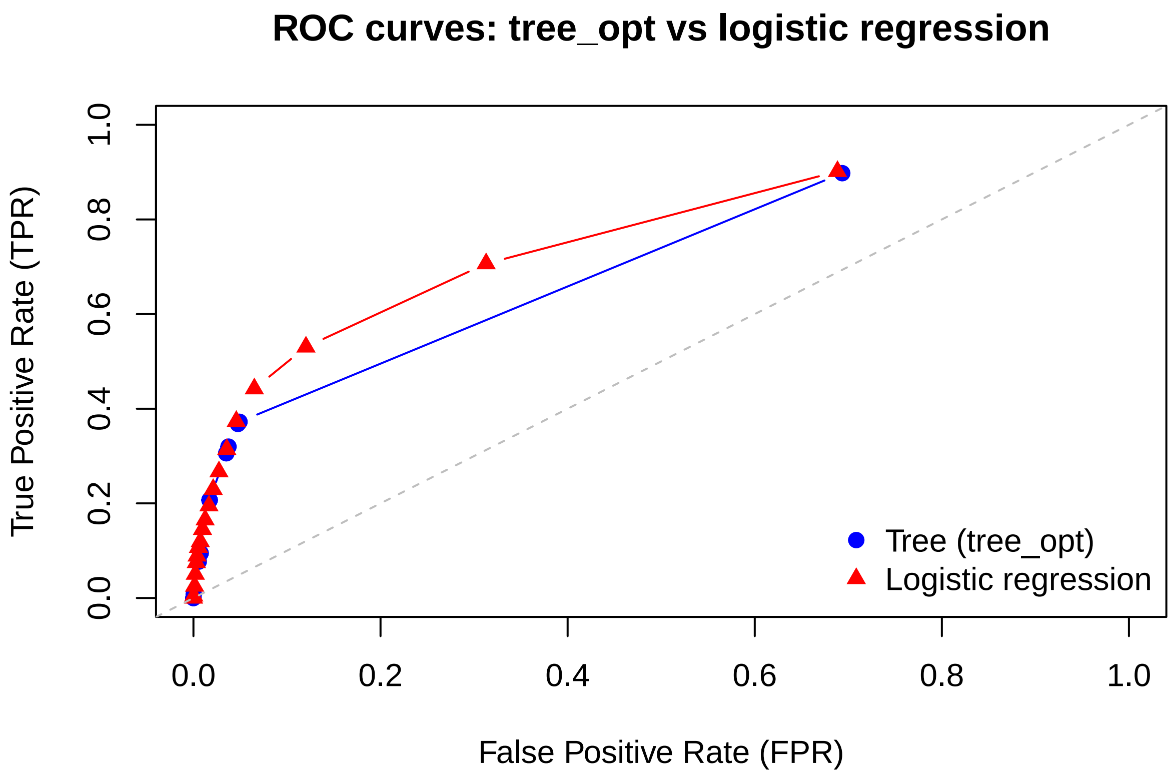
Table 3: Confusion matrix using custom loss matrix (rows = Observed, columns = Predicted)

| | Predicted yes | Predicted no |
|---|---|---|
| Observed yes | 749 (TP) | 836 (FN) |
| Observed no | 1069 (FP) | 10910 (TN) |

```
## [1] "Accuracy: 0.859554703627249"
## [1] "Precision: 0.411991199119912"
## [1] "Recall: 0.472555205047319"
## [1] "F1: 0.440199823684984"
```

As could be expected, when using a custom loss function that penalizes false negatives more harshly than false positives, we are seeing fewer false negatives than before. It results in a higher recall of the model and a higher F1 score at the expense of lower precision and accuracy.

