Math 533 Midterm

Cesar F. Pardede

19 OCT 2020

On Support Vector Machines

Support Vector Machines (SVMs) are a flexible supervised learning method which maximize the distance between classes. By maximizing the distance, or margin, between classes, SVMs maximize the probability of a new observation being correctly classified. Utilizing a variety of parameters, SVMs have the flexibility model a data set and provide highly accurate classifications. However, this flexibility also leads to SVMs being sensitive to changes in parameters; the accuracy of an SVM is strongly dependent on developing an appropriate model through parameter selection. In this paper, I will present the general concepts of an SVM, so that when I encounter SVMs in my career, I can quickly review this assignment to grasp the big ideas. The details, I have found, are mainly data-dependent; that is, the choice of kernels or parameters will depend on the data being investigated, but I will try to provide as much intuition as possible to assist in choosing suitable kernels or parameters.

First, let's begin with the big picture idea of an SVM. Consider the randomly generated data in Figure 1 (L), we have generated 100 observations each from two different bivariate normal distributions. We can clearly see there are two distinct groups, "class 1" in red, and "class 2" in blue. Suppose we wanted to draw a line separating the two groups, there are an infinite number of lines we could draw that partition the space such that all of class 1 falls on one side, and all of class 2 falls on the other side. An SVM will aim to find the decision boundary that results in the largest margin between the groups (i.e draw the line between both groups that leaves the most space between them and the line). By creating as large a margin as possible, an SVM maximizes the probability that a new observation will be correctly classified depending on its location in relation to the decision boundary. In Figure 1 (R), we show the decision boundary generated by R to classify observations from our randomly generated data. The margins are implied by the observations marked with an 'X', indicating the vectors which contribute to the support of the classification boundary, which is where SVMs and SVCs (support vector classifiers, a special linear classification case of SVMs) get their name.

By creating the largest margin possible with support vectors, we also create a relatively

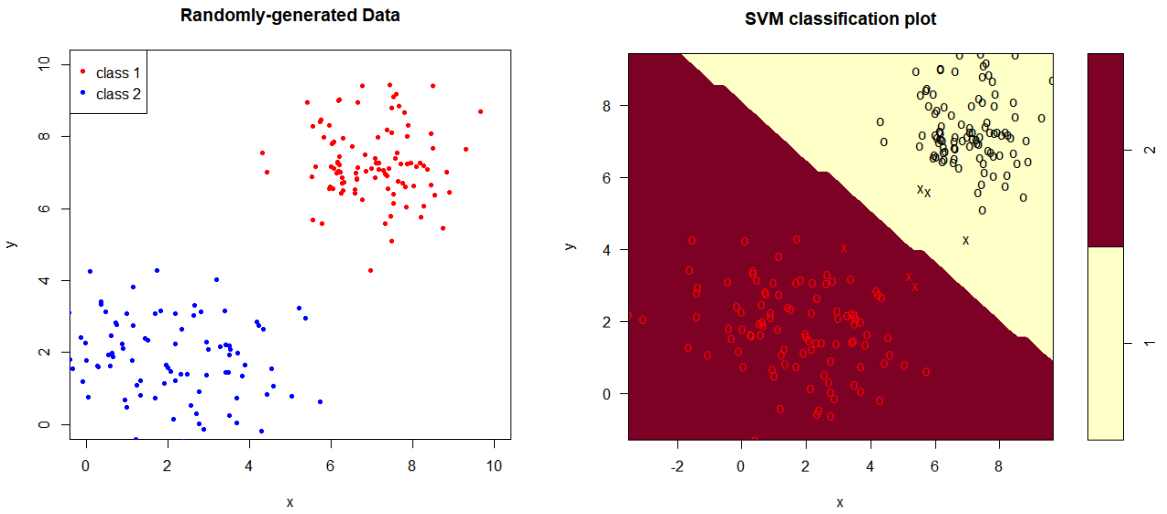

Figure 1: L: Randomly-generated Data from Bivariate Normal Distributions;
R: A Linear SVM Applied to the Random Data

robust method of classification. Since the margins are formed only by the observations nearest the decision boundary, then any new observations further from the boundary than any observations along the margins will not affect the SVM. In this way, SVMs are computationally efficient, because it is not required to hold in memory the location of every observation to determine the decision boundary and margins, only the support vectors that lie near the decision boundary within or on the margins.

