Predictive Modeling Exam

Lucas Cruz Fernandez

05.08.2021

Notes:

- For access to single files relating to each problem, go to my Github repository (available from 06.08.21 onwards)
- Functions and other steps are described in a more detailed way in comments inside the code, so I do not explain them at length again in the text.
- The seed is set to 2021 such that replication yields the same results as discussed in the text. Necessary packages are loaded but not included in PDF.

Problem 1

The goal is to predict a binary outcome using three different prediction methods and evaluate and discuss their performance.

a) Data splitting

As a first step, I split up the data into a training and a test set. The function *test_train_sample* below does exactly this by generating a vector of random integers that has the size the test set is supposed to have, slices the data accordingly and returns both sets as part of a named list.

```
test_train_sample=function(dat, test_n=1000){
  #split a sample in dat into a test and training sample
  #test_n defines size of test sample created
  #create slicing
  #need to set seed first - and individually before sampling each time bc not globally set
  set.seed(2021)
  slices=sample.int(test_n, n=nrow(dat))
  #create subsets
  train=dat[-slices, ]
  test=dat[slices, ]
  #return them in a named list
  dat_list=list('train'=train, 'test'=test)
  return (dat_list)
}
```

Lets now generate the two sets, which are supposed to be equally sized. First read in the data, then split it into two. The code below also generates a dataframe that for now only contains a column with the true values of the test set. The predicted values for each method will be added to this dataframe one-by-one later on.

```
#read in data
heart=read.csv('heart.csv')
#apply function test_train_sample
test_train_data=test_train_sample(heart, test_n=dim(heart)[1]/2)
```

```r
#retrieve test and training sample from named list that is returned by function
test_dat=test_train_data$test
train_dat=test_train_data$train
test_preds=test_dat%>%dplyr::select(y)
```

b) Prediction

I want to compare three different prediction methods for a binary outcome. The three methods of choice here are:

1. LASSO Logit Regression
2. Random Classification Forest
3. Nonparametric Regression

Lasso Logit Regression

As I am interested in modeling a binary outcome, the Logistic Regression model (Logit) comes to mind fairly quickly. Contrary to the Linear Probability Model (LPM) that predicts the conditional probability of an event happening, Logit models the log odds of the outcome being equal to one conditional on all observables. The advantage is that the Logit model is non-linear in probabilities but still a linear model of the log odds. Hence, it is easy to estimate; we can translate the odds predicted into probabilities later on, and - as we have seen in class - log-odds are better approximated by a linear function than odds themselves. At the same time, if one is interested in the role of single variables, its coefficients are still somewhat interpretable - even though they lose the clear interpretability, the LPM offers. However, the Logit has the major advantage compared to the LPM that the predicted probabilities will always lie within the interval [0, 1] and not outside it - as may be the case with the LPM. Additionally, as we are only interested in prediction, the interpretability of the coefficients plays no role in the setup here. To improve the model's predictions, I add interaction terms between all variable pairs and use a Lasso regularization to counter the loss in degrees-of-freedom otherwise occurring. The code below runs a cross-validated Lasso Logit regression using the *cv.glmnet* function provided in the *glmnet* R package. First, a matrix containing all observables from the training set and their respective interactions is generated before the model is fit and then used to predict the probabilities of the test set using the observables in the test set.

```r
#create a model matrix from all variables that contains interactions between observables
#delete intercept
#note: raising . to power of 2 adds all dual interactions to model matrix
x_train=model.matrix(y~.^2, data=train_dat)[, -1]
#then get y
y_train=train_dat$y
#and fit cross-validated Lasso
lasso_logit=cv.glmnet(x_train, y_train, alpha=1, family='binomial')
#then get predictions
#for these need model matrix of test data
x_test=model.matrix(y~.^2, data=test_dat)[, -1]
#use lambda.1se predictions as those are convention (seen on stackechange)
#use type=response as this automatically returns probabilities, not odds
test_preds$ll_preds=as.vector(predict(lasso_logit, newx=x_test, s='lambda.min',
                                      type='response'))
```

Random Forest

The Random Forest is the perhaps most widely used prediction algorithm in machine learning tasks. It performs quite well on various tasks and needs no specification of any relationship between observables and outcome, which allows it to detect any pattern of interactions as they are not ruled out from the beginning (as is the case in parametric specifications). However, it is pretty data-hungry and usually needs a lot of

data to identify patterns in the data clearly. I will discuss the possible implications of this later on in the evaluation section. Compared to single trees, the random forest is way more stable because it is not so prone to random disturbances in the training data as those are "averaged out" by the aggregation of many single trees. The code below uses the *randomForest* function of the *randomForest* package in R to fit a random classification forest to the data. Based on the observables, the forest learns to classify each observation to have either an outcome equal to zero or one. Taking the mean of each leave, this can be interpreted as the probability of having an outcome equal to one conditional on the observables. However, the random forest may return probabilities outside of the interval [0, 1]. Hence, the function *adjust_probs* sets all values below zero to zero and all values above one to one. The warning thrown by the *randomForest* function due to the outcome having less than five values can be ignored.

```
adjust_probs=function(probs_series){
  #Some predictions might result in probabilities outside of interval [0, 1].
  #Simply set those predictions < 0 to 0 and > 1 to 1.
  #all < 0 to 0
  probs_series=pmax(0, probs_series)
  #all > 1 to 1
  probs_series=pmin(1, probs_series)
  return(probs_series)
}

#fit random forest on train data
rf=randomForest(y~., data=train_dat)
```

```
## Warning in randomForest.default(m, y, ...): The response has five or fewer
## unique values. Are you sure you want to do regression?
```

```
#again get predictions
test_preds$rf_preds=predict(rf, newdata=test_dat)
#adjust the probability predictions that are outside of [0, 1]
test_preds$rf_preds=adjust_probs(test_preds$rf_preds)
```

Nonparametric Regression

Lastly, I apply a nonparametric regression to predict the outcome. Similar to the random forest, the nonparametric regression makes no assumptions on the functional form of the relationship between observables and outcome. Here I use a 1-dimensional nonparametric regression using the observable that has the highest importance in the random forest fit to the data before. This importance is measured by the mean increase in purity, which measures how much the MSE is reduced when the algorithm splits on this variable and then averaged across all instances in the random forest where the variable is split on. The code below finds this variable using the function *most_important_rf* and then applies the *loess* function for nonparametric regression to fit the model.

```
#can do nonparametric prediction based on one variable
#variable that has highest importance based on Mean Increase in Purity
most_important_rf=function(rf){
  #get most important variable in rf based on node purity increase
  importance_df=data.frame(rf$importance)%>%mutate(variable=rownames(.))
  #get name of most important variable
  mi_var=importance_df[which(importance_df['IncNodePurity']==
                               max(importance_df['IncNodePurity'])), 'variable']
  return(mi_var)
}
mi_rf=most_important_rf(rf)
#it's age!
#then use loess function
```

```r
np=loess(y~age, data=train_dat)
#then make prediction
test_preds$np_preds=predict(np, newdata=test_dat)
#adjust the probability predictions that are outside of [0, 1]
test_preds$np_preds=adjust_probs(test_preds$np_preds)
```

c) & d) Evaluation & Discussion

In this section, I combine tasks c) and d) by discussing the evaluation outcomes on the go. As already mentioned, I expect some potential issues for the random forest and the nonparametric regression as both require lots of information, but the data only contains 462 observations. For some evaluation methods having a *long data.frame* version of the dataframe containing the predictions and true values is useful and thus generated in the code below.

```r
#There are different methods for evaluation of probability predictions for
#binary outcomes; to calculate those need a long data.frame
test_preds_long=melt(test_preds, measure.vars=-test_preds$y, variable.name='pred_method',
                     value.name='pred_vals')
```

MSE

To get a first overview of the performance of the 3 predictors chosen, I take a look at the Mean Squared Error (MSE) and the Log-Score loss function. Note that another loss function based measure - the Brier Score - is computationally equivalent to the MSE and is therefore not specifically shown here.

```r
#Calculate MSE for different prediction methods
scores=test_preds_long%>%group_by(pred_method)%>%summarize(mse=mean((y-pred_vals)^2,
                                                                    na.rm=TRUE),
Log=mean(-y*log(pred_vals)-(1-y)*log(1-pred_vals),
na.rm=TRUE))%>%
  mutate(pred_method=c('Lasso Logit', 'Random Forest', 'NP Regression'))
#kable function from knitr package displays data structures
#such as data.frames and tibbles in a nice table in Rmd
kable(scores, col.names=c('Method', 'MSE', 'Log Score'),
      caption='MSE & Log Score for all 3 prediction methods')
```

Table 1: MSE & Log Score for all 3 prediction methods

Method	MSE	Log Score
Lasso Logit	0.1697405	0.5139703
Random Forest	0.1804271	0.5424939
NP Regression	0.1913303	0.5608622

Of the three methods, the Lasso Logit regression performs the best in terms of both of these scores, while the random forest and the nonparametric regression lie closer together. Still, the random forest slightly outperforms NP. The reason for their worse performance compared to the parameterized approach Lasso Logit follows may be the small sample size as both predictors are quite data-hungry. Still, the random forest can use more variables than the NP regression - which would otherwise suffer from the curse of dimensionality - which leads it to perform a bit better.

