In [112]:
```
# Write program that picks best predictors
```

In [113]:
```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.metrics import accuracy_score
```

In [114]:
```
X_train, X_test, y_train, y_test = preprocess(df,2)
```

In [115]:
```
#Create list of column names
feat_labels = X_train.columns.tolist()
```

In [116]:
```
# Create a random forest classifier
clf = RandomForestClassifier()

# Train the classifier
clf.fit(X_train, y_train)

importance_list = []
# Print the name and gini importance of each feature
for feature in zip(feat_labels, clf.feature_importances_):
    importance_list.append(feature)
```

In [117]:
```
importance_list.sort()
```

In [118]:
```
sorted(importance_list, key=lambda x: x[1],reverse=True)
```

Out[118]:
```
[('V14', 0.21813373430775512),
 ('V12', 0.1417328810969784),
 ('V4', 0.10690809274177429),
 ('V3', 0.1042313166584596l),
 ('V16', 0.061933994054364325),
 ('V7', 0.05858339014163422),
 ('V9', 0.04555370778227845),
 ('V10', 0.03978991708809655),
 ('V17', 0.036704067611840155),
 ('V21', 0.016172851678543092),
 ('V8', 0.012603643998747489),
 ('V13', 0.01223355928437633),
 ('V18', 0.011610083441147816),
 ('Amount', 0.011609669684307319),
 ('V1', 0.011518393922091776),
 ('V5', 0.010117549074782002),
 ('V20', 0.010052283332702729),
 ('V11', 0.009902279562967051),
 ('V26', 0.009591891580903882),
 ('V6', 0.009387382846320347),
 ('V19', 0.008256617175597092),
 ('V15', 0.008140507578531153),
 ('V28', 0.007103920614908966),
 ('V23', 0.007020358824875228),
 ('V27', 0.006774048736845699),
 ('V22', 0.00671512395356999),
 ('V2', 0.006302171675422795),
 ('V25', 0.006019434893518539),
 ('V24', 0.005297126656659555)]
```

In [119]:
```
# Modify Base DataFrame?
dfA = df[['Class','V12']]

print(run_logreg(dfA,1))
print(run_logreg(dfA,2))
```

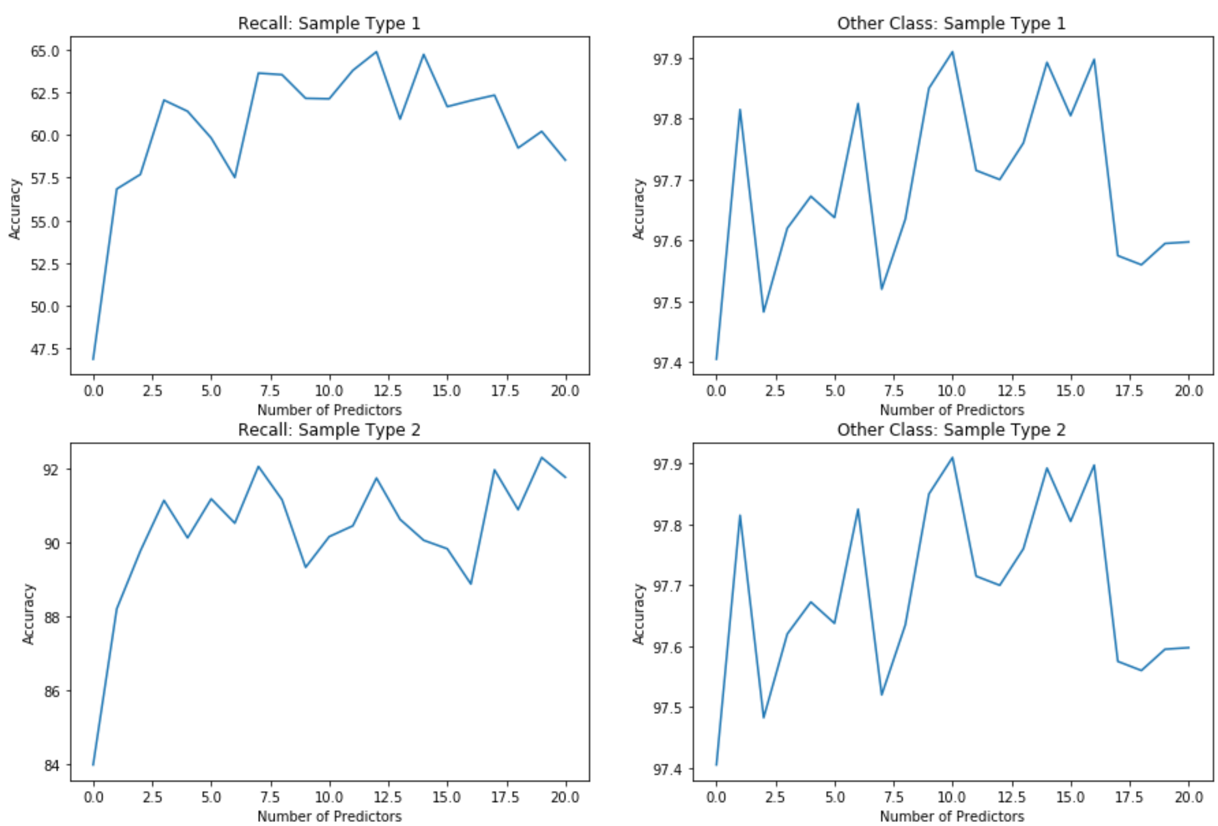
```
('recall:', 43.83, 'precision:', 99.98)
('recall:', 85.81, 'precision:', 93.61)
```