There are a number of ways we can draw boundaries to separate the two groups, the decision boundary does not necessarily have to be linear. In Figure 2 we show different decision boundaries generated by different kernels in the SVM algorithm. Depending on the structure of the data, or external information we have about the data, we might be more inclined to choose a particular kernel, and find the best tuning parameters that lead to the highest classification accuracy.

Kernels are simply a way of quanitifying similarity, but they have the added benefits in SVMs of simplifying calculations, and allow easy extension into higher dimensions. Rather than classifying observations and determining the decision boundary based on p dimensions and n observations, which would be computationally intensive given large number of dimensions or observations, SVMs implement the "kernel trick." By instead computing the kernel of every distinct pair of points, we instead classify the observations based on their similarity to one another, then project the points back to the original space to create the decision boundary. This is much more computationally efficient than trying to optimize a decision boundary based on the p features of each observation.

While in the simple 2-dimensional examples of Figure 2, we can visualize the linear decision boundaries as a line, a polynomial, a circle, and a sigmoid, for the linear, polynomial, radial, and sigmoid kernels respectively. In the case of three dimensions, the decision bound-

aries become a plane, a polynomial surface, a sphere, and a sigmoidal surface. As the number of dimensions increases, the form of the decision boundaries becomes less interpretable, such as the shape of a decision hyperplane in higher dimensions, but the idea of kernel clasification remains the same.

