Classifying Credit Card Applications

Eric Cai

2023-09-07

The files credit_card_data.txt (without headers) and credit_card_data-headers.txt (with headers) contain a dataset with 654 data points, 6 continuous and 4 binary predictor variables. It has anonymized credit card applications with a binary response variable (last column) indicating if the application was positive or negative. The dataset is the "Credit Approval Data Set" from the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Credit+Approval) without the categorical variables and without data points that have missing values.

In this project, I will be predicting the application statuses of future credit card applicants. I will be using two machine learning models, support vector machines and k-nearest neighbors, to perform the analysis. I will create training, validation, and test sets to determine which model performs the best.

```r
# Reading in txt file
file <- 'credit_card_data.txt'
data <- read.table(file)
```

Next, we split our data into training, validating, and test sets.

Amount of Dating Splitting How much should we partition our sets? Since we need to split our dataset into three different datasets and train models that generalize well to new data, it might be advisable to allocate a larger proportion to training. As such, we'll go with a 70-15-15 breakdown.

Method of Data Splitting The next thing we need to decide is how we want to split it. We can randomly sample 70% of the dataset and allocate it as our training set, but this could possibly introduce bias because they're not equally separated as in the rotation method. For example, the training set might not have an equal distribution of data points compared to the validating or testing sets. On the other hand, the rotation method could introduce bias as well especially if there's an inherent order or pattern in the data.

To address this problem, there's a potential solution in *stratified sampling* where we seek to split our data such that our training, validating, and testing sets have approximately the same percentage of samples of each target class, whether 0 or 1, as the complete set. In this way, we introduce randomness and equal distribution in our datasets, thereby minimizing bias even in the 70-15-15 ratio sets.

First, let's see if stratified sampling is even necessary. Stratified sampling might not be necessary if our dataset has a relatively balanced class distribution.

```r
kable(prop.table(table(data$V11)),
      format='markdown',
      col.names=c('Class', 'Frequency'),
      align='c',
      caption='Class Distribution')
```

Table 1: Class Distribution

Class	Frequency
0	0.5474006
1	0.4525994

Luckily for us, though not perfect, this is almost balanced. We see that there are a bit more denied applications than accepted ones. But we probably can't stratify any further. If we split this data set to allocate 50% 0s and 50% 1s in the training set, we'll have significantly less 0s in the test set.

There are many approaches to the problem, but this method applies cut() because it allows us to divide our sets into specified intervals. We'll then sample from these intervals and then use the split() function to subset our datasets.

```r
## setting our seed to reproduce results
set.seed(1)
## create our partitions
partitions <- c(train = 0.7, validate = 0.15, test = 0.15)
sampled_sets <- sample(cut(
        seq(nrow(data)),
        nrow(data)*cumsum(c(0, partitions)),
        labels=names(partitions)
))
split_sets <- split(data, sampled_sets)
train_set <- split_sets$train
validation_set <- split_sets$validate
test_set <- split_sets$test
```

Let's see the breakdown of our sets.

```r
kable(t(data.frame(sapply(split_sets, nrow)/nrow(data))),
      format='markdown',
      row.names=FALSE,
      col.names=c('Training Set', 'Validation Set', 'Test Set'),
      align='c',
      caption='Split Breakdown'
)
```

Table 2: Split Breakdown

Training Set	Validation Set	Test Set
0.6987768	0.1498471	0.1513761

Let's look at how the classes were distributed.

```r
train_df <- prop.table(table(train_set$V11))
valid_df <- prop.table(table(validation_set$V11))
test_df <- prop.table(table(test_set$V11))
combined <- cbind(train_df, valid_df, test_df)
kable(combined,
      format='markdown',
      col.names=c('Training Set', 'Validation Set', 'Test Set'),
      align='c',
      caption='Class Distribution')
```

Table 3: Class Distribution

	Training Set	Validation Set	Test Set
0	0.5623632	0.4489796	0.5757576
1	0.4376368	0.5510204	0.4242424

Since our original dataset has more 0s than 1s, naturally, we sampled more 0s than 1s. This is probably as close to stratified sampling as we can get, since the breakdown is largely representative of our original dataset.