ROC Curves

For a different look at the performance of the prediction methods, I plot the Receiver Operating Characteristics Curve (ROC Curve). The predicted probabilities are used to classify the outcome - i.e., if the predicted

probability is larger than some threshold z, the predicted outcome is set to be one, otherwise zero. The ROC Curve plots the hit rate (HR) against the false alarm rate (FAR), the probability of correct and wrong classification given z, respectively. Ideally, there is a value z at which the classification is perfect, resulting in a $HR = 1$ (or $FAR = 0$). However, this is usually not the case, and a prediction method can be evaluated because given a threshold value z it has a high HR, while the FAR stays relatively low. Moving along the x-axis can be understood as allowing for a higher FAR to improve the HR. Figure 1 depicts the ROC curve for all three prediction methods.

```
#use the plotROC library and ggplot to visualize the ROC Curve
p_ROC=ggplot(data=test_preds_long) +
  geom_roc(aes(d=y, m=pred_vals, color=pred_method),
           pointsize=0.3, position='identity') +
  xlab('FAR') +
  ylab('HR') +
  labs(color='Prediction Method', title='ROC') +
  theme(legend.position='bottom',
        legend.key.size = unit(0.5, 'cm'),
        #legend.key.height = unit(0.5, 'cm'),
        #legend.key.width = unit(0.5, 'cm'),
        legend.title = element_text(size=9),
        legend.text = element_text(size=7))
p_ROC
```

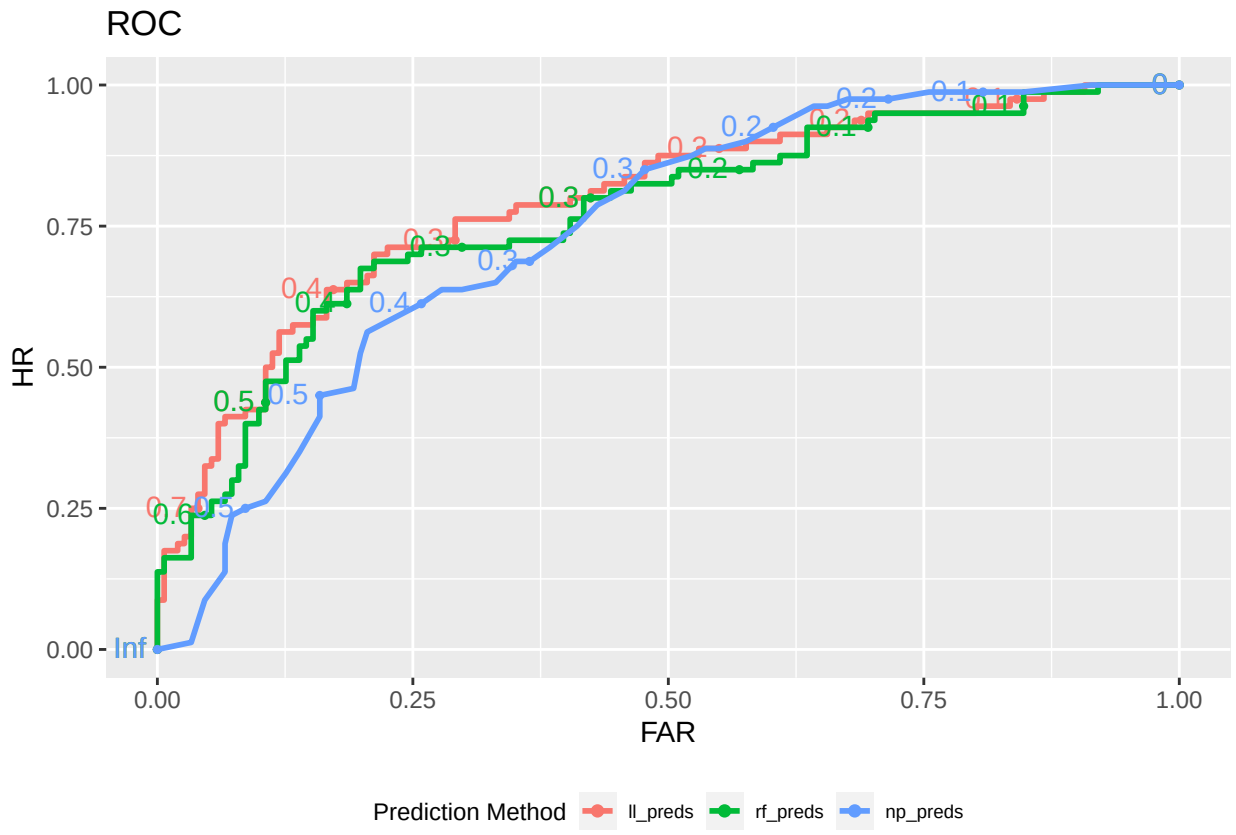


Figure 1: ROC Curve for all 3 prediction methods

The lasso logit and the random forest perform quite similarly: a slightly higher FAR - by reducing the threshold - results in strong gains in terms of the HR up to a certain threshold (around 0.3), at which the FAR starts to increase faster than the HR when reducing the threshold. The NP regression, however, performs the

worst with the lowest hit rate for a given threshold and FAR. This strengthens the picture already seen in Table 1 that the NP regression suffers from the lack of data and information.
To further quantify the visual evaluation of the ROC curve, I calculate the Area under the Curve (AUC). A value of $\frac{1}{2}$ implies that the ROC curve is equal to the 45 degree line, which is equivalent to randomly guessing the classification. Instead, the value moves closer to one, the better the classification becomes (1 signaling perfect classification).

```
#calculate AUC, change group column entries to estimator names
#and drop first column that contains no information
auc=calc_auc(p_ROC)%>%mutate(group=c('Lasso Logit', 'Random Forest',
                                     'NP Regression'))%>%select(c(2, 3))
#plot auc in table
kable(auc, col.names=c('Method', 'AUC'), caption='AUC Scores for each method')
```

Table 2: AUC Scores for each method

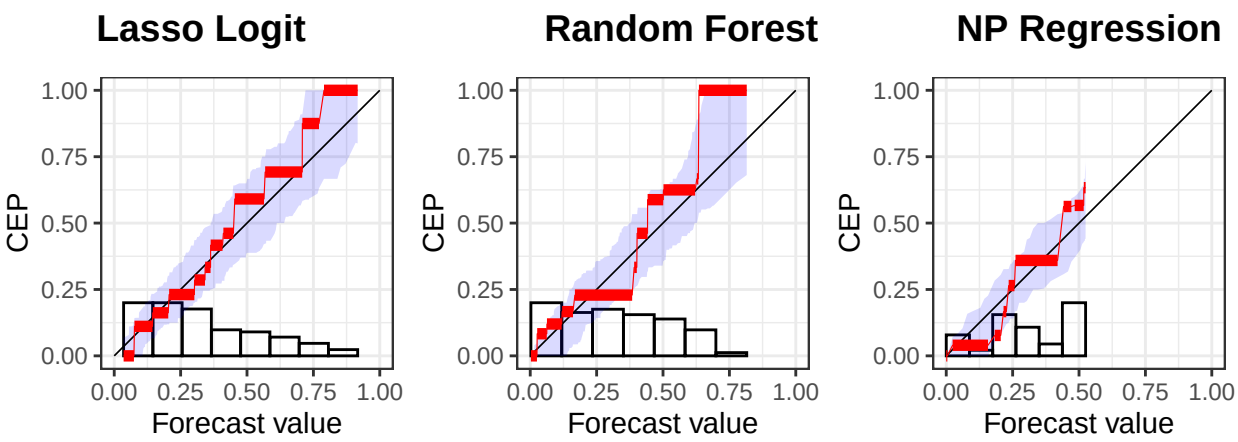
Method	AUC
Lasso Logit	0.7926325
Random Forest	0.7699503
NP Regression	0.7406043

The AUC values confirm the conclusions drawn from the ROC curve plot, underlining the performance of the Lasso Logit, while NP regression is indeed offering the worst performance. The measures do not contain any information on how close the predictions get to the true probability of the outcome being one (i.e., their precision) but instead offer insight into how well outcomes can be distinguished given the prediction.