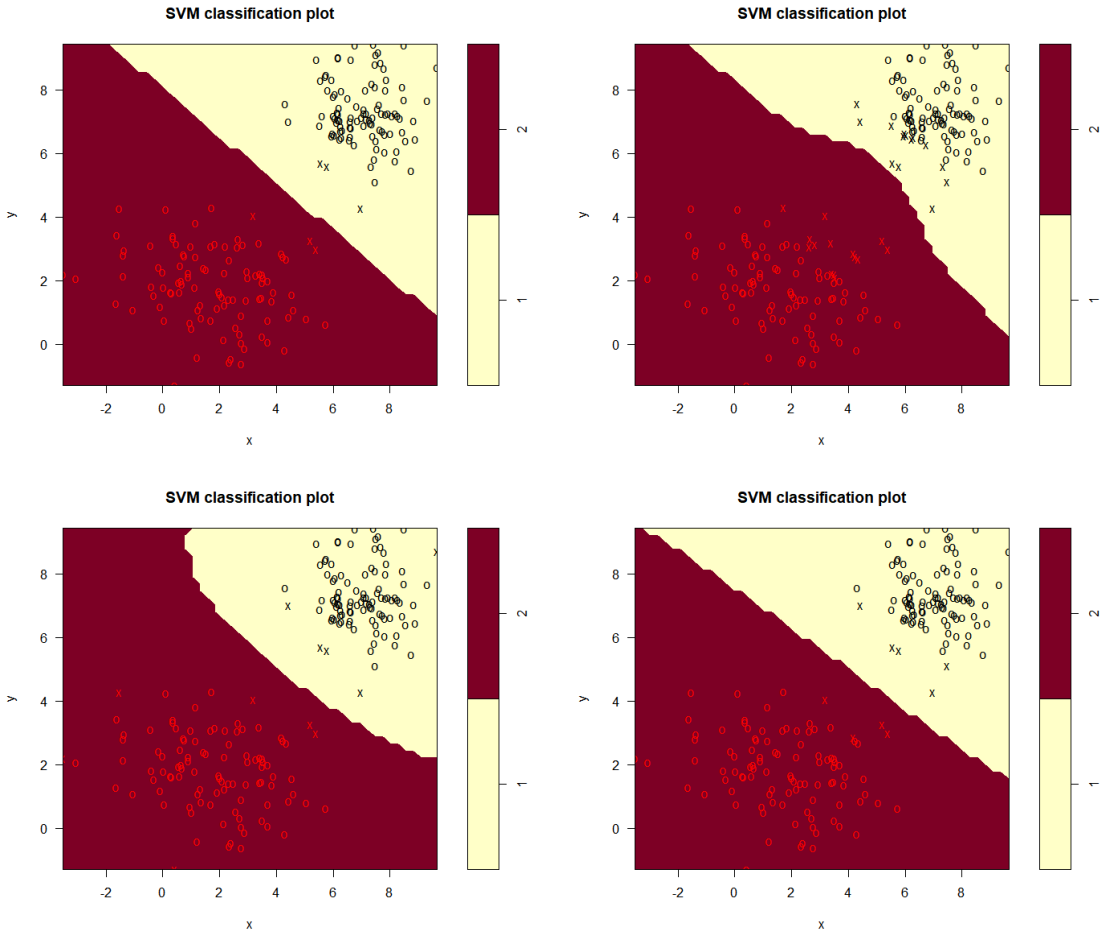

Figure 2: Top row, left to right: decision boundaries generated by a linear kernel, and a polynomial kernel of degree 3;
Bottom row, left to right: decision boundaries generated by a radial basis kernel, and a sigmoid kernel.

In our simple examples, SVMs perform wonderfully at creating clear separations between class, but what happens if the data is not so clear cut? There are various parameters that allow the SVM to relax its margins, and allow for soft classification - either allowing training data to fall within the margins, or even on the other side of the decision boundary. We can use a "budget" parameter (C in ISLR pg. 347, B in CASI pg. 378) to specify the total amount we will allow observations to lie within the margins or beyond the decision boundary. A higher budget allows for more violations, and so contributes to a wider margin.

A similar parameter, is the "cost" parameter which assigns a cost to observations that lie within the margins or beyond the decision boundary. A higher cost parameter indicates a

greater penalty for violating the margins, and so leads to harder margins with fewer violating observations, and so the margins are thinner. This is exemplified in Figure 3 with various values for the cost parameter C, implemented in R. Although the width of the margins are not explicitly pictured, we can infer the width of the margins at each C by the support vectors shown in each plot (marked by an "x"). At larger values of C, we see fewer support vectors, and the few support vectors that may violate the margins are not far within the margins. At smaller values of C, we see the opposite - more support vectors with support vectors further within the margins.