Split Data on SVM Finally, we're ready to evaluate our models. I'll start with SVM first and then KNN. I'm going to create a function now, so that I can use different data sets and C values for later use.

```
ksvm_models <- function(c_val, data_set) {
        ## we fit the model to our training set
        ## train_set remains static
        c_models <- ksvm(as.matrix(train_set[,1:10]),
                         as.factor(train_set[,11]),
                         type='C-svc',
                         kernel='vanilladot',
                         C=c_val,
                         scaled=TRUE)
        ## data_set is an argument which will be changed below
        pred <- predict(c_models, data_set[,1:10])
        acc <- sum(pred==data_set[,11])/nrow(data_set)
        error_rate <- 1 - acc
        return(c(format(c_val, scientific=FALSE), acc, error_rate))
}
```

I'm going to apply the function to a set of C values in log10 and record it into a table.

```
c_vals <- c(1 %o% 10^(-4:4))
table <- t(sapply(c_vals, ksvm_models, data_set = validation_set))
```

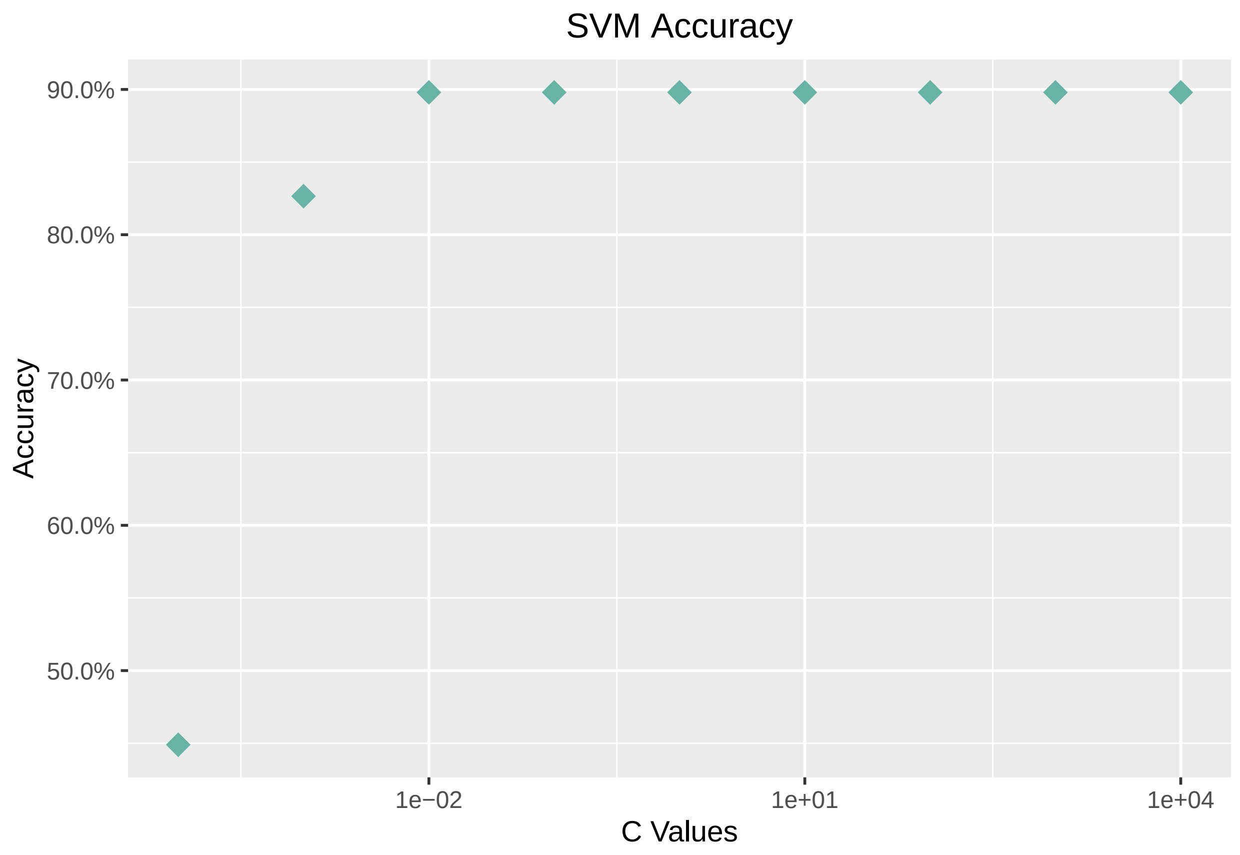
```
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
```

```r
colnames(table) <- c('C values', 'Accuracy', 'Error')
kable(table, format='markdown', align='c',
      caption='Accuracy of C values in SVM')
```

