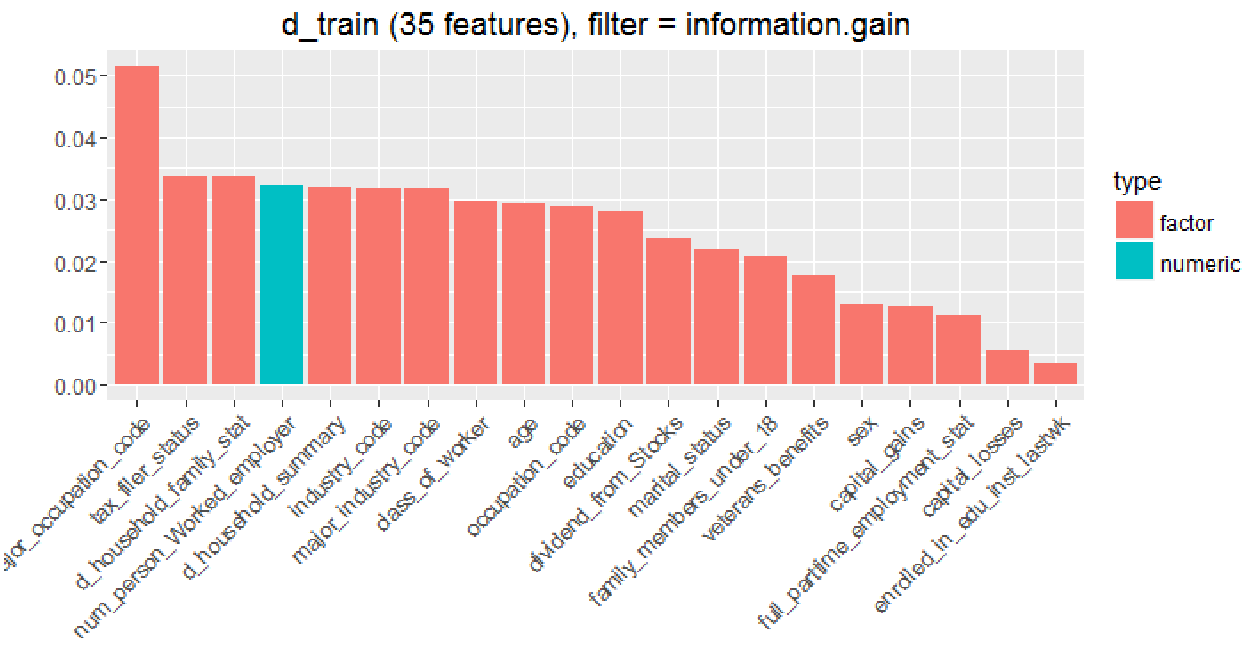
```
#get variable importance chart
> var_imp <- generateFilterValuesData(train.task, method = c("information.gain"))
> plotFilterValues(var_imp,feat.type.cols = TRUE)
```



In simple words, you can understand that the variable `major_occupation_code`would provide highest information to the model followed by other variables in descending order. This chart is deduced using a tree algorithm, where at every split, the information is calculated using reduction in entropy (homogeneity). Let's keep this knowledge safe, we might use it in coming steps.