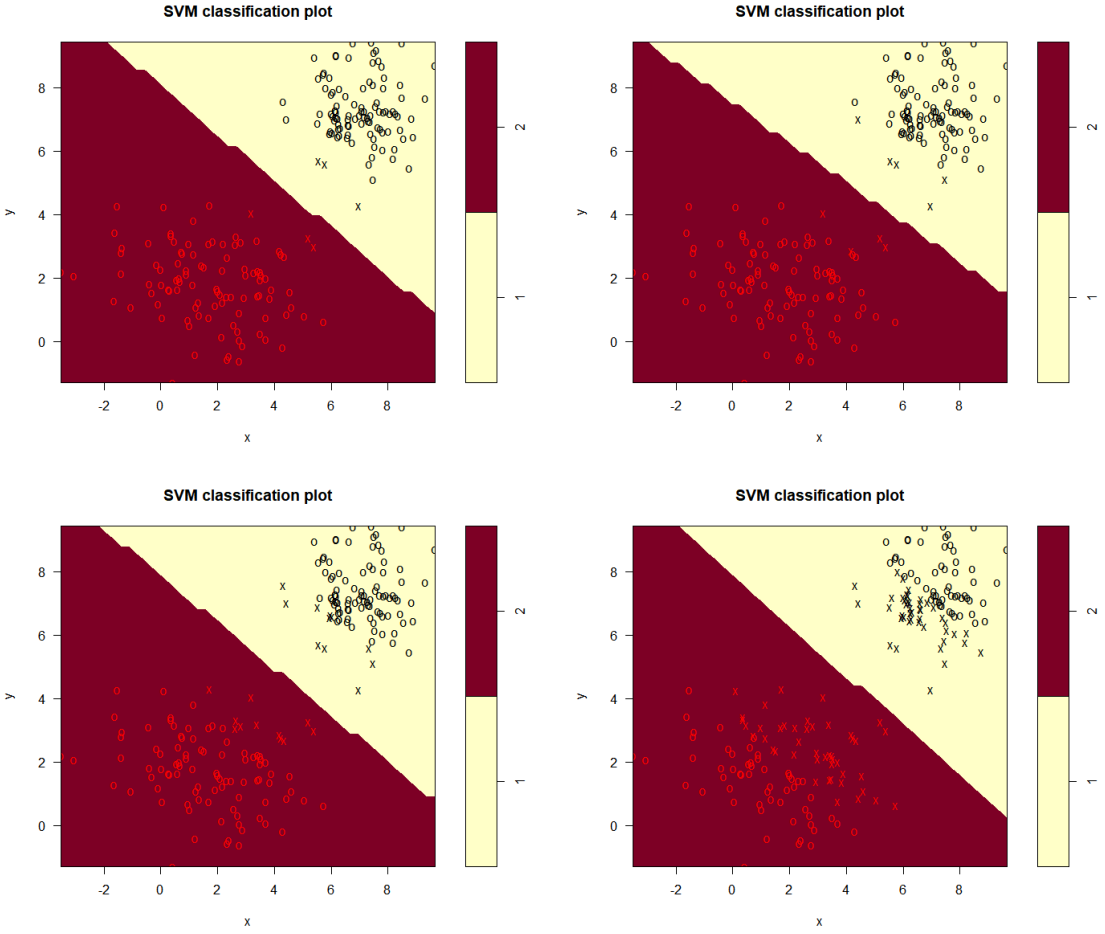

Figure 3: Various values of the "cost" parameter C in an SVM - top left: C = 10, top right: C = 1, bottom left: C = 0.1, bottom right: C = 0.01.

We have already considered SVMs in higher dimensions, now we discuss SVMs beyond binary classification. In Figure 4, we have added another group (in black) to our previously random data. SVMs can be extended beyond binary classification using "one-vs-all" (OVA) and "one-vs-one" (1V1) comparisons, essentially reducing the multi-class problem to a series of binary classifications which lets us use all of the previous methods, parameters, and properties of SVMs mentioned before.

In OVA comparisons, the multi-class problem is reduced to the same binary classification

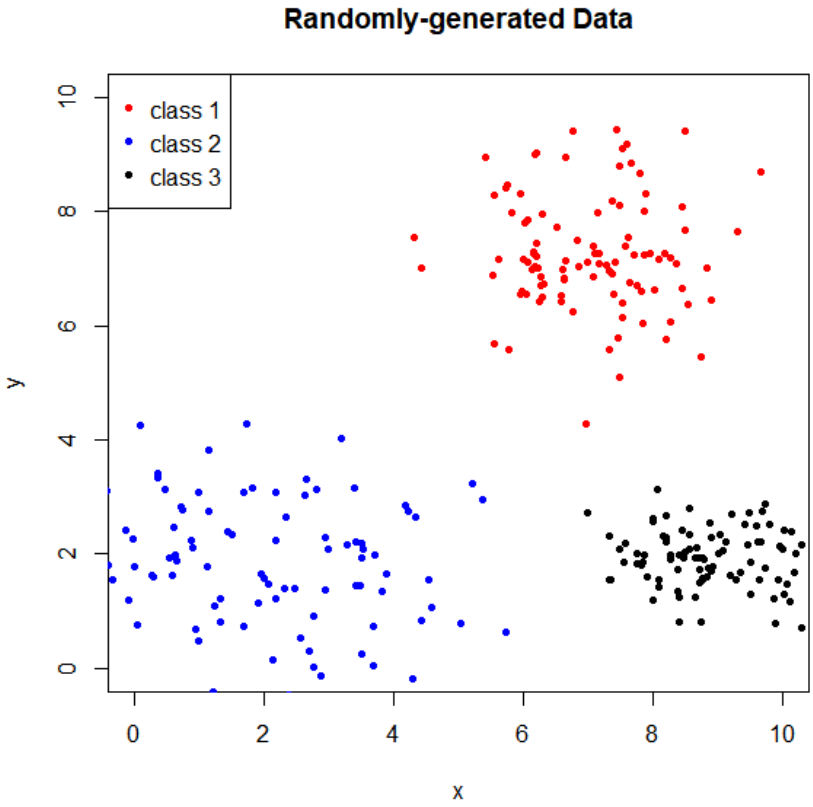

Figure 4: Randomly-generated data with 3 distinct groups.