Table 4: Accuracy of C values in SVM

C values	Accuracy	Error
0.0001	0.448979591836735	0.551020408163265
0.001	0.826530612244898	0.173469387755102
0.01	0.897959183673469	0.102040816326531
0.1	0.897959183673469	0.102040816326531
1	0.897959183673469	0.102040816326531
10	0.897959183673469	0.102040816326531
100	0.897959183673469	0.102040816326531
1000	0.897959183673469	0.102040816326531
10000	0.897959183673469	0.102040816326531

```r
df <- data.frame(C_Values = c_vals,
                 Accuracy = as.numeric(table[,'Accuracy']))
ggplot(df, aes(x = C_Values, y = Accuracy)) +
  geom_point(shape = 18, size = 4, color = "#69b3a2") +
  labs(
    title = 'SVM Accuracy',
    x = 'C Values',
    y = 'Accuracy'
  ) +
  scale_x_log10() +
scale_y_continuous(labels = scales::percent_format(accuracy = 0.1)) +
theme(plot.title = element_text(hjust=0.5)
)
```



As expected, the accuracy got better as our C values increased, because we increased the margin. Whether we want a softer or harder classifier really depends on our tolerance for errors or margin size. A smaller margin leads to higher error because we're classifying harder whereas a larger margin leads to lower error, but we may be misclassifying more points. (Under the ISL interpretation of tuning parameter C.) This is why KNN can be an appealing alternative since it classifies a data point based on its closest k neighbors rather than by a hyperplane and its support vectors.

```
best_model <- which.max(table[,2])
best_accuracy <- table[best_model, 2]
best_c <- table[best_model, 1]
cat('Best accuracy:', best_accuracy, '\n')
```

```
## Best accuracy: 0.897959183673469
```

```
cat('Best C:', paste(best_c), '\n')
```

```
## Best C: 0.01
```