Reliability Diagram

To evaluate the precision of the probability prediction, I inspect the reliability diagram of each prediction method. By plotting the non-parametrically estimated conditional expected probability of having an outcome equal to one against the predicted values, it becomes visible whether the predicted values actually display the expected probability - this feature is called auto-calibration. The figure below shows the respective reliability diagrams. The 45 degree line marks the optimal case, in which an individual with a predicted value of e.g. 0.5 actually has a conditional probability of 0.5 of its outcome being one. The red line depicts the actually relaized values.

```
#Lasso Logit
ll_rel=reliabilitydiag(y=test_preds$y, x=test_preds$ll_preds)
#RF
rf_rel=reliabilitydiag(y=test_preds$y, x=test_preds$rf_preds)
#Nonparametric Regression
np_rel=reliabilitydiag(y=test_preds$y, x=test_preds$np_preds)
#bind all single ggplot reliabilitydiagrams into one panel
#use the ggarrange function from ggpubr package
#adjust labels to prediction method names and move label higher up
#moving label doesn't work correctly right now
ggarrange(autoplot(ll_rel), autoplot(rf_rel), autoplot(np_rel), ncol=3, nrow=1,
          labels=c('Lasso Logit', 'Random Forest', 'NP Regression'), vjust=8)
```

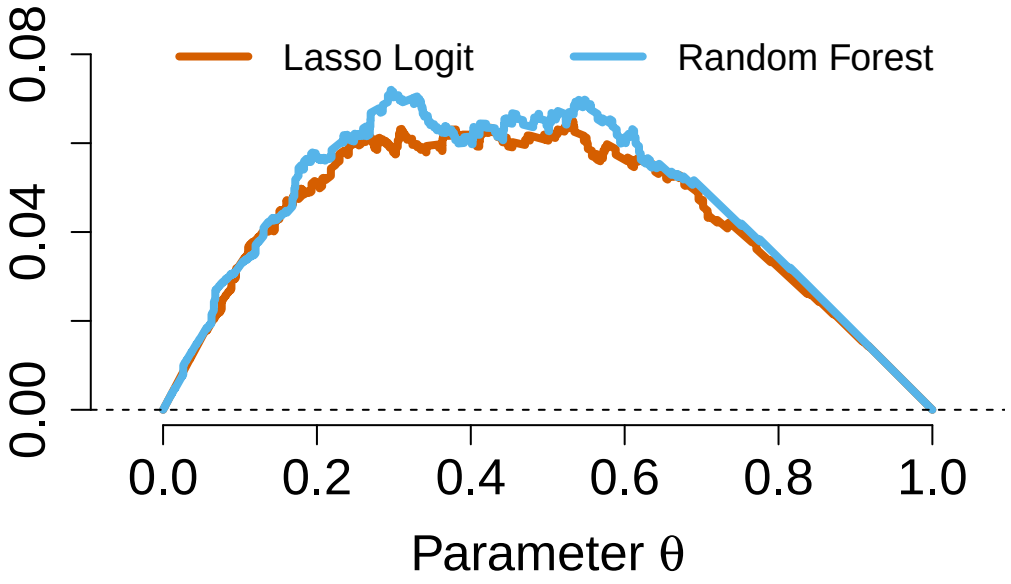


One can see clearly that the Lasso Logit regression performs the strongest here. Especially for small values, the predicted values are closely related to the expected conditional probability. However, the deviance increases with probabilities/predictions, which could be a sign that the group of individuals with a high probability of having an outcome equal is relatively small. Thus there is a lack in predictive power, or the relationship of having an outcome of one and the used predictors is very noisy. The first case is highly likely as only a third of the already small training set has an outcome equal to one. Compared to the Lasso Logit, both NP regression and Random Forest show a different pattern: while the Lasso Logit predictions result in a step function that rather closely follows the 45-degree line, the other two predictors have large intervals of expected conditional probability in which the same value is predicted. Additionally, in the case of the NP regression, there are no predicted probabilities larger than 0.6, while the random forest tends to overestimate the probability starting roughly at this value. Most likely, this is also related to the small size of the dataset. As already mentioned, both prediction methods are rather data-hungry (while the NP additionally suffers from the curse of dimensionality, which prohibits the use of all predictors), and there are not many individuals that have an outcome equal to one compared to zero, possibly resulting in this imprecise probability predictions. Additionally, both - NP and RF - are based on the idea of grouping similar observations. The random forest algorithm aims to put observations that are very similar into the same leaf and is therefore quite strong in making the correct classification (although lacking data) but using the averages of these laves tends to over-and underestimate the probability of the outcome being one. A similar effect could drive the rather weak performance of the NP regression.

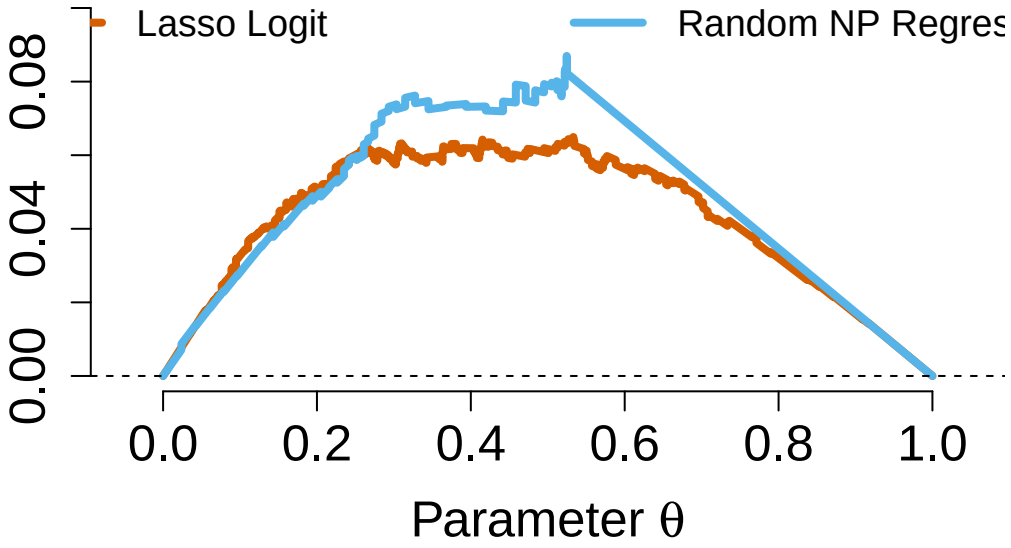
Murphy Diagram

Next up, I take a look at the Murphy diagrams for each prediction method. Again, this method shows how well the prediction methods help to classify the outcome correctly but at the same time allows to compare two prediction methods directly. For a given threshold θ, the outcome is classified as one with the predicted probability being larger than θ. The elementary score punishes false positives (classifying outcome as 1 with the true outcome being 0) with the weight θ and false negatives (classifying outcome as 0 with the true outcome being 1) with the weight $1-\theta$. Hence, accepting the possibility of more false positives by setting θ lower relaxes the punishment the elementary score puts on these false positives. The Murphy diagram depicts a curve showing the elementary score for two prediction methods over a range of threshold values θ.

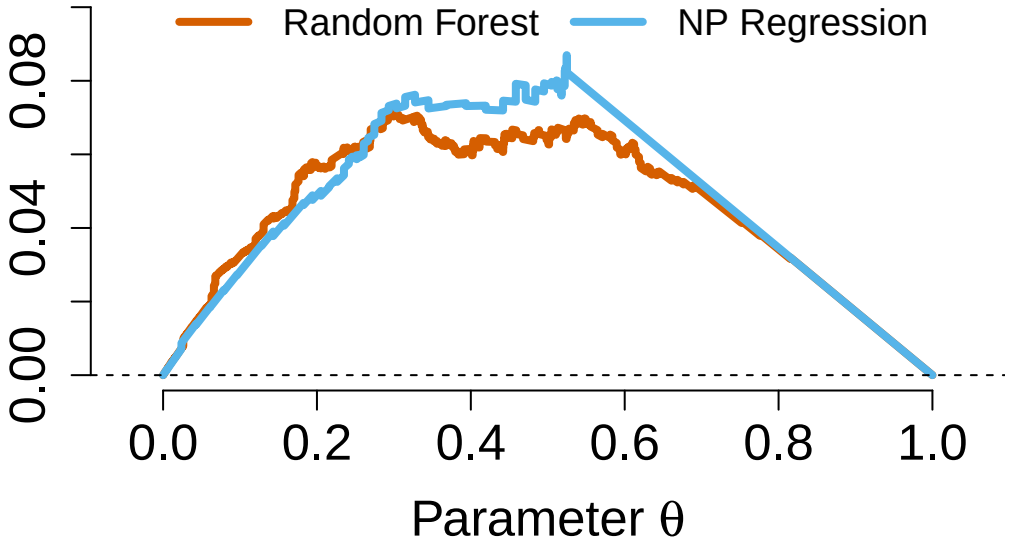
```
#Logit vs RF
murphydiagram(f1=test_preds$ll_preds,
              f2=test_preds$rf_preds,
              y=test_preds$y,
              labels=c('Lasso Logit', 'Random Forest'))
```




```
#Logit vs Nonparametric Regression
murphydiagram(f1=test_preds$ll_preds,
              f2=test_preds$np_preds,
              y=test_preds$y,
              labels=c('Lasso Logit', 'Random NP Regression'))
```




```
#RF vs Nonparametric Regression
murphydiagram(f1=test_preds$rf_preds,
              f2=test_preds$np_preds,
              y=test_preds$y,
              labels=c('Random Forest', 'NP Regression'))
```



The prior results are again confirmed, with the lasso logit regression outperforming both the nonparametric regression and the random forest, while the random forest does better than the NP regression. However, at the boundaries of the θ parameter values, the comparison of the methods shows that they perform equally. It again also becomes visible that there are no prediction values above a value of around 0.5 for the NP regression as from there on, the Murphy diagram plots a diagonal line. This prohibits drawing any conclusions comparing the behaviors of the other prediction methods with NP at the end of the θ values.