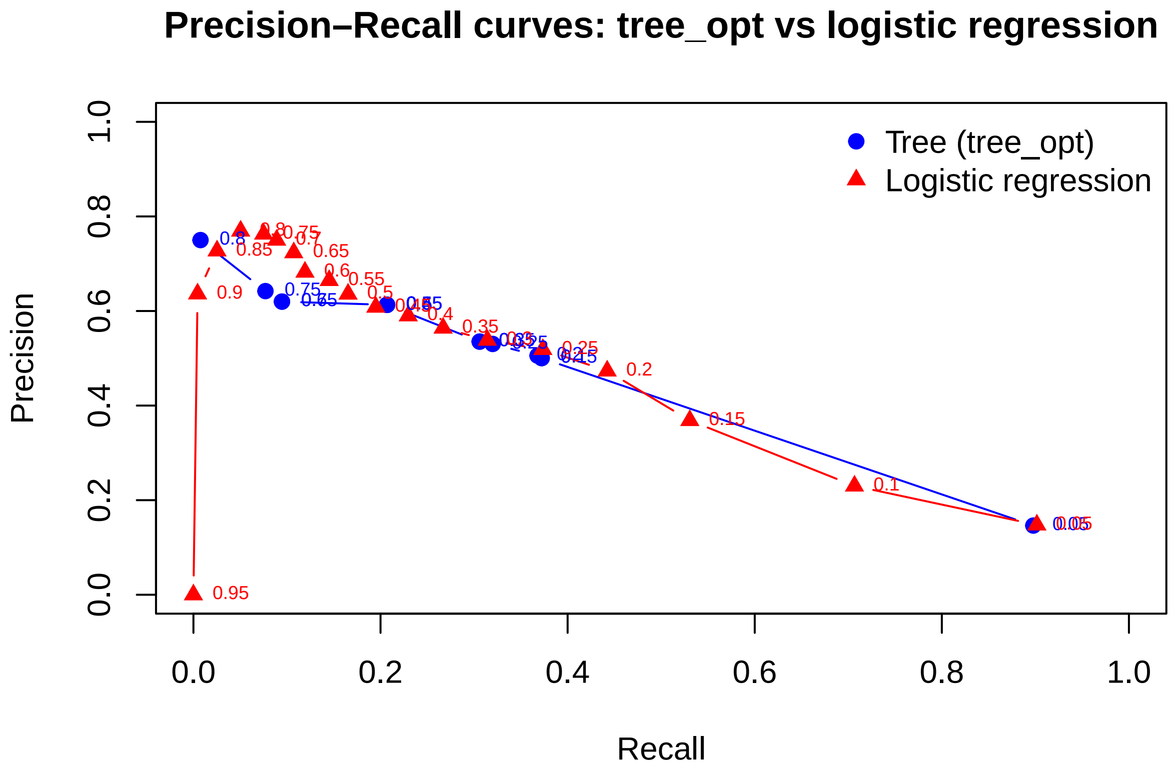
**Task 6**



The ROC plot shows that, while both models are clearly better than random (their curves lie well above the diagonal), the logistic regression curve (red) is consistently above or equal to the tree curve (blue), so for almost every operating point it achieves a higher TPR for the same FPR, i.e. it dominates the tree in ROC space. This suggests that, as probabilistic classifiers, the logistic regression model has better overall ability to separate “yes” from “no” than the optimal tree.

At the same time, it should be noted that ROC curves treat TPR and FPR symmetrically and do not depend on class prevalence, so a model can look good in ROC space even when the positive class is very rare and performance on that class is mediocre. By contrast, precision–recall (PR) curves focus only on the positive class with the recall being equal to the TPR and the precision showing the fraction of predicted “yes” that are truly “yes”.

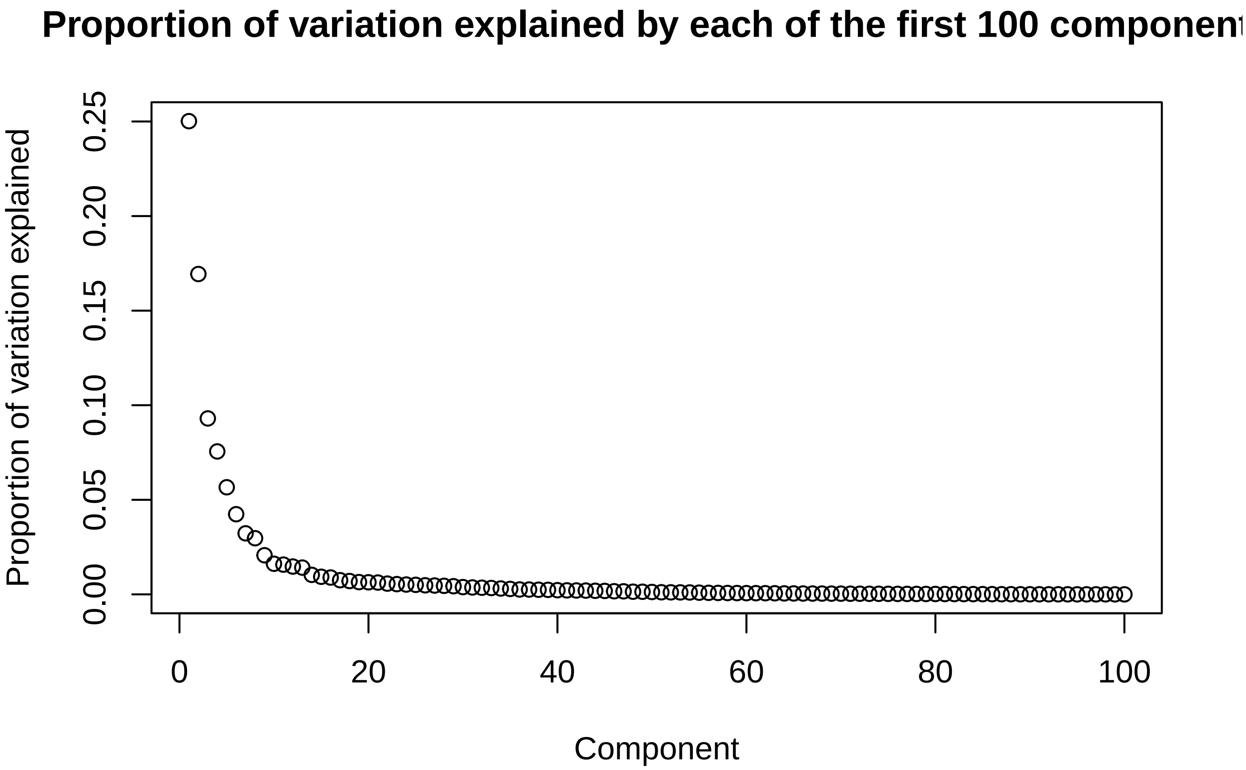
In our case, where we observe a significant imbalance between “yes” and “no” responses (with “nO” responses accounting for about 89% of the data), the precision-recall curve is indeed a better choice than relying on ROC alone as it will give a clearer view of how well each model retrieves the “yes” class at acceptable false-alert rates.



The precision-recall curve shown above demonstrates that by lowering the cutoff probability, we can achieve higher recall at the expense of lower precision and vice versa depending on the business goals.