In [120]:
```
# Create a function that figures out the best number of predictors iteratively.
columns = ['Class','V14','V10','V4','V12','V17','V3','V21','V16','V7','V19','V1
```

In [121]:
```
recall_1 = []
precision_1 = []
recall_2 = []
precision_2 = []
# Run function 29x10 times to figure out best predictor
for x in range(2,len(columns)-7):
    # Define columns
    cols = columns[:x]
    df_temp = df[cols]
    tempRecall_1 = []
    tempPrecision_1 = []
    tempRecall_2 = []
    tempPrecision_2 = []
    for x in range(4):
        #Output for Type 1 Sampling Method
        output1 = run_logreg(df_temp,1)
        tempRecall_1.append(output1[1])
        tempPrecision_1.append(output1[3])

        #Output for Type 2 Sampling Method
        output2 = run_logreg(df_temp,2)
        tempRecall_2.append(output2[1])
        tempPrecision_2.append(output2[3])

    #Append the average of the 10 for each variable to the respective lists. To
    recall_1.append(np.mean(tempRecall_1))
    precision_1.append(np.mean(tempPrecision_1))
    recall_2.append(np.mean(tempRecall_2))
    precision_2.append(np.mean(tempPrecision_2))
```

In [131]:
```
plt.figure(figsize=(15,10))

plt.subplot(2,2,1)
plt.plot(recall_1)
plt.title('Recall: Sample Type 1')
plt.ylabel('Accuracy')
plt.xlabel('Number of Predictors')

plt.subplot(2,2,2)
plt.plot(precision_2)
plt.title('Other Class: Sample Type 1')
plt.ylabel('Accuracy')
plt.xlabel('Number of Predictors')

plt.subplot(2,2,3)
plt.plot(recall_2)
plt.title('Recall: Sample Type 2')
plt.ylabel('Accuracy')
plt.xlabel('Number of Predictors')

plt.subplot(2,2,4)
plt.plot(precision_2)
plt.title('Other Class: Sample Type 2')
plt.ylabel('Accuracy')
plt.xlabel('Number of Predictors')
```

Out[131]:
```
Text(0.5, 0, 'Number of Predictors')
```



In [123]:
```
round(recall_1[3],2)
round(recall_2[3],2)
```

Out[123]: 91.14

In [124]:
```
recall_2
```

Out[124]:
```
[83.965,
 88.195,
 89.7625,
 91.135,
 90.1175,
 91.175,
 90.51750000000001,
 92.05499999999999,
 91.15249999999999,
 89.32000000000001,
 90.1525,
 90.44500000000001,
 91.74000000000001,
 90.62,
 90.05250000000001,
 89.82249999999999,
 88.8675,
 91.96249999999999,
 90.8825,
 92.29750000000001,
 91.76]
```

In [127]:
```
def temp(data,sample_type,Cost=1000000000000):
    """
    1. Takes in data
    2. Runs Logistic Regression
    3. Outputs Recall
    """
    X_train, X_test, y_train, y_test = preprocess(data,sample_type)
    from sklearn.linear_model import LogisticRegression

    logmodel = LogisticRegression(C=Cost)
    logmodel.fit(X_train,y_train,)

    predictions = logmodel.predict(X_test)

    from sklearn.metrics import classification_report,confusion_matrix
    rep = confusion_matrix(y_test,predictions)
    recall = np.round((rep[1][1]/sum(rep[1]))*100,2)
    precision = np.round((rep[0][0]/sum(rep[0]))*100,2)
    #print(len(X_train),len(X_test),len(y_train),len(y_test))
    #print(recall)
    #print(rep)
    #return('recall:', recall, 'precision:',precision)
    return rep
```