Score decomposition

Finally, to quantify the roles reliability and discrimination play in the MSE (or Brier score), I take a look at the score's decomposition. The function *reliabilitydiag* returns this decomposition of the Brier score (reminder: this is computationally equivalent to the MSE) by default. The respective decompositions are produced in the code below and shown in Table 3.

```
brier_decomps=summary(ll_rel)%>%add_row(summary(rf_rel))%>%add_row(summary(np_rel))%>%
  mutate(forecast=c('Lasso Logit', 'Random Forest', 'NP Regression'))
kable(brier_decomps,
      col.names=c('Method', 'Score', 'Miscalibration', 'Discrimination', 'Uncertainty'),
      caption='MSE/Brier score decomposition')
```

Table 3: MSE/Brier score decomposition

Method	Score	Miscalibration	Discrimination	Uncertainty
Lasso Logit	0.1697405	0.0099633	0.0666053	0.2263826
Random Forest	0.1804271	0.0162249	0.0621805	0.2263826
NP Regression	0.1913303	0.0080434	0.0430957	0.2263826

The decomposition is as follows: $S = MCB - DSC + UNC$ Intuitively, better discrimination, i.e., a larger DSC score, improves the capability of the prediction method to correctly classify the outcomes, hence improving the overall evaluation score. In contrast, larger uncertainty and miscalibration let the score deteriorate (increase as it is a loss score). Hence, larger values in the first and third column of Table X imply a worse performance in terms of score S, while a larger DSC score can be seen as positive. The results from the reliability diagram and the ROC curve are confirmed. The Lasso regression leads to the largest discrimination score, although not performing much better than the random forest. The latter, however, suffers from its weakness in auto-calibration, as already discussed above. The nonparametric regression cannot distinguish very well between outcomes but does not do as bad as the random forest when it comes to

miscalibration.

Concluding Remarks

Summarizing, it has become evident that the Lasso Logit regression performs the best among the three predictors I chose concerning both- reliability and discrimination. Although the Lasso Logit constrains the regression's functional form and parameter space, the two nonparametric approaches - the random forest and the np regression - most likely suffer from the lack of data, with the latter performing the worst as it additionally has the least information available.

Problem 2

Similarly to Problem 1, the goal is to predict an outcome given some observables using three different prediction methods. However, this time the outcome is a continuous variable: the number of insurance charges of the individual. Hence, some prediction methods are no longer part of the toolbox, while new ones are added.

a) Data splitting

As in Problem 1, the first step I take is to read in the data and split it into a training and test sample using the *test_train_sample* function. Again these samples are set to have the same size. Also, I create the same kind of dataframe containing only a column with the true values of the test data for now in which the predictions are saved later on.

```
#read in data
insurance=read.csv('insurance.csv')
#same procedure as in Problem 1
test_train_data=test_train_sample(insurance, test_n=dim(insurance)[1]/2)
test_dat=test_train_data$test
train_dat=test_train_data$train
#also prepare a dataframe in which predictions and true values are saved
test_preds=test_dat%>%dplyr::select(charges)
```

b) Prediction

This time the three methods I choose are:

1. Lasso Regression
2. K Nearest Neighbor Regression
3. Random Regression Forest

Lasso Regression

The Lasso regression in this setup is a simple linear regression with an added penalty term that penalizes coefficients being different from zero. I have already mentioned the advantages of regularizing the regression in the presence of many variables, mainly to reduce the degrees-of-freedom problem that might otherwise arise. I choose the Lasso approach over the Ridge approach as Lasso definitely rules out some predictors, which makes it useful as a stepping stone for the K-Nearest Neighbor regression presented in the next step.

```
#Lasso
#prepare training data: get x and interactions as well as y values as single
#dataframes/vectors
x_train=model.matrix(charges~.^2, data=train_dat)
y_train=train_dat$charges
#fit model using cross validation
```

```
lasso=cv.glmnet(x=x_train, y=y_train, alpha=1)
#prepare test data
x_test=model.matrix(charges~.^2, data=test_dat)
#then fit model to test data
#turn lasso predictions into vector as they are otherwise saved as a dataframe inside
#the dataframe with predictions
test_preds$lasso_preds=as.vector(predict(lasso, newx=x_test, s='lambda.1se'))
```

To determine which variables are used for the KNN regression, I choose the variables that are kept by the lasso regression, i.e. those with a non-zero coefficient. I retrieve these in the code below.

```
#get variables that are not removed by lasso for knn
#first get matrix of coefficients and remove intercepts
lasso_coefs=as.matrix(coef(lasso, s='lambda.1se'))[-1:-2, ]
#then extract names where coefficient is not 0
lasso_vars=names(lasso_coefs)[lasso_coefs!=0]
```

KNN Regression

The second approach I follow is the K-Nearest Neighbor regression. To predict an outcome given some observable values, the KNN estimator searches for the K closest observations in the training data based on the euclidean distance between their vector of observables. The outcome values of the K closest observations in the training data are then averaged and used as the prediction for the outcome of the observation in the test set. This algorithm is implemented in the function *knn_prediction*, while the function *euclid_dist* calculates the distance between two vectors of observables. For a more detailed description of the process in R, see the code comments. Like the nonparametric regression implemented in Problem 1, the KNN regression does not perform well when the feature space is large due to the curse of dimensionality. However, as mentioned above, I use only those variables that have not been eliminated by the Lasso regularization in the first prediction approach. Once this problem is circumvented, the KNN regression offers a straightforward way of estimating the conditional mean locally around a point without any specifications about the functional form. Similar to other nonparametric estimators, KNN has a bias-variance tradeoff: the larger K (the more neighbors are considered), the more generalized becomes the estimate, leading to an underfitted model with potentially large bias. On the other hand, using too few neighbors results in overfitting the model and high variance. Therefore, which is the best number of neighbors, I fit the estimator to the training data with various $K = k$ and choose the model that has the smallest MSE on the test set. This is implemented in the function *cv_knn_prediction*.

```
euclid_dist=function(a, b){
  #calculate euclidean distance between vectors/matrices/data.frames
  dist=sqrt(sum((a-b)^2))
  return(dist)
}

knn_prediction=function(x, y, k, x0){
  #need a distance matrix, i.e. for each point in x0 have a column with
  #distances to points in x (training set)
  #then rank the values in each column and get the ones from 1 to k
  #use the respective observations in the y vector to calculate the mean over
  #them as the prediction
  #set up vector in which predictions will be saved
  predictions=rep(NA, length(y))
  #then loop over each observation in test data
  for(i in 1:length(y)){
    #get euclidean distances between single row in test data and all rows
    #in training data
```

```
    euclid=apply(x, 1, euclid_dist, x0[i, ])
    #sort them by size and get first k values -> indices that are closest ones
    ranks=rank(euclid, ties.method='random')<=k
    #get corresponding y values in training data and get mean as prediction
    predictions[[i]]=mean(y[ranks])
  }
  return(predictions)
}

cv_knn_prediction=function(x, y, k_range, x0, y0){
  #this function makes KNN prediction using each value in k_range as number
  #of neighbors, then calculates MSE for each of them and returns predictions that
  #yield smallest MSE
  #create empty matrix that will be filled column wise with predictions for y
  #for different k
  predictions_k=matrix(NA, nrow=length(y), ncol=length(k_range))
  #and empty vector in which mse will be saved
  mses=rep(NA, length(k_range))
  for(i in 1:length(k_range)){
    #make prediction for given k
    predictions_k[, i]=knn_prediction(x=x, y=y, k=k_range[[i]], x0=x0)
    mses[i]=mean((y0-predictions_k[, i])^2)
  }
  #find index of predictions with smallest mse
  lowest=which(mses==min(mses))
  #get the predictions of the corresponding k
  predictions_fin=predictions_k[, lowest]
  #return these predictions, the min mse and the corresponding k
  return(list(predictions=predictions_fin, mse=mses[lowest], k_min=k_range[lowest]))
}

#use KNN based on variables that are selected by LASSO above
#lasso eliminates all variables but age, ageXbmi, ageXchildren and bmiXsmokeryes
#prepare data for knn predictions
knn_x_train=model.matrix(charges~.^2, data=train_dat)[, lasso_vars]
knn_y_train=train_dat$charges
knn_x_test=model.matrix(charges~.^2, data=test_dat)[, lasso_vars]
knn_y_test=test_dat$charges
#now get predictions using cv_knn_prediction
#function returns mse and optimal k as well, save only the predictions and the k,
#mse will be calculated later on anyways
cv_knn_results=cv_knn_prediction(x=knn_x_train, y=knn_y_train, k_range=seq(1, 15),
                                 x0=knn_x_test, y0=knn_y_test)
test_preds$knn_preds=cv_knn_results$predictions
k_min=cv_knn_results$k_min
```

Random Forest

Lastly, I also use a random forest for prediction in this case, as done in Problem 1, as it can also predict continuous outcomes. As already mentioned in Problem 1, the random forest is a highly flexible algorithm that performs well in many settings. The advantages and disadvantages are similar as before as the estimator itself is the same. As in the binary case, the random forest can detect interactions between variables that are highly complex, leading to the detection of patterns that one might not come up with in a parametric regression approach. Meanwhile, the size of the dataset might be an issue again with respect to the precision

of the prediction. With small datasets, it is possible that the random forest picks up noise in the training data and interprets it as valid patterns leading to high variance when predicting the test set outcomes.

```
#fit a random forest to training data
rf=randomForest(charges~., data=train_dat)
#then make predictions
test_preds$rf_preds=predict(rf, newdata=test_dat)
```

c) & d) Evaluation & Discussion

In this section, I use several methods to evaluate the quality of the predictions made above and discuss them in the appropriate context. Before starting to do so I create a long dataframe again, equivalently to Problem 1.

```
#need a long df again
test_preds_long=melt(test_preds, measure.vars=c('knn_preds', 'rf_preds', 'lasso_preds'),
                     variable.name='pred_method', value.name='pred_vals')
```

Visual Evaluation

Before starting to quantify the predictions' precisions, I do a simple visual check by plotting the true values against the predicted values for each method. The code below generates Figure 2.

```
#Facet of all predictions
ggplot(test_preds_long, aes(x=pred_vals, y=charges))+geom_point(size=0.2) +
  geom_abline(size=0.1, color='red') +
  facet_wrap(vars(pred_method), ncol=2, nrow=2)
```

The 45-degree line in the figure shows what a perfect prediction looks like, as in that case, the relationship between the predicted value and actual value is 1:1. It is clearly visible that the Lasso regression and the random forest yield more precise predictions than the KNN regression. It seems that both of the former perform fairly well for small and high values while they have some issues correctly predicting values in the middle of the outcomes' range as a set of values are too far to the left. The random forest seems to do best in this simple test as it fairs better than the Lasso regression in this middle section.