## Assignment 3. Principal components and implicit regularization

### 3.1



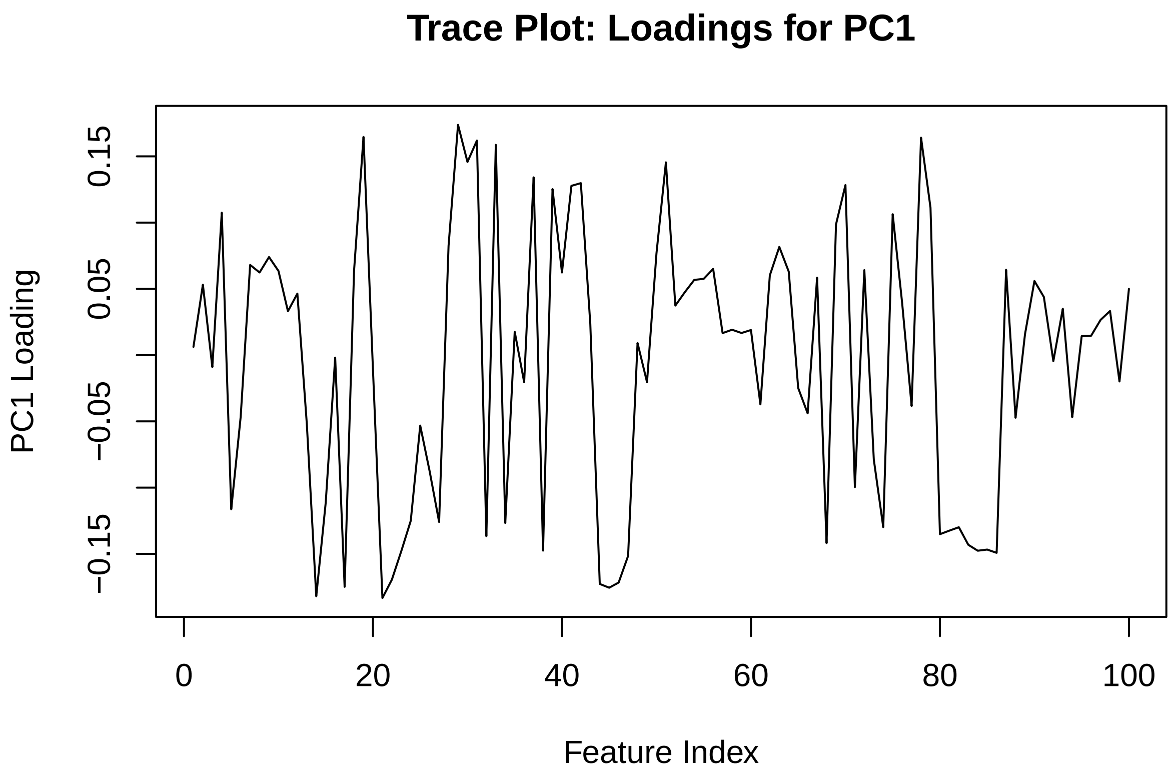
```
## Number of components needed for 95% variance: 35
```

```
## Proportion of variation explained by the first component: 0.2501699
```

```
## Proportion of variation explained by the second component: 0.1693597
```

According to the output, we need 35 components to obtain at least 95% of the variance in the data. The proportion of variation explained by the first and second principal components are 0.2501699 and 0.1693597 respectively.

### 3.2



From the plot, we found that PC1 is influenced by many features, but a small group of variables dominates the pattern.

The 5 most contributed features are:

```
## Top 5 PC1 contributors (absolute loadings):
```

```
##      medFamInc      medIncome    PctKids2Par     pctWInvInc PctPopUnderPov
##      0.1833080      0.1819830      0.1755423      0.1748683      0.1737978
```

medFamInc(median family income): median family income has a relationship with crime example, low-income parents with kids

medIncome(median household income): median household income is almost the same as medFamInc

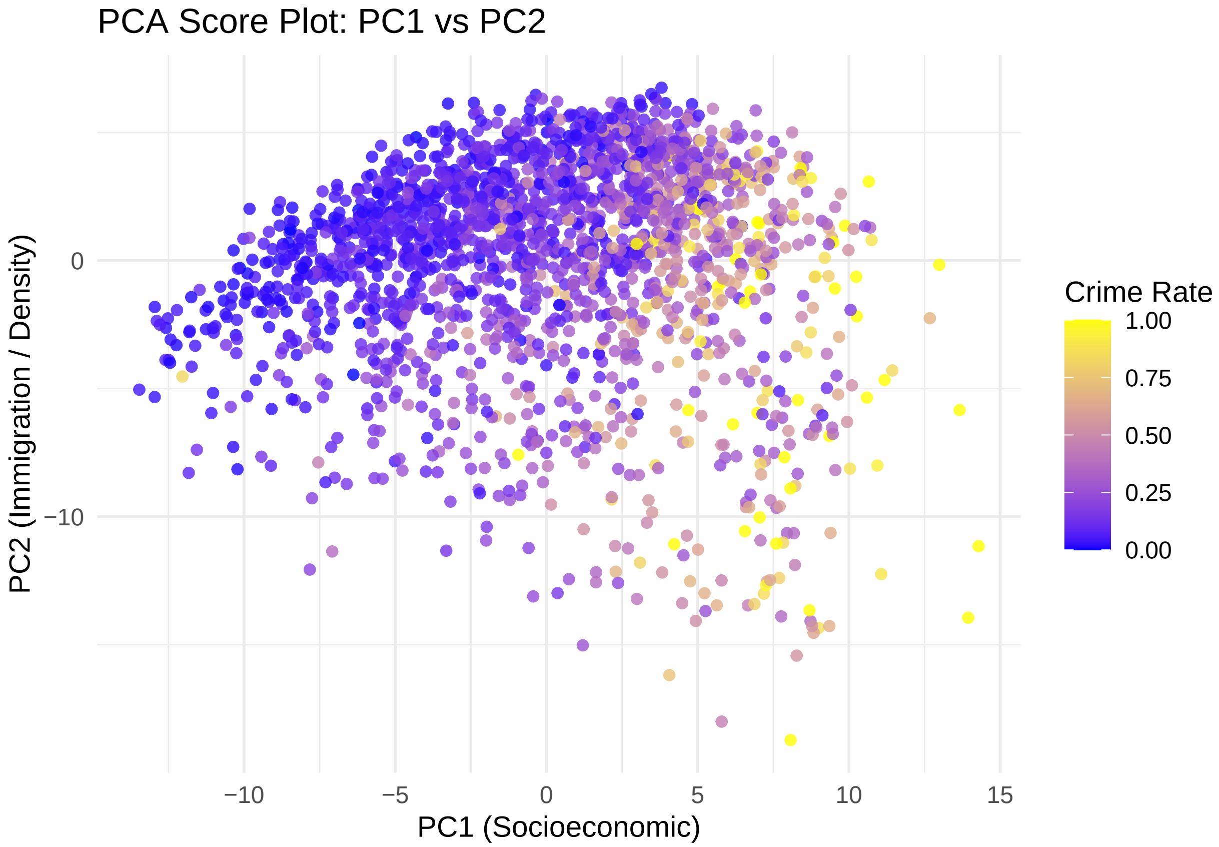
PctKids2Par(percentage of kids in family housing with two parents): family stability - kids with two parents are more likely to be well-educated and have a good life, so they are less likely to commit crimes.