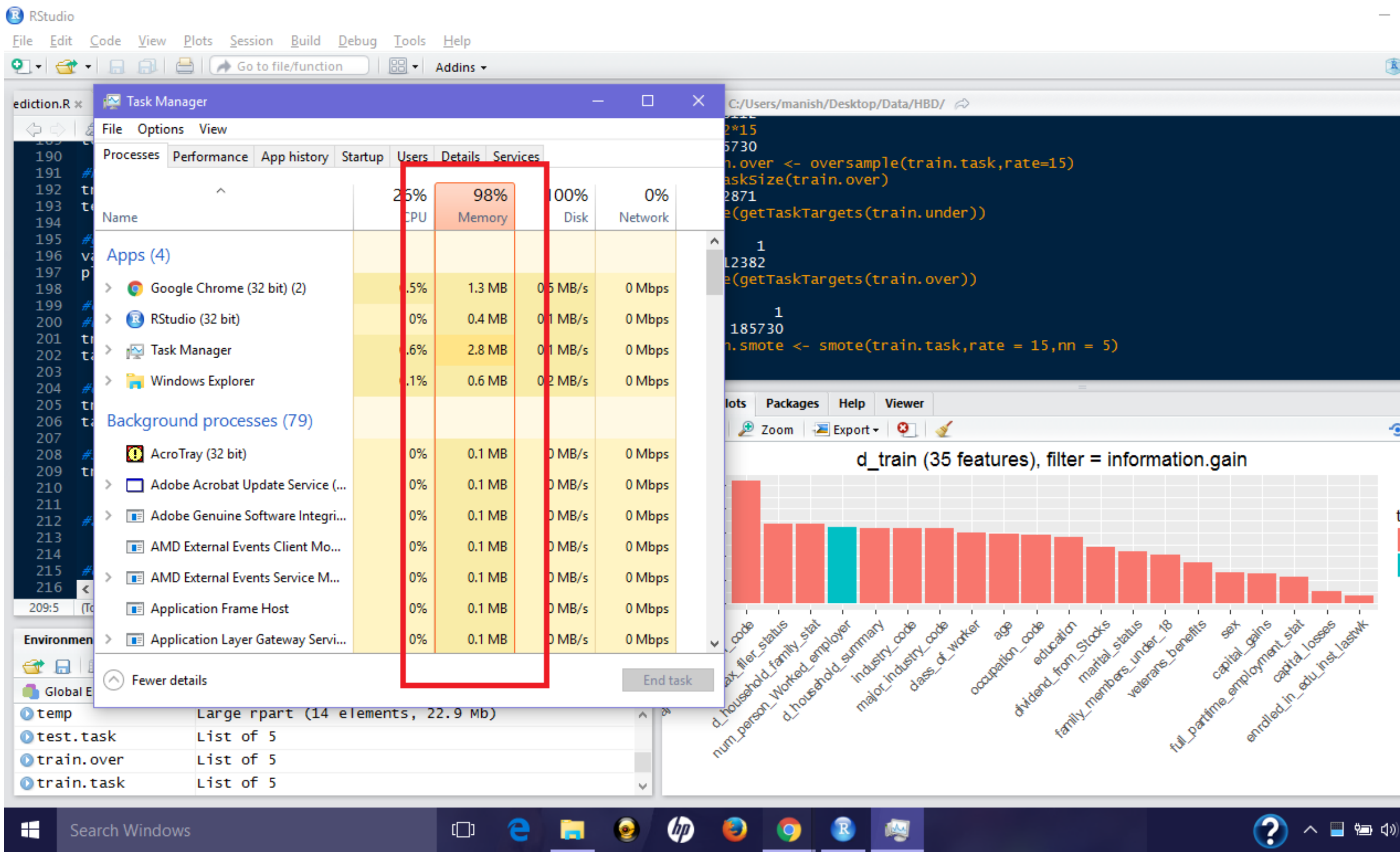
Now, we'll try to make our data balanced using various techniques such as over sampling, undersampling and SMOTE. In SMOTE, the algorithm looks at n nearest neighbors, measures the distance between them and introduces a new observation at the center of n observations. While proceeding, we must keep in mind that these techniques have their own drawbacks such as:

- undersampling leads to loss of information
- oversampling leads to overestimation of minority class

Being your first project(hopefully), we should try all techniques and experience how it affects.

```
#undersampling
> train.under <- undersample(train.task,rate = 0.1) #keep only 10% of majority class
> table(getTaskTargets(train.under))

#oversampling
> train.over <- oversample(train.task,rate=15) #make minority class 15 times
> table(getTaskTargets(train.over))

#SMOTE
> train.smote <- smote(train.task,rate = 15,nn = 5)
```

Looks like, my machine gave up at these SMOTE parameters. It's been over 50 minutes and this code hasn't executed. Look at the havoc this is creating in my poor machine:



```
2*15
5730
n.over <- oversample(train.task,rate=15)
askSize(train.over)
2871
e(getTaskTargets(train.under))

      1
12382
e(getTaskTargets(train.over))

        1
 185730
n.smote <- smote(train.task,rate = 15,nn = 5)
```