MSE

As a first quantification of the predictions' precision I calculate the MSE. It gives a general overview of their performance.

```
#calculate the MSE for each prediction method and normalize it to lie between 0 and 1
mse=test_preds_long%>%group_by(pred_method)%>%summarize(mse=mean((charges-pred_vals)^2,
                                                                 na.rm=TRUE))%>%
  mutate(pred_method=c('KNN', 'Random Forest', 'Lasso'))
kable(mse, col.names=c('Method', 'MSE'), caption='MSE for all 3 prediction methods')
```

Table 4: MSE for all 3 prediction methods

Method	MSE
KNN	75591202
Random Forest	20684314
Lasso	24933295

As seen in Table 4, the random forest outperforms both other methods in terms of MSE, suggesting that the lack of data is not an issue in this prediction setup and that the visual evaluation already suggested the

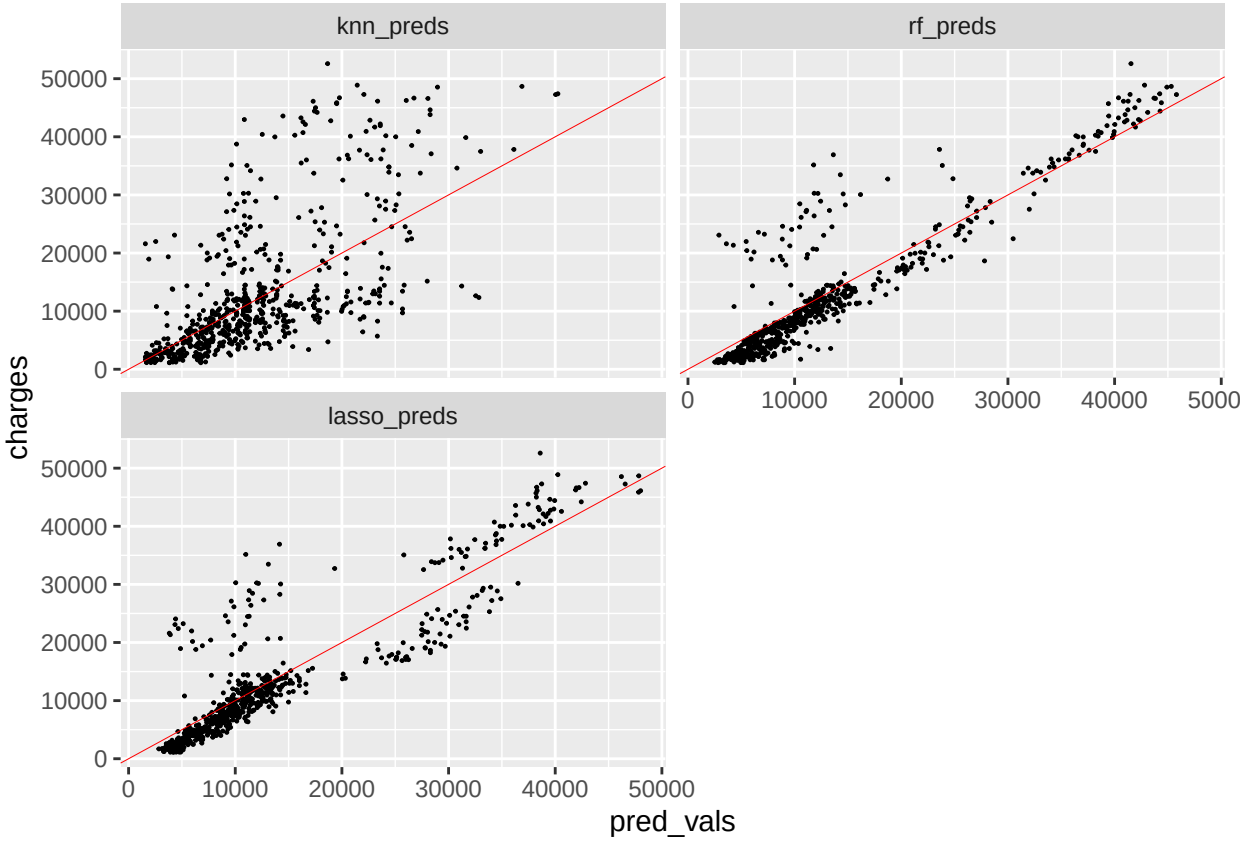


Figure 2: Visual evaluation of prediction methods

correct pattern of performance. However, the KNN predictor has by far the weakest performance. Its MSE is more than three times as high as the MSE of the Lasso regression and almost four times as high as for the Random Forest. One potential explanation is that the KNN estimator does not perform well in this setup because the number of neighbors within a reasonable distance is tiny, and the values experience a high variance, and the information contained in the variables used is not helpful enough. On the other hand, it is also possible that the set of variables chosen by the Lasso are not appropriate to make the best predictions based on KNN and that other interaction terms or variables have a higher predictive power in that case.

MZ test

The Mincer-Zarnowitz test is a simple regression-based test that quantifies whether the prediction-outcome relationship is statistically significantly different from the ideal 1:1 relationship. Hence, it is based on the same idea as the visual approach in Figure 2 but helps to quantify this result further. When regressing the true value on the predicted outcome, a perfect prediction yields an intercept that is zero because if it is not zero, this means that to reach the true value, a constant is added to each prediction - i.e., the prediction is systematically biased. At the same time, the coefficient of the prediction should be one as the prediction should transmit 1:1 to the true value. The regression and significance test is conducted in the function *mz_test*. I test the joint hypothesis that the intercept is not different from 0 while the coefficient of the prediction is not different from 1.

```
mz_test=function(true, preds){
  #This function runs the Mincer-Zarnowitz Test for prediction evaluation
  #first run a simple OLS regression of true values on predictions
  reg=lm(true~preds)
  #get confidence interval
  coefci=coefci(reg, vcov.=NeweyWest)%>%round(5)
  #calculate wald test statistic
  W=t(coef(reg)-c(0,1))%*%solve(NeweyWest(reg))%*%(coef(reg)-c(0,1))
  #and get pvalue of significance tes
  pval=1-pchisq(W,2)%>%round(5)
  #return all three
  return(list(ci=coefci, wald=W, pval=pval))
}

#run Mincer-Zarnowitz test for all predictions
#Lasso
MZ_lasso=mz_test(test_preds$charges, test_preds$lasso_preds)
#Random Forest
MZ_rf=mz_test(test_preds$charges, test_preds$rf_preds)
#KNN
MZ_knn=mz_test(test_preds$charges, test_preds$knn_preds)
```

```
kable(mz_table, caption='MZ Test results')
```

Table 5: MZ Test results

Method	Intercept CI	Predictions CI	Wald Stat	p value
Lasso	-1417.3801, -297.0436	0.98825, 1.06864	11.21471	0.00367
RF	-2043.265, -1060.255	1.05694, 1.10332	48.61656	0
KNN	-1418.6880, 983.5814	0.97831, 1.21646	8.368015	0.01524

All three tests reject the null hypothesis, which implies that the predictions are strongly biased or too noisy to explain the outcome sufficiently well. Taking a look at the individual confidence intervals shows that the CIs for the Lasso and RF predictions' coefficients are close around 1, while the CIs for the intercept

have a large negative tendency. The KNN predictions suggest the opposite picture for this estimator. Here the CI for the intercept includes zero and is very wide in both directions, while the predictions' coefficient varies more around 1. From these observations, I conclude that the MZ test is rejected because of the large variance in the predictions in the case of Lasso and RF, while the KNN is biased systematically - likely due to underfitting of the model because the amount for neighbors is too small.