pctWInvInc(percentage of households with investment / rent income in 1989): people with investment/rent income are more likely to be rich, and they are less likely to commit crimes.

PctPopUnderPov(percentage of people under the poverty level): people under the poverty level are more likely to commit crimes because they need money to survive.

Lower income, more poverty, and weaker family stability are well-established predictors of higher violent crime. This suggests PC1 behaves like a socioeconomic disadvantage factor.

```
##          PC1        PC2 Crime
## 1  -1.332980  0.9046451  0.20
## 2   1.419428 -0.3977787  0.67
## 3   2.118982  2.3928566  0.43
## 4  -2.975519 -1.9292117  0.12
## 5  -5.551139  2.7784657  0.03
## 6  -5.698463 -6.6241005  0.14
## 7  -4.000753  3.5159498  0.03
## 8   8.001313 -6.6474368  0.55
## 9   3.474277  4.7616070  0.53
## 10 -5.687635 -1.4312897  0.15
```



The PC1–PC2 score plot shows a clear gradient in crime levels. High-crime communities lie in the region of high PC1 values, corresponding to poverty, low income, and low education. Low-crime communities cluster in the opposite direction, where socioeconomic conditions are better. PC2 adds additional variation related to immigration and population density, but plays a secondary role compared to PC1.

### 3.3

```
## Train mean squared error: 0.2752071
```

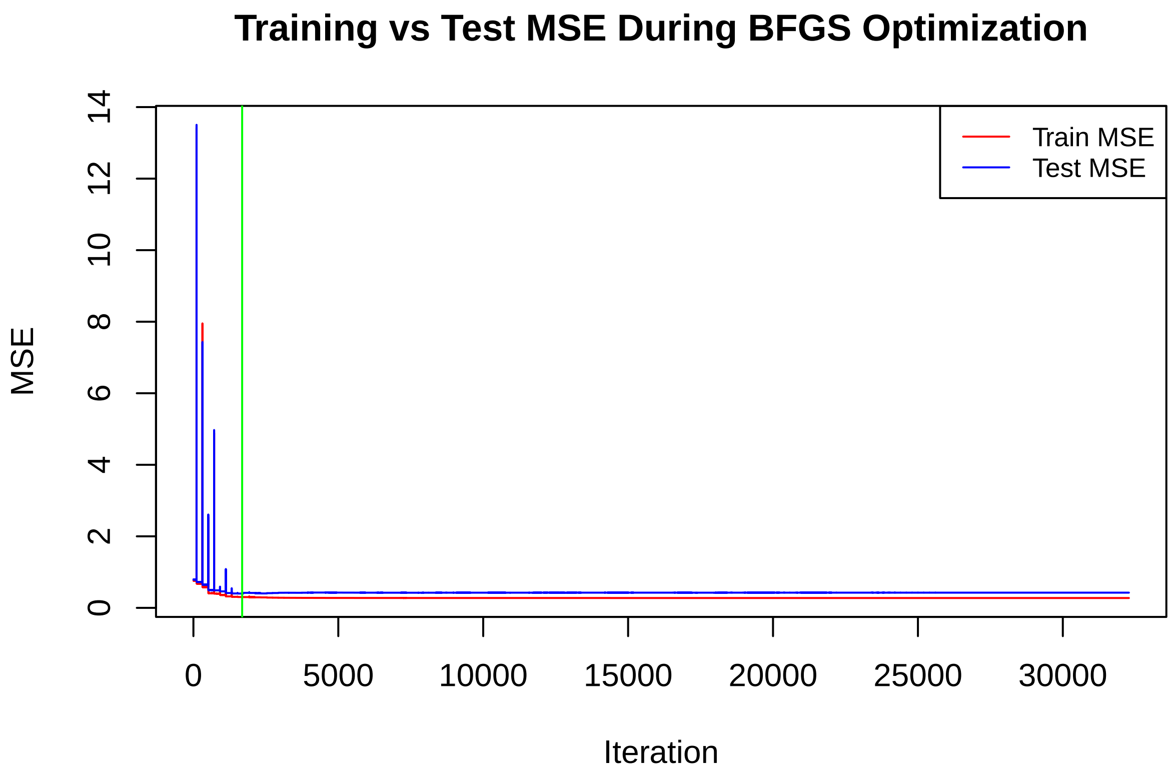
```
## Test mean squared error: 0.4248011
```

A test MSE of 0.425 is significantly larger than 0.275.

This indicates:

- Overfitting: The model captures patterns specific to the training sample, not general patterns.
- High dimensionality: The dataset has hundreds of features, and linear regression struggles with so many predictors.

### 3.4



```
## Iteration for early stopping: 2182
```

```
## Train mean squared error: 0.3032999
```

```
## Test mean squared error: 0.4002329
```

The plot shows how the training and test errors change during the BFGS optimization. At first, both errors are unstable, so we ignore the first 500 iterations. After that, the training error keeps decreasing, but the test error goes down and then starts to rise again. The best model is the one where the test error is smallest, which happens at iteration 2182. This is where we stop (early stopping) to avoid overfitting.