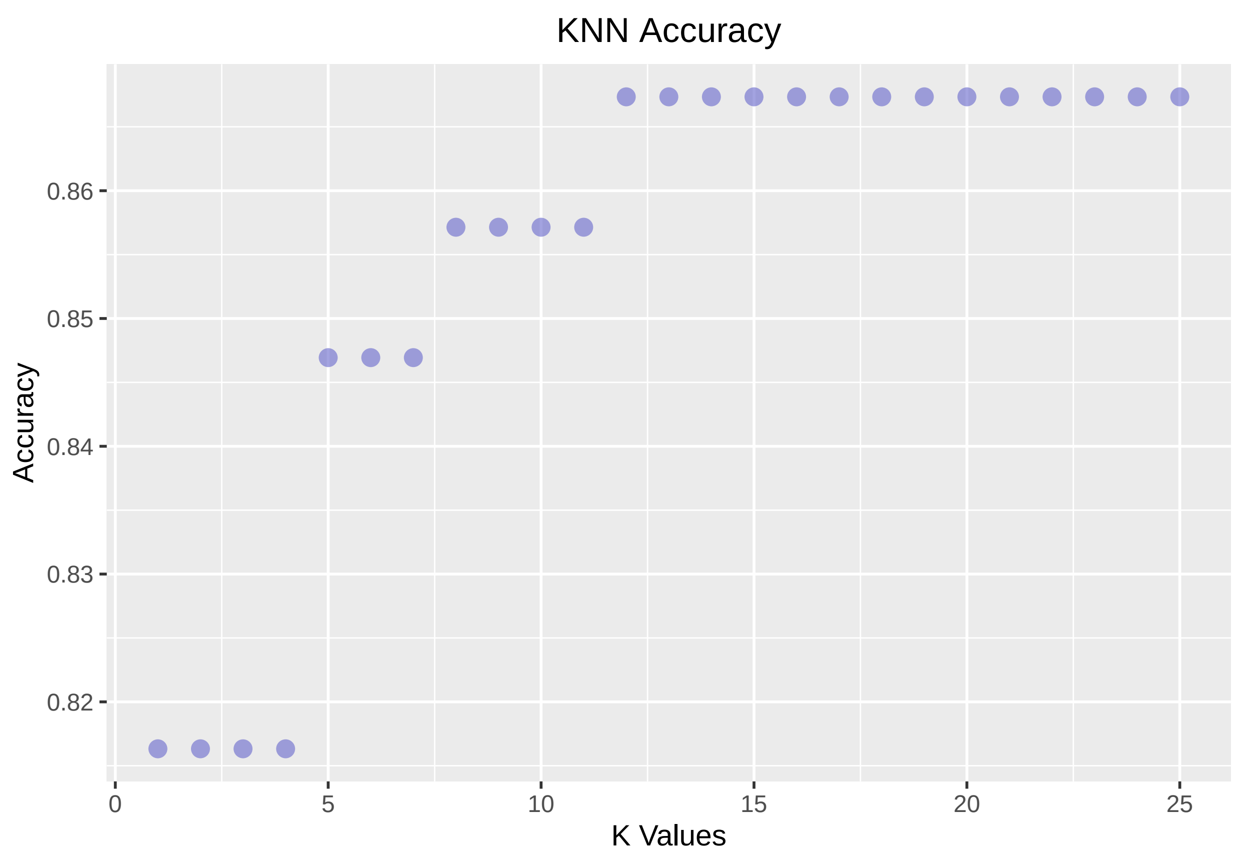
```
cat('Best SVM model: ', best_model)
```

```
## Best SVM model:  3
```

Split Data on KNN Speaking of KNN, let's now see how it performs on the split data, so we can determine which model is better to use to classify our credit card applications. I am not going to use 50 neighbors because that would be overkill and computationally expensive. Let's use 25 neighbors instead. As usual, I'm going to create a function, loop, then store the values so we can plot it.

```r
neighbors <- 25
table2 <- matrix(0, nrow=neighbors, ncol=2)
knn_model <- function(K, data_set){
        kmod <- kknn(V11 ~ ., train_set, data_set, k=K, scale=TRUE)
        predictions <- round(fitted(kmod))
        accuracy <- mean(predictions ==data_set[, 11])
        return(accuracy)
}
## loop over the neighbors
for(K in 1:neighbors){
        table2[K, 1] <- K
        table2[K, 2] <- knn_model(K, data_set=validation_set)
}
```

```r
ggplot(data.frame(table2), aes(x = table2[,1],
                               y = table2[,2])) +
  geom_point(shape = 16,
             size = 3,
             color = rgb(0.4, 0.4, 0.8, 0.6)) +
  labs(
    title = 'KNN Accuracy',
    x = 'K Values',
    y = 'Accuracy'
  ) +
theme(plot.title = element_text(hjust = 0.5))
```



Let's get the best model and k on the validation set so we can compare with the SVM model.

```
best_model2 <- which.max(table2[,2])
best_accuracy2 <- table2[best_model2, 2]
best_k <- table2[best_model2, 1]
cat('Best accuracy for validation set:', best_accuracy2, '\n')
```

```
## Best accuracy for validation set: 0.8673469
```

```
cat('Best K:', paste(best_k), '\n')
```

```
## Best K: 12
```

```
cat('Best KNN model: ', best_model2)
```

```
## Best KNN model:  12
```