DM test

Finally, I want to evaluate the relative performances of the different methods. To start with this, I conduct a Diebold-Mariano (DM) test comparing all pairs of predictors. Essentially, the DM test evaluates whether a pre-specified loss function of the prediction differs significantly between two prediction models. This test can be simply implemented in R by calculating the loss for each observation based on the respective prediction and then regressing the difference between the two on a constant. This constant should not be statistically different from zero if the two methods are performing similarly well. Equivalently, one can take a look at the confidence interval of this estimation to quantify the difference between the losses. These are retrieved in the code below and shown in Table 6.

```
#Bergman loss function with phi(x)=x^2
sq_loss=function(true, pred){(true-pred)^2}
#Bergman loss function with phi(x)=-log(x)
log_loss=function(true, pred){loss=log(true) - log(true) + true/pred - 1}
#Bergman loss function with phi(x)=exp(-x)
exp_loss=function(true, pred){exp(-true) - exp(-pred) + exp(-pred) *(true-pred)}

dm_test=function(true, pred1, pred2, loss='squared'){
  #this function runs a Diebold-Mariano Test using the loss function specified in
  #loss argument
  if(loss=='squared'){
    d=sq_loss(true, pred1)-sq_loss(true, pred2)
    reg=lm(d~1)
  }
  #if loss=exp use loss function with exp(-x)
  else if(loss=='exp'){
    d=exp_loss(true, pred1)-exp_loss(true, pred2)
    reg=lm(d~1)
  }
  #if loss=log use QLIKE loss function which arises from using phi(x)=-log(x)
  else if(loss=='log'){
    d=log_loss(true, pred1)-log_loss(true, pred2)
    reg=lm(d~1)
  }
  else{
    print('Specify loss to be squared, log, or exp!')
  }
  #then run significane test with heteroskedastiticy robust std errors
  sig_test=coefci(reg, vcov.=NeweyWest)
  return(list(mean_d=mean(d), sigtest=sig_test))
}

make_compare_frame=function(comp1, comp2, comp3){
  #function binding the dm test results for all three comparisons I make together into
  #a df for nice presentation of results in Rmd
  #WARNING: very specified for this use case here! Order of DM test results supplied
  #must always be Lasso vs RF, Lasso vs KNN and then RF vs KNN for correct presentation
  lowerbounds=c(comp1$sigtest[1], comp2$sigtest[1], comp3$sigtest[1])
```

```
  upperbounds=c(comp1$sigtest[2], comp2$sigtest[2], comp3$sigtest[2])
  comparison=c('Lasso vs RF', 'Lasso vs KNN', 'RF vs KNN')
  return(data.frame(Methods=comparison, Lower=lowerbounds, Upper=upperbounds))
}

#Diebold-Mariano Tests with squared loss function
#start with Lasso vs RF
dm_lasso_rf=dm_test(test_preds$charges, test_preds$lasso_preds, test_preds$rf_preds)
#Lasso vs KNN
dm_lasso_knn=dm_test(test_preds$charges, test_preds$lasso_preds, test_preds$knn_preds)
#KNN vs RF
dm_rf_knn=dm_test(test_preds$charges, test_preds$rf_preds, test_preds$knn_preds)
#bind everything together into a nice table
compare_sq=make_compare_frame(dm_lasso_rf, dm_lasso_knn, dm_rf_knn)
kable(compare_sq, caption='DM test results with squared loss function')
```

Table 6: DM test results with squared loss function

Methods	Lower	Upper
Lasso vs RF	2538971	5958989
Lasso vs KNN	-62187939	-39127875
RF vs KNN	-66538871	-43274904

The code below produces Table 7 which shows the results of the DM test when using a log-score function (QLIKE) as the loss function for comparison.

```
#Diebold-Mariano Tests with QLIKE loss function
#start with Lasso vs RF
dm_lasso_rf=dm_test(test_preds$charges, test_preds$lasso_preds, test_preds$rf_preds,
                    loss='log')
#Lasso vs KNN
dm_lasso_knn=dm_test(test_preds$charges, test_preds$lasso_preds, test_preds$knn_preds,
                     loss='log')
#KNN vs RF
dm_rf_knn=dm_test(test_preds$charges, test_preds$rf_preds, test_preds$knn_preds,
                  loss='log')
#again emply make_compare_frame
#WATCH OUT FOR ORDER!
compare_sq=make_compare_frame(dm_lasso_rf, dm_lasso_knn, dm_rf_knn)
kable(compare_sq, caption='DM test results with log loss function')
```

Table 7: DM test results with log loss function

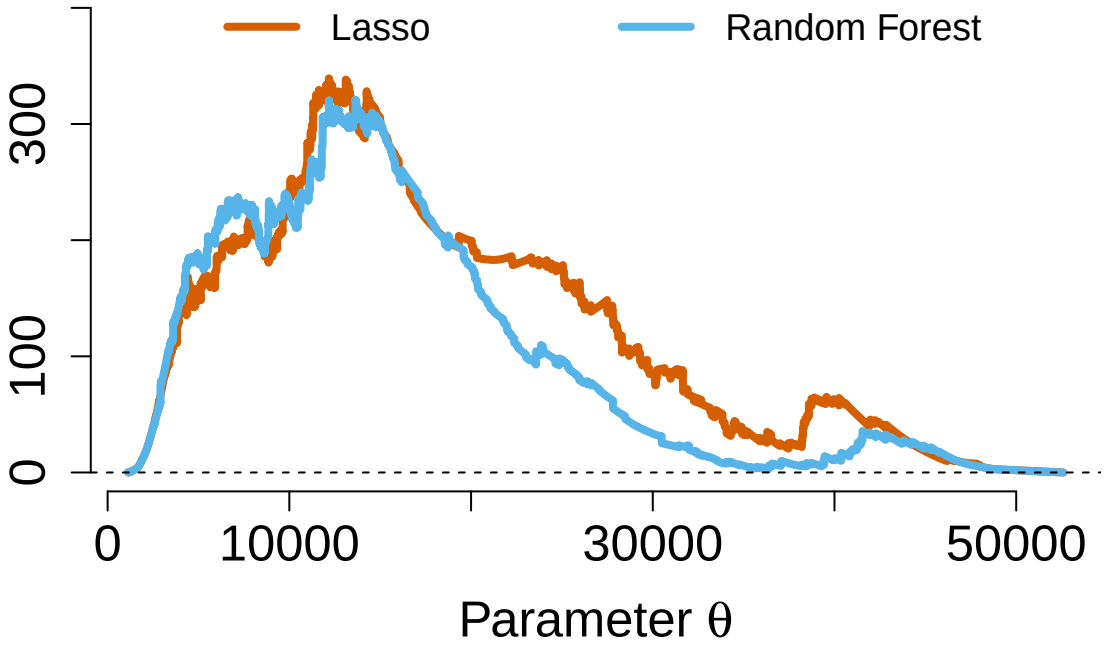
Methods	Lower	Upper
Lasso vs RF	-0.0024423	0.0335678
Lasso vs KNN	-0.2286875	-0.1101807
RF vs KNN	-0.2451422	-0.1248516

The qualitative results are the same for both loss functions. The KNN predictions are still performing the worst as the CI in both comparisons is entirely below zero. As I am looking at the difference between Lasso/RF predictions minus the KNN predictions, this implies that the loss function for the KNN predictions

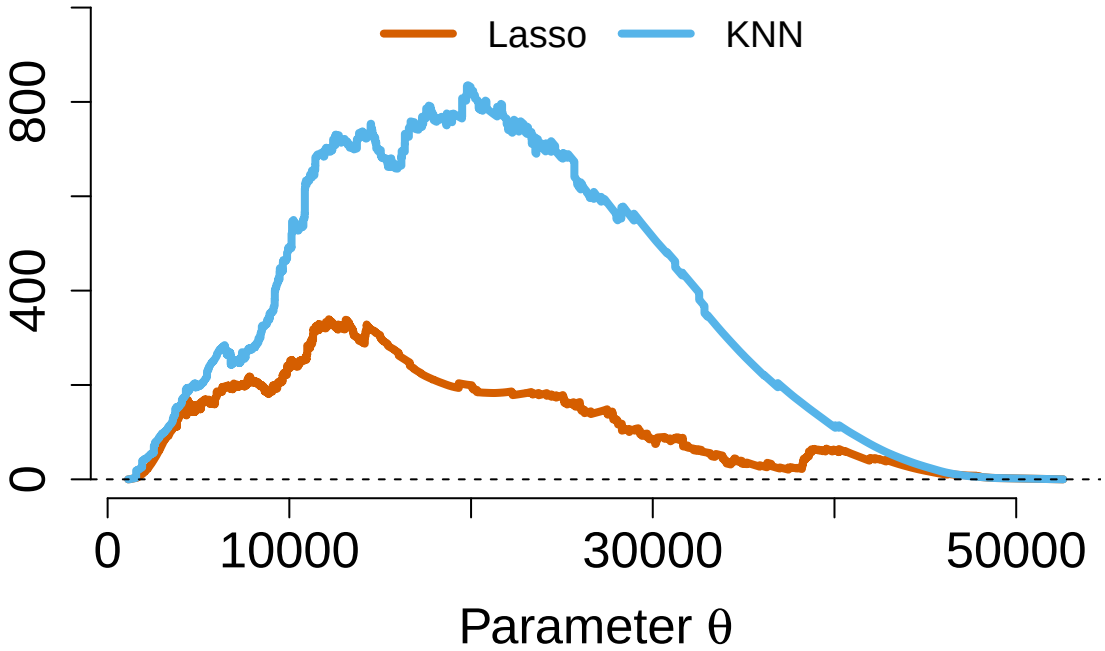
is consistently above the other predictions' losses. Meanwhile, the CI, when comparing the Lasso to the RF, is stretched around zero, prohibiting a clear stance on the relative performance. Let me note, however, that the lower bound is very close to zero, such that this also might be the result of some noise.