In [130]:
```
temp(df,1)
```

Out[130]:
```
array([[85293,     6],
       [   52,    92]], dtype=int64)
```

```
cc <- read.csv("~/Downloads/creditcard.csv")

#sampling: 70% of each class
library(class)
n1=length(which(cc$Class==0))*0.7
n2=length(which(cc$Class==1))*0.7
idx_test=c(sample(which(cc$Class==0), n1, replace=FALSE),
        sample(which(cc$Class==1), n2, replace=FALSE))
cc_test = cc[idx_test,]
cc_train = cc[-idx_test,]

#knn all predictors
knn_pred = knn(
  train = cc_train,
  test = cc_test,
  cl = cc_train$Class,
  k = 5)
knn_con = table(true = cc_test$Class, model = knn_pred)
knn_con
cc.pred.knn_error = (knn_con[1,2] + knn_con[2,1])/sum(knn_con)

#knn 2 variables balanced sampling
#knn using only the best 10 predictors
list_best_predictor = c(12,14,10,4,17,11,3,19,8,5)
list_best=sort(list_best_predictor)
library(class)
for(i in 1:10){
  for(j in 1:10){
    if(j>i) {
      knn_pred = knn(
      train = cc_train[,c(list_best[i],list_best[j])],
      test = cc_test[,c(list_best[i],list_best[j])],
      cl = cc_train$Class,
      k = 5)
      knn_con = table(true = cc_test$Class, model = knn_pred)
      knn_con
      cc.pred.knn_error = (knn_con[1,2] + knn_con[2,1])/sum(knn_con)
      print(c(round(list_best[i],1),round(list_best[j],1),cc.pred.knn_error))
      cc.pred.knn_error[i] = cc.pred.knn_error
    }}}
cc.pred.knn_error

#knn 3 variables balanced sampling
```

```
library(class)
for(i in 1:10){
  for(j in 1:10){
    for(x in 1:10){
    if(j>i & x>j) {
      knn_pred = knn(
        train = cc_train[,c(list_best[i], list_best[j], list_best[x])],
        test = cc_test[,c(list_best[i], list_best[j], list_best[x])],
        cl = cc_train$Class,
        k = 5)
      knn_con = table(true = cc_test$Class, model = knn_pred)
      knn_con
      cc.pred.knn_error = (knn_con[1,2] + knn_con[2,1])/sum(knn_con)
      print(c(list_best[i],list_best[j],list_best[x], cc.pred.knn_error))
    }}}}
```

```
my_data <- read.table(file = "clipboard",
                      sep = "\t", header=TRUE)
attach(my_data)
summary(my_data)
str(my_data)

library(caret)
table(my_data$Class)
prop.table(table(my_data$Class))

#Logistic regression before splitting

glm.model=glm(Class~.,my_data,family=binomial())
summary(glm.model)

glm.model.pred.prob = predict(glm.model, my_data, type = "response")
# Convert predictions to class lables (1 or 2, for category 1 or 2, respectively).
glm.model.pred = (glm.model.pred.prob > 0.5) + 1
# Create the confusion matrix by tabulating true classes against predicted classes.
glm.model.conf = table(true = my_data$Class, predicted = glm.model.pred)
glm.model.conf

# Precision: tp/(tp+fp):
glm.model_prec = glm.model.conf[1,1]/sum(glm.model.conf[1,1:2])
glm.model_prec
# Recall: tp/(tp + fn):
glm.model_recall= glm.model.conf[1,1]/sum(glm.model.conf[1:2,1])
glm.model_recall
# F1 score F1 Score might be a better measure to use if
# there is an uneven class distribution (large number of Actual Negatives).
glm.model_F1score = (2*glm.model_recall*glm.model_prec) /
sum(glm.model_recall,glm.model_prec)
glm.model_F1score

#Random Oversampling of the data (ROS)

n_legit <- 284807
new_frac_legit <- 0.50
new_n_total <- n_legit/new_frac_legit # = 284807/0.50 = 569614
library(ROSE)
oversampling_result <- ovun.sample(Class ~ .,
                                   data = my_data,
                                   method = "over"
```

```
                        ,
                        N = new_n_total,
                        seed = 2018)
oversampled_credit <- oversampling_result$data
table(oversampled_credit$Class)

barplot(table(oversampled_credit$Class), col = 4)

#Random Undersampling of the data

n_fraud <- 492
new_frac_fraud <- 0.50
new_n_total1 <- n_fraud/new_frac_fraud # = 492/0.50 = 984
undersampling_result <- ovun.sample(Class ~ .,