problem by comparing one class against ALL other classes together to draw decision boundaries and maximize margins. This is repeated for each class until every class has been "the one", then, a new observation is classified based on which class maximizes the kernel of the observation.

In 1v1 comparisons, pairwise comparisons are made between each group of observations to create boundaries and maximize margins. Since a new observation is classified multiple times during the 1v1 comparisons, it is assigned to the class for which it was classified the most during pairwise comparisons. The e071 package in R implements multi-class SVMs using 1v1 comparisons. Figure 5 shows examples of SVM classifications for three classes with various kernels using the e071 package.

Now, we take a moment to reflect on the concepts and parameters mentioned, and lend a word of warning to myself who might want to use SVMs in the future. As powerful as SVMs have appeared in this paper, note the extremely simple data used throughout. We can see even now that in the binary and multi-class examples, while the SVMs are able to correctly classify most (if not all) of the observations, SVMs don't actually reveal the underlying nature of the data (each group was generated from a bivariate normal). SVMs will draw decision boundaries any way you tell it, it is indiscriminate in that manner. It will

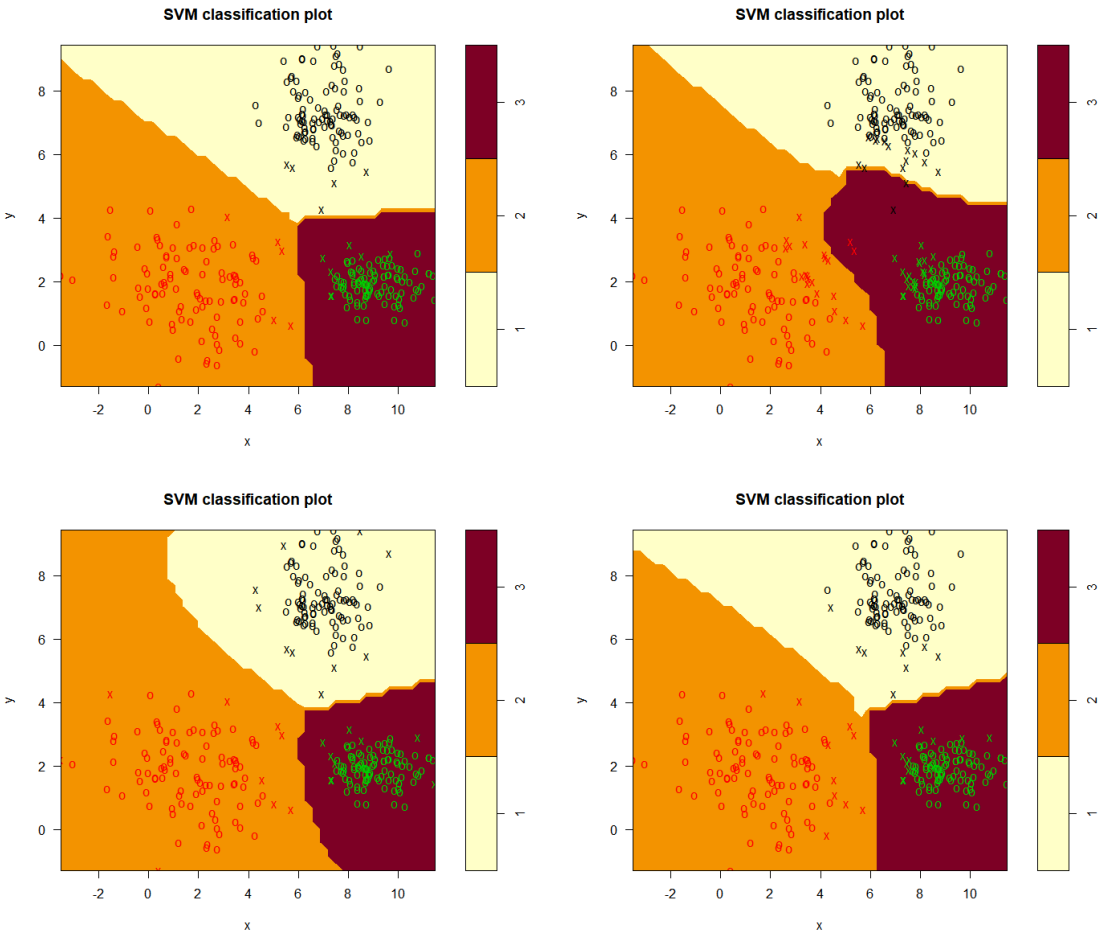

Figure 5: Decision boundaries generated by various kernels for 3 distinct groups.

be up to you to uncover the underlying structure of data you encounter, through judicious applications of statistical methods.

We mentioned the flexibility of SVMs in our introduction, and how its flexibility can lend great predicative power in classification using a variety of parameters and kernels, but this is a double-edged sword. We have seen that SVMs are sensitive to kernel specification, the cost parameter, budget parameter, and the number of perceived groups. There are even more parameters we did not mention here including the hinge loss, and parameters specific to kernel choice. The same flexibility that allows us to finely tune an SVM also invites us to overfit a model.

With so many knobs to turn and levers to pull, it is too easy to create an SVM with a high accuracy rate on the training data only to suffer a significantly lower accuracy in testing. Each parameter or kernel specification is an assumption on the data, and it must be asked whether one is selecting a parameter or kernel because we believe it more closely emulates the data, or only because it improves our metric for accuracy. Take for example the linear SVM in Figure 5 (top left); we know the underlying structure of each group is

a bivariate normal, the linear SVM correctly classifies the observations with straight line decision boundaries, but we know that is not the true structure of the data.