Let's put our results in a table and compare.

```
svm_validation <- ksvm_models(c_val=best_c, data_set=validation_set)
```

```
##  Setting default kernel parameters
```

```r
knn_validation <- knn_model(K=best_k, data_set=validation_set)
svm_df <- data.frame(
  C_Value = svm_validation[1],
  K_Value = "",
  Accuracy = svm_validation[2]
)
knn_df <- data.frame(
  C_Value = "",
  K_Value = best_k,
  Accuracy = knn_validation
)
colnames(svm_df) <- colnames(knn_df)
comparison_table <- t(rbind(svm_df, knn_df))
rownames(comparison_table) <- c('C Value', 'K', 'Accuracy')
kable(comparison_table,
      format='markdown',
      row.names = TRUE,
      col.names=c('SVM','KNN'),
      align='c',
      caption='Best Validation Performance on SVM & KNN Model'
)
```

Table 5: Best Validation Performance on SVM & KNN Model

	SVM	KNN
C Value	0.01	
K		12
Accuracy	0.897959183673469	0.86734693877551

Model Selection So it looks like SVM performs better on our validation set than KNN. So we'll use SVM on our test data.

```r
svm_3 <- ksvm_models(c_val=best_c, data_set=test_set)
```

```
##  Setting default kernel parameters
```

```r
test_performance <- data.frame(C_Value = svm_3[1],
                               Accuracy = svm_3[2],
                               Error_Rate = svm_3[3])
kable(test_performance,
      format='markdown',
      row.names = FALSE,
      col.names=c('C Value', 'Accuracy', 'Error Rate'),
      align='c',
      caption=paste('Test Performance for SVM Model', best_model)
)
```

Table 6: Test Performance for SVM Model 3

C Value	Accuracy	Error Rate
0.01	0.838383838383838	0.161616161616162

```r
best_ksvm <- function(c_val){
        model <- ksvm(as.matrix(train_set[,1:10]),
                         as.factor(train_set[,11]),
                         type='C-svc',
                         kernel='vanilladot',
                         C=c_val,
                         scaled=TRUE)
        return(model)
}
svm_predictions <- as.factor(predict(best_ksvm(best_c), test_set[,1:10]))
```

```
##  Setting default kernel parameters
```

```r
confusion_matrix <- confusionMatrix(svm_predictions, reference=as.factor(test_set$V11))
confusion_matrix
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 43  2
##          1 14 40
##
##                Accuracy : 0.8384
##                  95% CI : (0.7509, 0.9047)
##     No Information Rate : 0.5758
##     P-Value [Acc > NIR] : 2.095e-08
##
##                   Kappa : 0.6812
##
##  Mcnemar's Test P-Value : 0.00596
##
##             Sensitivity : 0.7544
##             Specificity : 0.9524
##          Pos Pred Value : 0.9556
##          Neg Pred Value : 0.7407
##              Prevalence : 0.5758
##          Detection Rate : 0.4343
##    Detection Prevalence : 0.4545
##       Balanced Accuracy : 0.8534
##
##        'Positive' Class : 0
##
```

So our confusion matrix tells us that our model classified 43 true negatives and 40 true positives, while misclassifying 2 positives and 14 negatives. This is actually consistent with the breakdown of our test set, where there were more negatives (0) than there were positives (1). An accuracy of 83.84% is still very good. However, accuracy doesn't consider the possibility of correct classifications *due to randomness*. In other words, it's possible for the accuracy to be boosted due to above-average random effects, which is what makes the Kappa statistic a helpful value metric.

The Kappa statistic compares the accuracy of the system to the accuracy of a random system. That is, it evaluates predicted labels alongside the training labels (observed accuracy), while taking into account randomness (expected accuracy), shedding light on how the classifier itself performed.

As Kappa reaches 1, it indicates a near perfect agreement between predictions and actual labels. A Kappa value of 0.6811594 indicates substantial agreement.