                         data = my_data,
                         method = "under"
                         ,
                         N = new_n_total1,
                         seed = 2018)
undersampled_credit <- undersampling_result$data
table(undersampled_credit$Class)

barplot(table(undersampled_credit$Class), col = 4)

#Undersampling and Oversampling Combination of the data


n_new <- nrow(my_data) # = 24600
fraction_fraud_new <- 0.50
sampling_result <- ovun.sample(Class ~ .,
                    data = my_data,
                    method = "both"
                    ,
                    N = n_new,
                    p = fraction_fraud_new,
                    seed = 2018)
sampled_credit <- sampling_result$data
table(sampled_credit$Class)

barplot(table(sampled_credit$Class), col = 4)

#Now let's look at the compare the three methods
```

```r
prop.table(table(oversampled_credit$Class))
prop.table(table(undersampled_credit$Class))
prop.table(table(sampled_credit$Class))


#We will choose the combination of under/oversampling
#Next, split data into training and testing groups

require(caTools)
set.seed(101)
sample = sample.split(my_data$Class, SplitRatio = .75)
train = subset(my_data, sample == TRUE)
test  = subset(my_data, sample == FALSE)


#Note the unbalance between classes
prop.table(table(train$Class))
prop.table(table(test$Class))


#Here we use ubSMote resampling and splitting using unbalanced package from R
#In order to produce featurePlot

balanced <- ubSMOTE(X = my_data[,-31], Y = as.factor(my_data$Class),
                    perc.over=200, perc.under=800,  verbose=TRUE )

balanceddf <- cbind(balanced$X, Class = balanced$Y)

for (i in seq(from =1, to = 30, by = 2))
{
  show(
    featurePlot(
      x = balanceddf[, c(i,i+1)],
      y = balanceddf$Class,plot = "density",
      scales = list(x = list(relation="free"),
                    y = list(relation="free")),
      adjust = 1.5, pch = "|", layout = c(2,1 ), auto.key=TRUE
    )
  )
}


#Run logistic regression on the newbalanced data without those variables
```

```
log.reg.glm=glm(Class~.,test,family=binomial())
summary(log.reg)

log.reg.glm.pred.prob = predict(log.reg.glm, test, type = "response")
# Convert predictions to class lables (1 or 2, for category 1 or 2, respectively).
log.reg.glm.pred = (log.reg.glm.pred.prob > 0.5) + 1
# Create the confusion matrix by tabulating true classes against predicted classes.
log.reg.glm.conf = table(true = test$Class, predicted = log.reg.glm.pred)
log.reg.glm.conf
# Precision: tp/(tp+fp):
log.reg.glm_prec = log.reg.glm.conf[1,1]/sum(log.reg.glm.conf[1,1:2])
log.reg.glm_prec
# Recall: tp/(tp + fn):
log.reg.glm_recall= log.reg.glm.conf[1,1]/sum(log.reg.glm.conf[1:2,1])
log.reg.glm_recall
# F1 score F1 Score might be a better measure to use if
# there is an uneven class distribution (large number of Actual Negatives).
log.reg_F1score = (2*log.reg.glm_recall*log.reg.glm_prec) /
sum(log.reg.glm_recall,log.reg.glm_prec)
log.reg_F1score



library(rpart)
tree1 = rpart(Class ~ ., data = train)

library(partykit)
plot(as.party(tree1))

threshold <- 0.5
predicted_classes <- predict(tree1, test, type = "vector") >= threshold

# Confusion matrix & recall
library(caret)
conf.tree = table(data = predicted_classes, reference = test$Class)
conf.tree

#Recall
conf.tree_rec = conf.tree[2,2]/sum(conf.tree[2,1:2])
conf.tree_rec
# Precision: tp/(tp+fp):
conf.tree_prec = conf.tree[1,1]/sum(conf.tree[1,1:2])
```

```
conf.tree_prec
# Recall: tp/(tp + fn):
conf.tree_recall= conf.tree[1,1]/sum(conf.tree[1:2,1])
conf.tree_recall
# F1 score F1 Score might be a better measure to use if
# there is an uneven class distribution (large number of Actual Negatives).
conf.treeF1score = (2*conf.tree_recall*conf.tree_prec) / sum(conf.tree_prec,conf.tree_recall)
conf.treeF1score
```