In conclusion, SVMs are a very powerful tool for data scientists. As a computationally efficient supervised classification algorithm with a host of tuning parameters, it is a fast and flexible method for performing classification on large multidimensional datasets. Its major disadvantage is its sensitivity to parameter and kernel selection; small differences in parameters or a difference choice of kernel can lead to wildly different models. Take care when applying SVMs not to over-specify parameters for a high training accuracy rate; just like any machine learning algorithm, be wary of over-fitting the data. The code to generate the data and plots from this section can be found in the appendix in the file "svm_examples.R".

SVMs Applied to the Adult Dataset

In this section, we apply SVMs to the Adult Dataset while taking into consideration the structure of the data. We choose to select the first 10000 entries in the Adult Dataset as training data, and the next 2000 entries as testing data. We remove the variables "education" and "fnlwgt". We remove "education" because the variable "education_num", a numeric value for education level already exists. We remove "fnlwgt", because it is a weighting factor for each observation calculated by the data curators, while it may be useful for classification, classifying by "fnlwgt" is senseless to us - we want to classify on the variables we can interpret right now.

At this point, we let 'race' be our response variable, and let all others be our predictor variables. We want to see if SVMs and other classification algorithms can accurately predict a person's race based on the other data collected about them. We move into some exploratory data analysis, and discover that the classes in our training data are heavily imbalanced. Out of 10000 observations, 8556 are labeled 'White' - almost six times as all the other labels combined! We are faced with a decision, do we continue with our raw training data and the issue of imbalanced classes, or do we try to "fix" the imbalance somehow. We take a quick detour to explore the possibility of correcting the imbalance.

We discovered a few methods of addressing imbalances through our online research. Downsampling is the practice of restricting the number of observations from some class or classes in a training set so that over-represented classes don't heavily bias the classification method. Upsampling is a similar idea which resamples from classes that are under-represented in the training data to artificially inflate their sample size. Considering the large disparity in classes with this data (the smallest class contains only 83 observations, while the largest class contains 8556 observations), we try to create our training data with a combined method that uses both approaches.