***Comparing Step 3 and Step 4***

In Step 3, the model fits the training data very well (MSE = 0.275), but the test error is noticeably higher (0.425), meaning it overfits. In Step 4, using early stopping, the training error is a little worse (0.303), but the test error improves to 0.400. This means early stopping prevents overfitting and gives a model that works better on new data.

## Assignment 4.

**Q1: What are the practical approaches for reducing the expected new data error, according to the book?** Page 79-80: $E_{New} = E_{Train} + generalisation gap$. If the error in the test data is lower than the train error a better model could be used. The generalisation gap drops as n increases, assuming no systematic error in the sampling. During cross validation, if $E_{Holdout} \approx Etrain$ we should try to decrease $E_{Train}$ by making the model more complex. If $E_{Train}$ is very low and $E_{Holdout}$ is high we probably need more regularization, overfitting. There is nuance, for neural nets there are often many hyperarameters that control the regularization, they do not all perform the same, some are better for ceratain architectures.

**Q2: What important aspect should be considered when selecting minibatches, according to the book?** p.125: When using mini-batches, each batch should be constructed so that it roughly reflects the overall class distribution and diversity of the full dataset. If the data are ordered by class and the first $n_b$ examples all belong to a single class, the first mini-batch will contain only that class and its gradient will be a poor stand-in for the gradient on the whole dataset. To avoid this problem, mini-batches should be formed by randomly sampling or shuffling the data before splitting into batches.

**Q3: Provide an example of modifications in a loss function and in data that can be done to take into account the data imbalance, according to the book?** According to page 300 of the book, data imbalance can be addressed in two main ways. First, the loss function can be modified by assigning higher weights to data points from the minority or more important class, making their misclassification more costly. Second, the data itself can be adjusted by upsampling the minority-class examples so they appear more frequently during training. These two approaches help the model focus more on underrepresented classes and improve performance on imbalanced datasets.