Name	25% CPU	98% Memory	100% Disk	0% Network
Apps (4)				
Google Chrome (32 bit) (2)	.5%	1.3 MB	0.5 MB/s	0 Mbps
RStudio (32 bit)	0%	0.4 MB	0.1 MB/s	0 Mbps
Task Manager	.6%	2.8 MB	0.1 MB/s	0 Mbps
Windows Explorer	.1%	0.6 MB	0.2 MB/s	0 Mbps
Background processes (79)				
AcroTray (32 bit)	0%	0.1 MB	0 MB/s	0 Mbps
Adobe Acrobat Update Service (...	0%	0.1 MB	0 MB/s	0 Mbps
Adobe Genuine Software Integri...	0%	0.1 MB	0 MB/s	0 Mbps
AMD External Events Client Mo...	0%	0.1 MB	0 MB/s	0 Mbps
AMD External Events Service M...	0%	0.1 MB	0 MB/s	0 Mbps
Application Frame Host	0%	0.1 MB	0 MB/s	0 Mbps
Application Layer Gateway Servi...	0%	0.1 MB	0 MB/s	0 Mbps

```
temp         Large rpart (14 elements, 22.9 Mb)
test.task    List of 5
train.over   List of 5
train.task   List of 5
```

RStudio
File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
Addins
Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task
C:/Users/manish/Desktop/Data/HBD/
Plots Packages Help Viewer
Zoom
Export
d_train (35 features), filter = information.gain
Environment
Global Environment
Search Windows

While working on such data sets, it's important for you to learn the ways to hop such obstacles. Let's modify the parameters and run it again:

```
> system.time(
  train.smote <- smote(train.task,rate = 10,nn = 3)
  )
Warning messages:
# 1: In is.factor(x) :
# Reached total allocation of 8084Mb: see help(memory.size)
# 2: In is.factor(x) :
# Reached total allocation of 8084Mb: see help(memory.size)
# user system elapsed
# 81.95 21.86 184.56
> table(getTaskTargets(train.smote))
```

It did run with some warning messages. We can ignore them for now. Let's now look at the available algorithms we can use to solve this problem.

```
#lets see which algorithms are available
> listLearners("classif","twoclass")[c("class","package")]
```

We'll start with naive Bayes, an algorithms based on bayes theorem. In case of high dimensional data like text-mining, naive Bayes tends to do wonders in accuracy. It works on categorical data. In case of numeric variables, a normal distribution is considered for these variables and a mean and standard deviation is calculated. Then, using some standard z-table calculations probabilities can be estimated for each of your continuous variables to make the naive Bayes classifier.

We'll use naive Bayes on all 4 data sets (imbalanced, oversample, undersample and SMOTE) and compare the prediction accuracy using cross validation.

```
#naive Bayes
> naive_learner <- makeLearner("classif.naiveBayes",predict.type = "response")
> naive_learner$par.vals <- list(laplace = 1)

#10fold CV - stratified
> folds <- makeResampleDesc("CV",iters=10,stratify = TRUE)

#cross validation function
> fun_cv <- function(a){
    crv_val <- resample(naive_learner,a,folds,measures = list(acc,tpr,tnr,fpr,fp,fn))
    crv_val$aggr
}

> fun_cv (train.task)
# acc.test.mean tpr.test.mean tnr.test.mean fpr.test.mean
# 0.7337249       0.8954134     0.7230270    0.2769730
```

```
> fun_cv(train.under)
# acc.test.mean tpr.test.mean tnr.test.mean fpr.test.mean
# 0.7637315      0.9126978     0.6651696     0.3348304

> fun_cv(train.over)
# acc.test.mean tpr.test.mean tnr.test.mean fpr.test.mean
#   0.7861459     0.9145749     0.6586852    0.3413148

> fun_cv(train.smote)
# acc.test.mean tpr.test.mean tnr.test.mean fpr.test.mean
#   0.8562135     0.9168955    0.8160638     0.1839362
```

This package names cross validated results are `test.mean`. After comparing, we see that `train.smote` gives the highest true positive rate and true negative rate. Hence, we learn that SMOTE technique outperforms the other two sampling methods.