We take the total size of the training data (10,000 observation), divide it by the number of classes (in this case, 5), then sample or resample that number of observations from each class. This gives us a perfectly balanced training set, albeit with quite a bit of resampling for originally smaller classes. We felt this was an interesting compromise that needed to be explored, because we thought the massive imbalance between classes could not be resolved by simple downsampling or upsampling. Unfortunately, this method did not improve our classification results; it performed worse than the raw data set in every iteration of the SVM we tried, with every kernel choice and with every set of parameters we selected. We disregard this training set, and continue our exploration of SVM with the raw training set. We also disregard most of the code in the associated R file, to clean up the code, and prevent us from spending more time on it in the future.

Before we begin exploring SVMs, note the proportion of observations labeled 'White' (0.8556). If we were to guess every class of the training data was 'White', we would be correct 0.8556 of the time without any sophisticaed form of classifcation. In order for an SVM, or any other classification algorithm, to be considered successful on this dataset, we

need to exceed a 0.8556 accuracy rate.

We begin examining SVMs with a linear kernel. There aren't many parameters to tune with a linear kernel, but we can allow cost (the penalty for an observation falling within the margins) to vary among 10, default = 1, 0.1, 0.01. When tested on the test data, the highest accuracy a linear kernel SVM was able to achieve for these parameter values was 0.889 with cost equal to the default value 1.

Next, we look at SVMs with a polynomial kernel. We allow the degree parameter of the polynomial to vary among 2, default = 3, 4, 5, and again allow cost to vary among 10, default = 1, 0.1, 0.01. We did not notice any significant differences in classification accuracy when applied to the test data; for every degree-cost pair, the accuracy rate was 0.866 or 0.8655. We might as well leave the default parameters for a polynomial kernel when we move on to tune the γ parameter.

We try a sigmoid kernel next, and tune the γ and coef0 parameters of sigmoid kernel together with the γ and coef0 parameters of a polynomial kernel where γ varies among 0, 0.001, 0.01, 0.1 and coef0 varies among default = 0, 0.1, 1, 10. We don't find any improvement with different parameters compared to the default parameter values where $\gamma = 1/\text{p}$. The accuracy rate when tested with the test data is 0.866.

The last kernel we try is a radial kernel and choose to allow the γ parameter to vary among 0, 0.001, 0.01, 0.1, and the cost parameter to vary among 10, default = 1, 0.1, 0.01 again. We get a slightly better accuracy rate (0.8865) with $\gamma = 0.01$ and cost = 10 than we do with the default parameters $\gamma = 1/\text{p}$, cost = 1. The radial kernel provided better results than the polynomial kernel, and the sigmoid kernel, but is still beaten by a regular SVM with a linear kernel and no parameter tuning.

Based on classification accuracy, the SVM with a linear kernel with default parameter cost = 1 performed the best out of all the other SVMs with every kernel and parameter combination mentioned here. Additionally, there was some parameter testing not shown here, such as tuning the nu-parameter and performing nu-classification, and tuning the epsilon insensitive-loss parameter, but these did not produce competitive results (the classification accuracy rates were far below the baseline 0.8556 proportion) so are disregarded in the code and not explained here.

Since the SVM with a linear kernel and default parameters performed the best, we carry on that model to make comparisons against random forest models, and boosting (decision tree) models.

Comparing SVMs to Tree-Based Methods

Similar to what we did for SVMs, we try various parameters for random forests and decision trees. For random forests, we call the randomForests function, and iterate through different

values for ntree (the number of trees to generate) and mtry (the number of variables to consider for each split). We iterate through 100, default = 500, 800 for ntree, and the floor of $\sqrt{p}$, $p/2$, p for mtry, where p is the number of predictor variables (12). We found the most accurate random forest model, as measured by accuracy in predicting the testing data, when ntree = 500 and mtry = 3, with an accuracy rate of 0.8865.