## Appendix

```r
library(stringr)
library(glmnet)

data <- read.csv("data/tecator.csv")

n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.5))
train=data[id,]

# the ids not in id
id1=setdiff(1:n, id)
test=data[id1,]
index <- str_detect(colnames(train), "(Channel)|(Fat)")
model_linreg <- lm(Fat ~ ., train[, index])
summary(model_linreg)

MSE_t <- mean((model_linreg$fitted.values - train$Fat)^2)

y_hat_test <- predict(model_linreg, test)

mse_fun <- function(y_hat, y) {
  mean((y - y_hat)^2)
}

print("Train MSE")
MSE_t <- mse_fun(model_linreg$fitted.values, train$Fat)
MSE_t
print("Test MSE")
MSE_test <- mse_fun(y_hat_test, test$Fat)
MSE_test

par(mfrow = c(1,2))
hist(model_linreg$fitted.values - train$Fat, breaks = 20, main = "Residuals in train")
hist(y_hat_test - test$Fat, breaks = 20, main = "Residuals in test")
```

```r
X_train <- model.matrix(Fat ~ ., train[, index])

model_lasso <- glmnet(X_train, train$Fat, alpha = 1)



plot(model_lasso)

print("Channels are almost 1 to 1 correlated")
cor(X_train)[2:6, 2:6] %>% round(6)

coefficients <- sapply(1:ncol(model_lasso$beta), function(x) nrow(model_lasso$beta) - sum(model_lasso$be
plot(model_lasso$lambda, coefficients)

lambda_index <- which(coefficients <= 3) %>% max()

print("Smallest lambda tested that results in 3 or fewer nonzero coefficients:")
model_lasso$lambda[lambda_index]


model_ridge <- glmnet(X_train, train$Fat, alpha = 0)

par(mfrow = c(1, 2))
plot(model_lasso)
plot(model_ridge)


model_cross_val <- cv.glmnet(X_train, train$Fat, alpha = 1)
plot(model_cross_val)

best_index <- which.min(model_cross_val$cvm)

print("Best lambda according to lowest mean CV-loss:")
model_cross_val$lambda[best_index]
print("log(lambda_optimal)")
model_cross_val$lambda[best_index] %>% log()

print("Amount of nonzero coefficients:")
model_cross_val$nzero[best_index]



model_final <-glmnet(X_train, train$Fat, alpha = 1, lambda = model_cross_val$lambda[best_index])

X_test <- model.matrix(Fat ~ ., test[, index])

y_hat_test_lasso <- predict(model_final, X_test)
plot(test$Fat, y_hat_test_lasso)
abline(a = 0, b = 1, col = "red", lwd = 2)

print("Test MSE")
MSE_test <- mse_fun(y_hat_test_lasso, test$Fat)
MSE_test
```

```r
#######################
#
# TASK 2 CODE
#
#######################
library(data.table)
library(rpart)
library(knitr)

data <- fread('data/bank-full.csv', stringsAsFactors = FALSE)
data$duration <- NULL
categorical_cols <- c('job', 'marital', 'education', 'default', 'housing', 'loan',
                      'contact', 'month', 'day', 'poutcome', 'y')

for (col in categorical_cols) {
  data[[col]] <- factor(data[[col]], levels = unique(data[[col]]))
}

n=dim(data)[1]
set.seed(12345)
id=sample(1:n, floor(n*0.4))
train=data[id,]
id1=setdiff(1:n, id)
set.seed(12345)
id2=sample(id1, floor(n*0.3))
valid=data[id2,]
id3=setdiff(id1,id2)
test=data[id3,]

# a. Decision tree with default settings
tree_default <- rpart(y ~ ., data = train, method = "class")

# b. Decision tree with smallest allowed node size (minbucket) = 7000
tree_minbucket <- rpart(y ~ ., data = train, method = "class", control = rpart.control(minbucket = 7000

# c. Decision tree with minimum deviance (mindev) = 0.0005
# Note: rpart does not have a mindev control parameter; deviance threshold is controlled differently by
# We first need to obtain the deviance at the root node and then calculate the ratio between our
# absolute minimum deviance threshold of 0.0005 and that to use the output as the input to the cp
# parameter in rpart

tmp <- rpart(y ~ ., data = train, method = "class",
             control = rpart.control(cp = 0.0, xval = 0))
D_root <- tmp$frame$dev[1]  # deviance at the root node
mindev_target <- 0.0005  # absolute deviance reduction we want
cp_val <- mindev_target / D_root

tree_cp <- rpart(y ~ ., data = train, method = "class", control = rpart.control(cp = cp_val))

misclass_rate <- function(model, data) {
  pred <- predict(model, newdata = data, type = "class")
  mean(pred != data$y)
}
```

```r
# Calculate misclassification rates
results <- data.frame(
  Model = c("Default", "Minbucket 7000", "CP 0.0005"),
  Train_Misclass = c(misclass_rate(tree_default, train),
                     misclass_rate(tree_minbucket, train),
                     misclass_rate(tree_cp, train)),
  Valid_Misclass = c(misclass_rate(tree_default, valid),
                     misclass_rate(tree_minbucket, valid),
                     misclass_rate(tree_cp, valid))
)

print(results)

num_splits <- c(
  Default = length(tree_default$frame$var[tree_default$frame$var != "<leaf>"]),
  Minbucket7000 = length(tree_minbucket$frame$var[tree_minbucket$frame$var != "<leaf>"]),
  CP_0.0005 = length(tree_cp$frame$var[tree_cp$frame$var != "<leaf>"])
)

print(num_splits)

library(rpart.plot)

# Plot 3 original trees side-by-side for comparison
par(mfrow = c(1, 3))

rpart.plot(tree_default, main = "Default Settings")
rpart.plot(tree_minbucket, main = "Minbucket = 7000")
rpart.plot(tree_cp, main = "cp = 0.0005")

par(mfrow = c(1, 1))
n_leaves <- function(fit) sum(fit$frame$var == "<leaf>")

tree_deviance <- function(fit, data) {
  p_hat <- predict(fit, newdata = data, type = "prob")
  y <- data$y
  idx <- cbind(seq_len(nrow(p_hat)), match(y, colnames(p_hat)))
  -2 * mean(log(p_hat[idx]))
}

# This extracts the vector of complexity-parameter values from the complexity-parameter
# table (cptable) stored in the rpart fit object.
# We're iterating through pruned trees stored in this table which should allow us to
# determine the optimal number of leaves
cp_seq <- tree_cp$cptable[, "CP"]

dev_list <- lapply(seq_along(cp_seq), function(i) {
  fit_i <- prune(tree_cp, cp = cp_seq[i])
  data.frame(
    cp        = cp_seq[i],
    leaves    = n_leaves(fit_i),
    train_dev = tree_deviance(fit_i, train),
    valid_dev = tree_deviance(fit_i, valid)
  )
})

dev_df <- do.call(rbind, dev_list)
dev_df_50 <- subset(dev_df, leaves <= 50)

library(ggplot2)

ggplot(dev_df_50, aes(x = leaves)) +