Murphy Diagrams

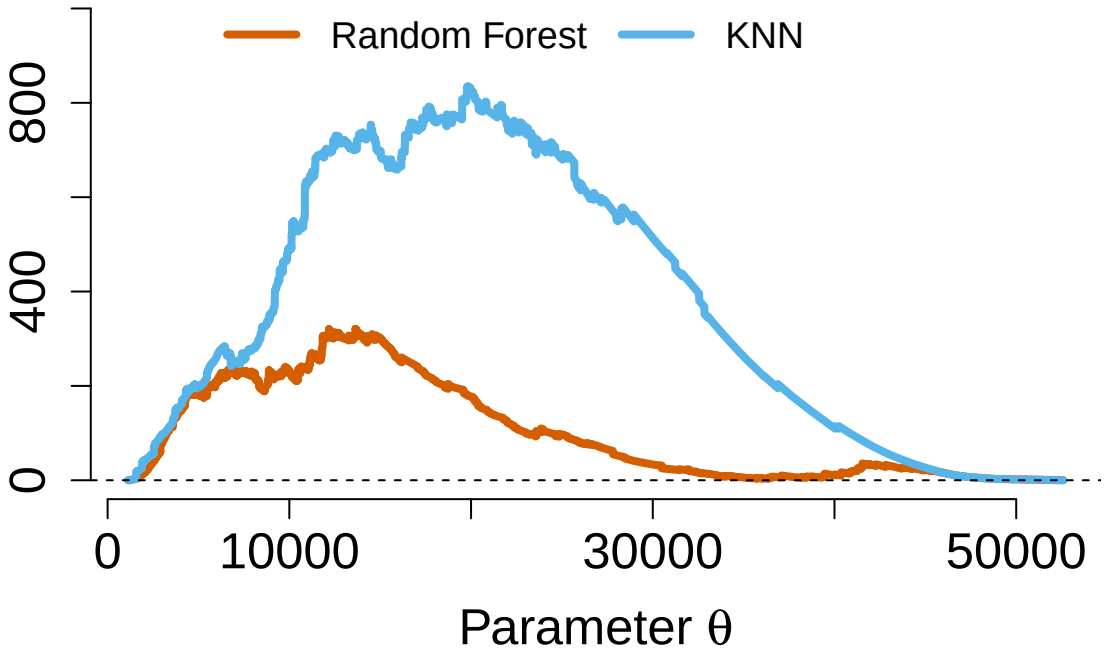
```
#Lasso vs RF
murphydiagram(test_preds$lasso_preds, test_preds$rf_preds, test_preds$charges,
              labels=c('Lasso', 'Random Forest'))
```




```
#Lasso vs KNN
murphydiagram(test_preds$lasso_preds, test_preds$knn_preds, y=test_preds$charges,
              labels=c('Lasso', 'KNN'))
```




```
#RF vs KNN
murphydiagram(test_preds$rf_preds, test_preds$knn_preds, y=test_preds$charges,
              labels=c('Random Forest', 'KNN'))
```



The Murphy diagrams confirm the comparison results from above. The random forest does not clearly outperform the random forest for all parameter values but fairs better for higher values of θ. Meanwhile, comparing the two predictors to KNN, it is not even close! Both clearly outperform the KNN regression for any parameter value.

Concluding Remarks

Overall it has become evident that in this setup, the random forest performs the best. However, I have to note that all predictions are very noisy, with MSEs in the millions and MZ tests failing, suggesting that the data used contains too little information - most likely due to the small size of the dataset - to make robust predictions.

Problem 3

Using the same data as in Problem 2, the goal is to estimate the 10% quantile now.

a) Data splitting

As the data used is the same as in Problem 2, I simply use the same test and training data as before.

b)

Now the task is no longer to predict the conditional mean but rather the conditional quantile. To do so, I employ the following two methods:

1. Quantile Regression
2. Quantile Random Forest

Quantile Regression

The quantile regression approach developed by Koenker does no longer minimize the squared deviation but rather the so-called *check* function. One can think of this approach as a weighted approach to regression. The estimate for the τ quantile q to minimize the check function. The check function penalizes each deviation above q (overprediction) with a weight of 1-tau, while each deviation below q (underprediction) is penalized

with tau. In the case at hand, the goal is to minimize the 10% percentile of the distribution. One can show that the minimizing argument q to the check function is indeed the 10% quantile of the distribution (given that all other assumptions hold) and that it can be approximated by linear regression. The quantile regression is implemented in the code below, and the resulting predictions are saved in a dataframe in the same manner as in all previous problems.

```
#fit quantile regression model to training data
qr_10p=rq(charges~., tau=0.1, data=train_dat)
#then predict using test data and save in predictions df
test_preds$qr10=predict(qr_10p, newdata=test_dat)
```

Quantile Regression Forest

Athey et al. (2018) have introduced the Generalized Random Forest estimator and predictor, which allows using forest-based regression to estimate any moment equation one desires - of course, under some assumptions on the moment equation, etc. This framework allows to rephrase the quantile prediction problem into a moment equation and then estimating it using the GRF framework. The authors also provide the respective functions in the R package *grf*. The code below uses the *quantile_forest* function provided to estimate the 10% quantile.

```
#create model matrix for observables for training and test data
x_train=model.matrix(charges~., data=train_dat)
y_train=train_dat$charges
x_test=model.matrix(charges~., data=test_dat)
#then fit a generalized random forest with quantile loss function
#aka quantile random forest
q_rf=grf::quantile_forest(x_train, y_train, quantiles=0.1)
#predict the outcome using quantile random forest
test_preds$qrf10=as.vector(predict(q_rf, new_x=x_test)[[1]])
```

c) & d) Evaluation and Discussion

To get a first feeling of the performance of the two prediction methods, I calculate the unconditional coverage of the two predictors. If the predicted value indeed represents the 10% quantile of the distribution, the percentage of individuals with an outcome below or equal to the predicted value should be 10%. The coverage is calculated in the code below and shown in Table 8.

```
#Unconditional coverage
#quantile regression
qr_un_cover=mean(test_preds$charges<=test_preds$qr10)
#quantile random forest
qrf_un_cover=mean(test_preds$charges<=test_preds$qrf10)
#bin in dataframe for table
un_cover=data.frame(Method=c('QR', 'QRF'),
                    'Unconditional Coverage'=c(qr_un_cover, qrf_un_cover))
kable(un_cover, caption='Unconditional Coverage of 10% Quantile Predictions')
```

Table 8: Unconditional Coverage of 10% Quantile Predictions

Method	Unconditional.Coverage
QR	0.0881913
QRF	0.3318386

The quantile regression performs very well, with its unconditional coverage roughly at 10%. However, this

simple metric suggests that the quantile forest is quite off - similarly to the previous discussions, the issue is likely related to data availability that prohibits the random forest from generalizing enough. In this scenario, the parametric setup seems to have an advantage over the data-driven format of the random forest.

MZ Quantile Test

The Mincer-Zanrowitz test can also be conducted for quantile prediction. The OLS regression in the original MZ test is simply replaced by a quantile regression of the quantile of interest. The interpretation, however, is similar. In the ideal case, the predicted value fully explains the 10% conditional quantile of the distribution, and thus, the relationship is described by a 45-degree line. Additionally to the MZ test, the function *mz_quant_test* also creates a plot of the true values against the predictions with the 45-degree line and the fitted quantile line added to it. Figures 3 and 4 display the plot for both methods, respectively, while Table 9 displays the MZ test results.

```
mz_quant_test=function(true, preds, tau=0.1){
  #this function runs a Mincer-Zernowitz test for quantile prediction,
  #i.e. using quantile regression of true values on prediction to determine whether
  #auto-calibration is fulfilled
  qreg=quantreg::rq(true~preds, tau=tau)
  #get summary with bootstrapped std errors
  estim=summary(qreg, covariance=TRUE, se='boot')
  #make Wald-test of joint significance,
  #intercept should be 0 and prediction coefficient should be 1
  #solve(matrix) returns inverse of matrix
  wald=t(coef(qreg)-c(0, 1))%*%solve(estim$cov)%*%(coef(qreg)-c(0, 1))
  #then get p-values
  pval=1-pchisq(wald, 2)
  #and create a plot
  p <- ggplot(data=data.frame(y=true, preds=preds), aes(x=preds, y=true)) +
    geom_abline(slope=1, intercept=0, col="black") +
    geom_point(color='darkgreen', size=0.2) +
    geom_quantile(quantiles=tau, color="red")

  return(list(estimate=estim, wald=as.numeric(wald), pvals=as.numeric(pval), plot=p))
}
#quantile regression
mz_qr=mz_quant_test(test_preds$charges, test_preds$qr10)
#quantile random forest
mz_qrf=mz_quant_test(test_preds$charges, test_preds$qrf10)

#create table with wald test statistics
mzq_wald_table=data.frame(Method=c('QR', 'QRF'), Wald=c(mz_qr$wald, mz_qrf$wald),
                          'p Value'=c(mz_qr$pval, mz_qrf$pval))
kable(mzq_wald_table, caption='MZ test results for QR and QRF')
```