Now, let’s build our model SMOTE data and check our final prediction accuracy.

```
#train and predict
> nB_model <- train(naive_learner, train.smote)
> nB_predict <- predict(nB_model,test.task)

#evaluate
> nB_prediction <- nB_predict$data$response
> dCM <- confusionMatrix(d_test$income_level,nB_prediction)
# Accuracy : 0.8174
# Sensitivity : 0.9862
# Specificity : 0.2299

#calculate F measure
> precision <- dCM$byClass['Pos Pred Value']
> recall <- dCM$byClass['Sensitivity']

> f_measure <- 2*((precision*recall)/(precision+recall))
> f_measure
```

The function confusionMatrix is taken from `library(caret)`. This naive Bayes model predicts 98% of the majority class correctly, but disappoints at minority class prediction (~23%). Let us not get hopeless and try more techniques to improve our accuracy. Remember, the more you hustle, better you get!

Let’s use xgboost algorithm and try to improve our model. We’ll do 5 fold cross validation and 5 round random search for parameter tuning. Finally, we’ll build the model using the best tuned parameters.

```
#xgboost
> set.seed(2002)
> xgb_learner <- makeLearner("classif.xgboost",predict.type = "response")
> xgb_learner$par.vals <- list(
                    objective = "binary:logistic",
                    eval_metric = "error",
                    nrounds = 150,
                    print.every.n = 50
)

#define hyperparameters for tuning
> xg_ps <- makeParamSet(
               makeIntegerParam("max_depth",lower=3,upper=10),
               makeNumericParam("lambda",lower=0.05,upper=0.5),
               makeNumericParam("eta", lower = 0.01, upper = 0.5),
               makeNumericParam("subsample", lower = 0.50, upper = 1),
               makeNumericParam("min_child_weight",lower=2,upper=10),
               makeNumericParam("colsample_bytree",lower = 0.50,upper = 0.80)
)

#define search function
> rancontrol <- makeTuneControlRandom(maxit = 5L) #do 5 iterations

#5 fold cross validation
> set_cv <- makeResampleDesc("CV",iters = 5L,stratify = TRUE)

#tune parameters
> xgb_tune <- tuneParams(learner = xgb_learner, task = train.task, resampling = set_cv, measures = 
list(acc,tpr,tnr,fpr,fp,fn), par.set = xg_ps, control = rancontrol)
# Tune result:
# Op. pars: max_depth=3; lambda=0.221; eta=0.161; subsample=0.698; min_child_weight=7.67; 
colsample_bytree=0.642
# acc.test.mean=0.948,tpr.test.mean=0.989,tnr.test.mean=0.324,fpr.test.mean=0.676
```

Now, we can use these parameter for modeling using `xgb_tune$x` which contains the best tuned parameters.

```
#set optimal parameters
> xgb_new <- setHyperPars(learner = xgb_learner, par.vals = xgb_tune$x)

#train model
> xgmodel <- train(xgb_new, train.task)

#test model
> predict.xg <- predict(xgmodel, test.task)

#make prediction
> xg_prediction <- predict.xg$data$response

#make confusion matrix
> xg_confused <- confusionMatrix(d_test$income_level,xg_prediction)
Accuracy : 0.948
Sensitivity : 0.9574
Specificity : 0.6585

> precision <- xg_confused$byClass['Pos Pred Value']
> recall <- xg_confused$byClass['Sensitivity']

> f_measure <- 2*((precision*recall)/(precision+recall))
> f_measure
#0.9726374
```

As we can see, xgboost has outperformed naive Bayes model’s accuracy (as expected!). Can we further improve ?

Until now, we’ve used all the variables in the data. Shall we try using the important ones? Consider it your homework. Let me provide you hint to do this:

```
#top 20 features
> filtered.data <- filterFeatures(train.task,method = "information.gain",abs = 20)
#train
> xgb_boost <- train(xgb_new,filtered.data)
```

After this, follow the same steps as above for predictions and evaluation. Tell me your understanding in comments below.

Until now, our model has been making label predictions. The threshold used for making these predictions in 0.5 as seen by:

```
> predict.xg$threshold
[[1]] 0.5
```

Due to imbalanced nature of the data, the threshold of 0.5 will always favor the majority class since the probability of a class 1 is quite low. Now, we'll try a new technique:

- •Instead of labels, we'll predict probabilities
- •Plot and study the AUC curve
- •Adjust the threshold for better prediction

We'll continue using xgboost for this stunt. To do this, we need to change the `predict.type` parameter while defining learner.