  geom_line(aes(y = train_dev, colour = "Train")) +
  geom_line(aes(y = valid_dev, colour = "Validation")) +
  labs(x = "Number of leaves", y = "Deviance", colour = "Data") +
  theme_minimal()

# Optimal subtree (within 50 leaves) by validation deviance
opt_row <- dev_df_50[which.min(dev_df_50$valid_dev), ]
opt_leaves <- opt_row$leaves
opt_cp     <- opt_row$cp

tree_opt <- prune(tree_cp, cp = opt_cp)
print(paste("Optimal number of leaves:", opt_leaves))
# Variable importance
sort(tree_opt$variable.importance, decreasing = TRUE)
## To get percentage variables:
# imp_raw  <- tree_opt$variable.importance
# imp_pct  <- 100 * imp_raw / sum(imp_raw)
# sort(imp_pct, decreasing = TRUE)
models <- list(
  "Default"        = tree_default,
  "Minbucket 7000" = tree_minbucket,
  "CP 0.0005"      = tree_cp,
  "Optimal 16-leaf" = tree_opt
)

misclass_df <- do.call(rbind, lapply(models, function(m) {
  c(
    Train = misclass_rate(m, train),
    Valid = misclass_rate(m, valid)
  )
}))

misclass_df <- data.frame(
  Model = rownames(misclass_df),
  Train_Misclass = misclass_df[, "Train"],
  Valid_Misclass = misclass_df[, "Valid"],
  row.names = NULL
)

kable(
  misclass_df,
  digits = 3,
  caption = "Train and validation misclassification rates for tree models"
)
```

```r
# Plot the tree for interpretation
library(rpart.plot)
rpart.plot(tree_opt, type = 2, extra = 104)

format_cm <- function(cm) {
  # positive class is "yes" and negative is "no"
  tp <- cm["yes", "yes"]
  fn <- cm["yes", "no"]
  fp <- cm["no",  "yes"]
  tn <- cm["no",  "no"]

  cm_labeled <- matrix(
    c(
      paste0(tp, " (TP)"),
      paste0(fn, " (FN)"),
      paste0(fp, " (FP)"),
      paste0(tn, " (TN)")
    ),
    nrow = 2, byrow = TRUE,
    dimnames = list(
      Observed  = c("Observed yes", "Observed no"),
      Predicted = c("Predicted yes", "Predicted no")
    )
  )
  return(cm_labeled)
}

pred_test <- predict(tree_opt, newdata = test, type = "class")

cm <- table(Observed = test$y, Predicted = pred_test)
cm_labeled <- format_cm(cm)
kable(
  cm_labeled,
  caption = "Confusion matrix for tree_opt on test data (TP = true positives, FP = false positives, FN =
)

calculate_metrics <- function(pred_test, test, cm) {
  accuracy <- mean(pred_test == test$y)
  print(paste("Accuracy:", accuracy))

  positive <- "yes"

  tp <- cm[positive, positive]
  fp <- sum(cm[, positive]) - tp
  fn <- sum(cm[positive, ]) - tp

  precision <- tp / (tp + fp)
  recall    <- tp / (tp + fn)
  f1        <- 2 * precision * recall / (precision + recall)

  print(paste("Precision:", precision))
  print(paste("Recall:", recall))
  print(paste("F1:", f1))
```

```r
}
calculate_metrics(pred_test, test, cm)
loss_mat <- matrix(
  c(0, 1,    # observed = "no":  pred "no", pred "yes"
    5, 0),  # observed = "yes": pred "no", pred "yes"
  nrow = 2, byrow = TRUE,
  dimnames = list(
    Observed  = c("no", "yes"),
    Predicted = c("no", "yes")
  )
)

loss_mat

# Fit cost-sensitive tree on training data
tree_loss <- rpart(
  y ~ .,
  data   = train,
  method = "class",
  parms  = list(split="information", loss = loss_mat),
  # control = ctrl
)

pred_test_loss <- predict(tree_loss, newdata = test, type = "class")
cm_loss <- table(Observed = test$y, Predicted = pred_test_loss)
cm_df <- format_cm(cm_loss)
kable(
  cm_df,
  caption = "Confusion matrix using custom loss matrix (rows = Observed, columns = Predicted)"
)
calculate_metrics(pred_test_loss, test, cm_loss)
library(dplyr)

prob_yes_tree <- predict(tree_opt, newdata = test, type = "prob")[, "yes"]

# Fit logistic regression (binomial GLM with logit link)
logit_fit <- glm(
  y ~ .,
  data   = train,
  family = binomial(link = "logit")
)

# summary(logit_fit)
prob_yes_logit <- predict(logit_fit, newdata = test, type = "response")

pi_grid <- seq(0.05, 0.95, by = 0.05)

metrics_at_threshold <- function(thresh, y_true, p_yes) {
  pred_yes <- p_yes > thresh
  y_pos    <- y_true == "yes"
  y_neg    <- y_true == "no"

  tp <- sum(pred_yes & y_pos)
  fp <- sum(pred_yes & y_neg)
  fn <- sum(!pred_yes & y_pos)
  tn <- sum(!pred_yes & y_neg)

  tpr <- ifelse(tp + fn == 0, NA, tp / (tp + fn)) # sensitivity/recall
  fpr <- ifelse(fp + tn == 0, NA, fp / (fp + tn))

  c(threshold = thresh, TPR = tpr, FPR = fpr)
}

roc_tree <- t(sapply(pi_grid, metrics_at_threshold,
                     y_true = test$y, p_yes = prob_yes_tree))
roc_tree <- as.data.frame(roc_tree)


# ROC data for logistic regression
roc_logit <- as.data.frame(
  t(sapply(pi_grid, metrics_at_threshold,
           y_true = test$y, p_yes = prob_yes_logit))
)

plot(roc_tree$FPR, roc_tree$TPR, type = "b", pch = 19, col = "blue",
     xlab = "False Positive Rate (FPR)",
     ylab = "True Positive Rate (TPR)",
     main = "ROC curves: tree_opt vs logistic regression",
     xlim = c(0, 1), ylim = c(0, 1))

lines(roc_logit$FPR, roc_logit$TPR, type = "b", pch = 17, col = "red")

abline(0, 1, lty = 2, col = "grey")
legend("bottomright",
       legend = c("Tree (tree_opt)", "Logistic regression"),
       col    = c("blue", "red"),
       pch    = c(19, 17),
       bty    = "n")
pr_at_threshold <- function(thresh, y_true, p_yes) {
  pred_yes <- p_yes > thresh
  y_pos    <- y_true == "yes"
  y_neg    <- y_true == "no"

  tp <- sum(pred_yes & y_pos)
  fp <- sum(pred_yes & y_neg)
  fn <- sum(!pred_yes & y_pos)

  precision <- ifelse(tp + fp == 0, NA, tp / (tp + fp))
  recall    <- ifelse(tp + fn == 0, NA, tp / (tp + fn))  # = TPR

  c(threshold = thresh, Precision = precision, Recall = recall)
}


pr_tree <- as.data.frame(
  t(sapply(pi_grid, pr_at_threshold,
```

```r
           y_true = test$y, p_yes = prob_yes_tree))
)

pr_logit <- as.data.frame(
  t(sapply(pi_grid, pr_at_threshold,
           y_true = test$y, p_yes = prob_yes_logit))
)

plot(pr_tree$Recall, pr_tree$Precision, type = "b", pch = 19, col = "blue",
     xlab = "Recall",
     ylab = "Precision",
     xlim = c(0, 1), ylim = c(0, 1),
     main = "Precision-Recall curves: tree_opt vs logistic regression")