Table 9: MZ test results for QR and QRF

Method	Wald	p.Value
QR	5.011795	0.0816023
QRF	1715.347460	0.0000000

```
mz_qr$plot
```

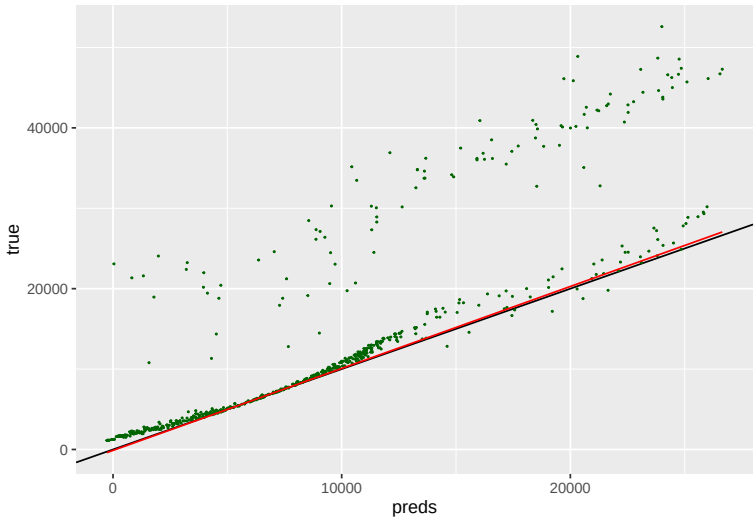


Figure 3: MZ Test Plot QR

```
mz_qrf$plot
```

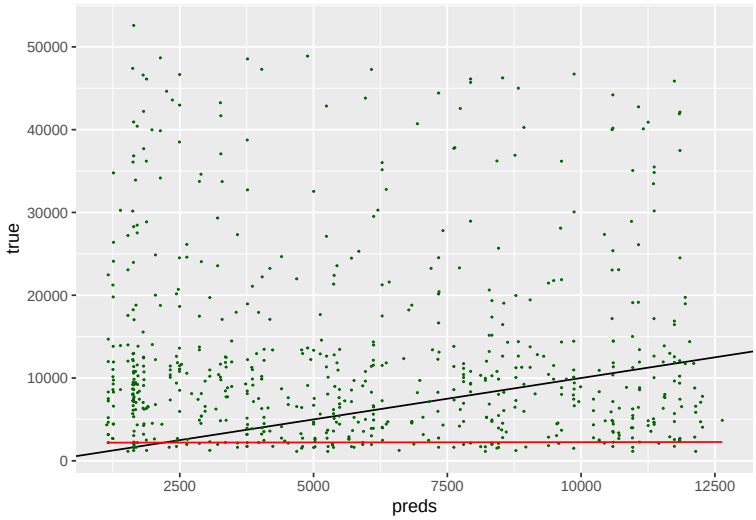


Figure 4: MZ Tets Plot QRF

At the 5% level, the MZ test cannot reject the null when using the predictions produced by the quantile regression. This underlines the strong performance of the quantile regression and can also be seen in Figure 3, which plots the true values against the predicted conditional quantiles. The red line depicts the fitted quantile. Apparently, the quantile regression can produce precise and unbiased results, at least for a certain range of values. The picture is completely different for the quantile random forest. Its bad performance in the unconditional coverage is confirmed by the MZ test and the visual depiction in Figure 4. It becomes evident how spurious the predictions of the quantile random forest are. As already mentioned, I assume that this is due to the lack of data such that the forest picks up random noise in the training data and interprets them as actual relationships.

Quantile Loss Score and DM test

To evaluate the relative performance of the two methods more closely, I inspect the quantile loss score the two produce and perform a Diebold-Mariano test on the difference in these scores. The intuition of the test is the same as before, the only difference being that the score function differs. Note that I am regressing the difference between the QR score and the QRF score, i.e., a negative sign on the CI boundaries implies that the QRF performs worse.

```
#Quantile Loss Score
#make a Diebold-Mariano test to evaluate which model performs relatively better
#for interpretation see pdf solutions file
#quantile loss score
qloss_score <- function(q_pred, true, tau){ (1-tau)*(q_pred-true)*(q_pred>true)
              + tau*(true-q_pred)*(true>=q_pred) }
qr_score=qloss_score(test_preds$qr10, test_preds$charges, tau=0.1)
qrf_score=qloss_score(test_preds$qrf10, test_preds$charges, tau=0.1)
score_diff=qr_score-qrf_score

dm_test_quant=lm(score_diff~1)%>%coefci(vcov.=NeweyWest)

kable(dm_test_quant, caption='DM Test Results')
```

Table 10: DM Test Results

	2.5 %	97.5 %
(Intercept)	-473.8272	-382.8159

Indeed, my expectations and prior evaluations are confirmed as the DM test CI boundaries are both negative, suggesting that the quantile loss of the QR is significantly lower than the one of the QRF.

Prediction Intervals

As the last step, I investigate the overall prediction uncertainty of the two approaches by looking at the prediction interval. By predicting the 2.5% and the 97.5% quantile, I construct a predicted interval in which 95% of all values should be and check how many observations are indeed falling into their respective interval. This coverage should be roughly around 95% if the prediction method can estimate the correct quantiles.

```
#Prediction Interval
#let's summarize prediction uncertainty with prediction interval with alpha=0.05
#quantile regression
qr_low=quantreg::rq(charges~., data=train_dat, tau=0.025)
pi_low=predict(qr_low, newdata=test_dat)
qr_high=quantreg::rq(charges~., data=train_dat, tau=0.975)
pi_up=predict(qr_high, newdata=test_dat)
#how many percent of true values are in PI?
in_pi=(pi_low<=test_preds$charges)&(test_preds$charges<=pi_up)
#pct in
pct_in_pi=mean(in_pi)
#94.91%!

#quantile random forest
#in quantile rf function can fit model to multiple quantiles at once
qrf_both=grf::quantile_forest(X=x_train, Y=y_train, quantiles=c(0.025, 0.975))
#predict both quantiles at once
qrf_both_pred=predict(qrf_both, new_x=x_test)[[1]]
```

```
#first column is lower, second one is upper quantile
#check coverage of PI
in_pi_qrf=(qrf_both_pred[, 1]<=test_preds$charges)&
  (test_preds$charges<=qrf_both_pred[, 2])
#percentage coverage
pct_in_pi_qrf=mean(in_pi_qrf)
#only 67.9!
kable(data.frame(Method=c('QR', 'QRF'), 'PI Coverage'=c(pct_in_pi, pct_in_pi_qrf)),
      caption='Prediction Interval Coverage')
```

Table 11: Prediction Interval Coverage

Method	PI.Coverage
QR	0.9491779
QRF	0.6860987

Again the QR performs very well with an almost exact coverage of 94.91%, while the quantile random forest fails to reach the correct coverage. However, one must keep in mind that this metric does not specify how precise the estimations are - the QR's prediction intervals could be very wide, making it very likely that the true values fall between the two boundaries. To get a more precise picture of the tradeoff between this coverage and the precision (or: sharpness) of the two estimators, I calculate the interval score by Gneiting and Katzfuss (2014). However, given the prior results, I expect that the quantile regression will outperform the QRF in this metric as well because it has done so so far in metrics dedicated to sharpness and coverage.

```
int_score=function(y, r, l, alpha){
  #supply the true values, upper and lower bound of PIs as vectors and calculate
  #the interval score for a given alpha
  scores=(r-l)+2/alpha*(l-y)*as.numeric(y<l)+2/alpha*(y-r)*as.numeric(y>r)
  #sum all scores together to get interval score
  is=sum(scores)
  return(is)
}
#calculate IS for both predictions
is_qr=int_score(test_preds$charges, pi_low, pi_up, alpha=0.05)
is_qrf=int_score(test_preds$charges, qrf_both_pred[, 1], qrf_both_pred[, 2], alpha=0.05)
#table
kable(data.frame(Method=c('QR', 'QRF'), 'Interval Score'=c(is_qr, is_qrf)),
      caption='Interval Score')
```

Table 12: Interval Score

Method	Interval.Score
QR	545339301
QRF	1039102795

As expected, the quantile regression results in a smaller interval score than the quantile regression forest, underlining that it does not perform so strongly in coverage because of its imprecision but can handle both.

Concluding Remarks

The quantile random forest seems to have quite some trouble in predicting the conditional quantiles, while the quantile regression performs very well. The parametric approach seems to have an advantage in this setup

again - contrary to when predicting the conditional mean as in Problem 2. Estimating conditional quantiles instead of the conditional mean can have multiple advantages depending on the setting and research question. First of all, many research questions nowadays are concerned with social and economic inequality, and quantiles can help to understand better how income, education, or other variables of interest are distributed conditional on individuals' characteristics. Moreover, quantiles are indifferent to monotonic transformations of the data, which can make them useful in settings in which such transformations are necessary (e.g., when interested in logarithmic values to explore percentage changes). Finally, quantile predictions are very robust to outliers, and in case one is not interested in their role, this can be a helpful feature. However, one has to keep in mind that in the quantile regression framework using predictions to estimate treatment effects can be dangerous, as one predicts the quantile before and after treatment, but the individuals that make up this quantile can be completely different, hence only showing how treatment shifts the distribution.