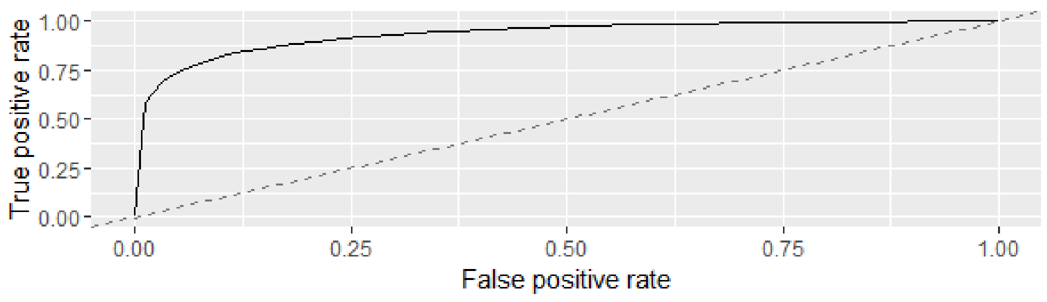
```
#xgboost AUC
> xgb_prob <- setPredictType(learner = xgb_new,predict.type = "prob")

#train model
> xgmodel_prob <- train(xgb_prob,train.task)

#predict
> predict.xgprob <- predict(xgmodel_prob,test.task)
```

Now, let's look at the probability table thus created:

```
#predicted probabilities
> predict.xgprob$data[1:10,]
```

Since, we have obtained the class probabilities, let's create an AUC curve and determine the basis to modify prediction threshold.

```
> df <- generateThreshVsPerfData(predict.xgprob,measures = list(fpr,tpr))
> plotROCCurves(df)
```



AUC is a measure of true positive rate and false positive rate. We aim to reach as close to top left corner as possible. Therefore, we should aim to reduce the threshold so that the false positive rate can be reduced.

```
#set threshold as 0.4
> pred2 <- setThreshold(predict.xgprob,0.4)
> confusionMatrix(d_test$income_level,pred2$data$response)
# Sensitivity : 0.9512
# Specificity : 0.7228
```

With 0.4 threshold, our model returned better predictions than our previous xgboost model at 0.5 threshold. Thus, you can see that setting threshold using AUC curve actually affect our model performance. Let's give one more try.

```
> pred3 <- setThreshold(predict.xgprob,0.30)
> confusionMatrix(d_test$income_level,pred3$data$response)
#Accuracy : 0.944
# Sensitivity : 0.9458
# Specificity : 0.7771
```

This model has outperformed all our models i.e. in other words, this is the best model because 77% of the minority classes have been predicted correctly.

Similarly, you can try and test other threshold values to check if your model improves. In this xgboost model, there is a lot you can do such as:

- Increase the number of rounds
- Do 10 fold CV
- Increase repetitions in random search
- Build models on other 3 data sets and see which one is better

Apart from the methods listed above, you can also assign class weights such that the algorithm pays more attention while classifying the class with higher weight. I leave this part as **homework** to you. Run the code below and update me if you model surpassed our previous xgboost prediction. Use SVM in homework. An important tip: The code below might take longer than expected to run, therefore close all other applications.

```
#use SVM
> getParamSet("classif.svm")
> svm_learner <- makeLearner("classif.svm",predict.type = "response")
> svm_learner$par.vals<- list(class.weights = c("0"=1,"1"=10),kernel="radial")
> svm_param <- makeParamSet(
            makeIntegerParam("cost",lower = 10^-1,upper = 10^2),
            makeIntegerParam("gamma",lower= 0.5,upper = 2)
)

#random search
> set_search <- makeTuneControlRandom(maxit = 5L) #5 times

#cross validation #10L seem to take forever
> set_cv <- makeResampleDesc("CV",iters=5L,stratify = TRUE)

#tune Params
> svm_tune <- tuneParams(learner = svm_learner,task = train.task,measures = list(acc,tpr,tnr,fpr,fp,fn),
par.set = svm_param,control = set_search,resampling = set_cv)
)

#set hyperparameters
>svm_new <- setHyperPars(learner = svm_learner, par.vals = svm_tune$x)

#train model
>svm_model <- train(svm_new,train.task)

#test model
>predict_svm <- predict(svm_model,test.task)

> confusionMatrix(d_test$income_level,predict_svm$data$response)
```