lines(pr_logit$Recall, pr_logit$Precision, type = "b", pch = 17, col = "red")

legend("topright",
       legend = c("Tree (tree_opt)", "Logistic regression"),
       col    = c("blue", "red"),
       pch    = c(19, 17),
       bty    = "n")

text(pr_tree$Recall, pr_tree$Precision,
     labels = round(pr_tree$threshold, 2),
     pos = 4, cex = 0.6, col = "blue")

text(pr_logit$Recall, pr_logit$Precision,
     labels = round(pr_logit$threshold, 2),
     pos = 4, cex = 0.6, col = "red")
library(ggplot2)
set.seed(12345)
######################  Assignment 3.1 #########################################
data <- read.csv("data/communities.csv",header = TRUE, sep = ",")

# Remove response
features <- data[, setdiff(names(data), "ViolentCrimesPerPop")]

# Scale predictors
scaled_features <- scale(features)

# Covariance matrix
S <- cov(scaled_features)

# Eigen decomposition
eig <- eigen(S)


# proportion of variation explained by each of the first 100 components

lambda <- eig$values
exp_var <- lambda / sum(lambda)        # proportion of variance explained
cum_var <- cumsum(exp_var)            # cumulative proportion
q_95 <- which(cum_var >= 0.95)[1]      # first PC reaching 95%
```

```r
# plot
plot(exp_var, type = "p",
     xlab = "Component",
     ylab = "Proportion of variation explained",
     main = "Proportion of variation explained by each of the first 100 components")


cat("Number of components needed for 95% variance:", q_95, "\n")
cat("Proportion of variation explained by the first component:", lambda[1] / sum(lambda), "\n")
cat("Proportion of variation explained by the second component:", lambda[2] / sum(lambda), "\n")

######################  Assignment 3.2 ##########################################
PCA <- princomp(scaled_features, cor = TRUE)

# 1. TRACE PLOT OF FIRST PC
plot(PCA$loadings[,1],
     type = "l",  # vertical lines
     main = "Trace Plot: Loadings for PC1",
     xlab = "Feature Index",
     ylab = "PC1 Loading")



# 2. TOP 5 FEATURES CONTRIBUTING TO PC1

# Extract PC1 loadings
pc1_loadings <- PCA$loadings[,1]

# Sort by absolute value
top5_pc1 <- sort(abs(pc1_loadings), decreasing = TRUE)[1:5]

cat("Top 5 PC1 contributors (absolute loadings):\n")
print(top5_pc1)

# 3. GET PC SCORES FOR PC1 & PC2

scores <- as.data.frame(PCA$scores)[,1:2]
colnames(scores) <- c("PC1", "PC2")

# Add crime back into scores
crime <- data$ViolentCrimesPerPop
scores$Crime <- crime

print(head(scores[, c("PC1", "PC2", "Crime")], 10))

# 4. PC1-PC2 SCORE PLOT (COLORED BY CRIME)
ggplot(scores, aes(x = PC1, y = PC2, color = Crime)) +
  geom_point(alpha = 0.8) +
  scale_color_gradient(low = "blue", high = "yellow") +
  theme_minimal() +
  labs(
    title = "PCA Score Plot: PC1 vs PC2",
    x = "PC1 (Socioeconomic)",
    y = "PC2 (Immigration / Density)",
    color = "Crime Rate"
  )
######################  Assignment 3.3 #########################################
# Train and test data
n <- nrow(data)
set.seed(12345)
id <- sample(1:n, floor(n*0.5))
train_set <- data[id,]
test_set <- data[-id,]

# -------------------------------
# 2. Scale features AND target
# -------------------------------

# function that returns scaled data AND retains mean/sd for later
scale_data <- function(df) {
  mu <- sapply(df, mean)
  sd <- sapply(df, sd)
  scaled <- as.data.frame(scale(df, center = mu, scale = sd))
  list(data = scaled, mean = mu, sd = sd)
}

# scale training set
train_scaled <- scale_data(train_set)
train_dat <- train_scaled$data

# scale test WITH training mean/sd
test_dat <- as.data.frame(
  scale(test_set,
        center = train_scaled$mean,
        scale = train_scaled$sd)
)

# -------------------------------
# 3. Fit linear regression
# -------------------------------

model <- lm(ViolentCrimesPerPop ~ ., data = train_dat)

# -------------------------------
# 4. Compute training & test MSE
# -------------------------------

# predictions
train_pred <- predict(model, train_dat)
test_pred  <- predict(model, test_dat)

# mean squared errors
train_mse <- mean((train_dat$ViolentCrimesPerPop - train_pred)^2)
test_mse  <- mean((test_dat$ViolentCrimesPerPop - test_pred)^2)

cat("Train mean squared error:", train_mse)
```

```r
cat("Test mean squared error:", test_mse)
###################### Assignment 3.4 ##########################################

# Prepare design matrices without intercept
X_train <- model.matrix(ViolentCrimesPerPop ~ . - 1, train_dat)
X_test  <- model.matrix(ViolentCrimesPerPop ~ . - 1, test_dat)

y_train <- train_dat$ViolentCrimesPerPop
y_test  <- test_dat$ViolentCrimesPerPop

p <- ncol(X_train)

train_err <- list()
test_err <- list()

i <<- 1     # counter must be global

cost_fun <- function(theta_vec){

  theta <- matrix(theta_vec, ncol = 1)

  # predictions
  pred_train <- X_train %*% theta
  pred_test  <- X_test  %*% theta

  # MSE
  train_mse <- mean((y_train - pred_train)^2)
  test_mse  <- mean((y_test  - pred_test )^2)

  # store errors
  train_err[[i]] <<- train_mse
  test_err[[i]]  <<- test_mse
  i <<- i + 1

  return(train_mse)   # BFGS minimizes training MSE
}

theta0 <- rep(0, p)

opt <- optim(theta0, fn = cost_fun, method = "BFGS",
             control = list(maxit = 3000))

# convert lists to numeric
train_vec <- as.numeric(train_err)
test_vec  <- as.numeric(test_err)

# drop first 500 iterations for clarity
start <- 500

plot(train_vec[start:length(train_vec)], type="l", col="red",
     ylim = range(c(train_vec[start:length(train_vec)],
                    test_vec[start:length(test_vec)])),
     xlab="Iteration", ylab="MSE",
     main="Training vs Test MSE During BFGS Optimization")

lines(test_vec[start:length(test_vec)], col="blue")
abline(v = which.min(test_vec) - start, col="green")

legend("topright", legend=c("Train MSE","Test MSE"),
       col=c("red","blue"), lty=1, cex=0.8)

best_iter <- which.min(test_vec)

cat("Iteration for early stopping:", best_iter, "\n")
cat("Train mean squared error:", train_vec[best_iter], "\n")
cat("Test mean squared error:",  test_vec[best_iter],  "\n")
```