For decision trees, we call the gbm function (from the package gbm, generalized boosting modeling), and iterate through various values for n.trees (the number of trees to fit). We let n.trees be 100, 500, 800, and found the highest accuracy when n.trees = 800, with accuracy being 0.8905.

So, in the end, we have SVMs achieved a highest accuracy rate of 0.889, random forests achieved a highest accuracy rate of 0.8865, and boosting achieved a highest accuracy rate of 0.8895 on the testing data. All of these rates are higher than the baseline proportion of 0.8655 we mentioned before. Although the achieved accuracy rates appear close to the baseline proportion, we are confident that the difference between the achieved accuracy rates and the baseline proportion is significant. While not shown in the code, we have generated some of these models multiple times with different training data; the results were always very similar to simply sampling the first 10000 samples, so we continued generating the training set that way to simplify things. In addition, many of the functions utilized during this discussion have built in cross-validation outputs, and observing those show these algorithms continually exceed the baseline proportion. Although the difference between randomly guessing every label is 'White', and implementing sophisticated classification algorithms is slight, it does exist; we can say confidently that SVMs, random forests, and boosting perform better than no classification at all on the Adult Dataset.

We want to get a more detailed look at the predictions from each algorithm, so we print the contingency tables for each algorithm below.

```
> table(svm_pred, test_data[, 'race'])
svm_pred               Amer-Indian-Eskimo  Asian-Pac-Islander  Black  Other  White
Amer-Indian-Eskimo                     0                   0      0      0      0
Asian-Pac-Islander                     0                  42      0      1      2
Black                                  0                   0      7      0      0
Other                                  0                   0      0      0      0
White                                 13                  17    180     10   1729

> table(rf_pred, test_data[, 'race'])
rf_pred                Amer-Indian-Eskimo  Asian-Pac-Islander  Black  Other  White
Amer-Indian-Eskimo                     0                   0      0      0      0
Asian-Pac-Islander                     0                  37      0      1      2
Black                                  0                   0     22      0     15
Other                                  0                   1      0      0      0
White                                 13                  21    165     10   1714

> table(gb_pred, test_data[, 'race'])
gb_pred                Amer-Indian-Eskimo  Asian-Pac-Islander  Black  Other  White
Asian-Pac-Islander                     0                  41      0      1      2
Black                                  0                   0     11      0      2
White                                 13                  18    176     10   1727
```

As we feared, many of the misclassifications are due to each algorithm classifying obser-

vations as 'White' when they actually belonged to another class, presumably because of the overwhelming number of 'White' observations. This was to be expected; the class imbalance is a difficult problem to solve, and is something we would like to look into more deeply if I had time available. We suspect it will require more sophisticated data augmentation techniques to resolve.

There are other little details we notice in the contingency tables. The SVM, for some reason, is able to predict observations belonging to the "Asian-Pac-Islander" (API) class. While only have 309 samples in the training data (3rd most amount of samples), the SVM makes the second most number of predictions for the API class, and correctly classifies 42/45 of them. Despite being the second largest class, only 7 observations were predicted to be "Black" with 100% accuracy. It is also worth noting that no observations were predicted as belonging to the "Amer-Indian-Eskimo" (AIE) and "Other" classes, probably because of the small sample sizes for each class (99 and 83, respectively).

It is a similar sentiment for the random forest model. There are multitude of misclassifications for 'White' predictions. The API class has the second most number of predictions with a high accuracy rate (37/40). This time, the "Black" class has more observations, 37 in total, but a lower accuracy - correctly predicting 22/37 observations. Once again, no observations were predicted to belong to the AIE class, and one observation predicted to be "Other" actually belonged to the API class.

The boosting model strongly resembles the linear SVM with 41/44 correct classifications to the API class, and 11/13 correct classifications to the "Black" class. No predictions were made to the AIE or "Other" class, a row for the number of predictions doesn't even appear in the contingency table.

What we can conlude from these tables, is there there is something about the API class that distinguishes it from the "White" class more strongly than the other classes. While the number of observations misclassified as "White" is large, more than half of the API samples were able to be correctly predicted as belonging to the API class (42/59 for SVMs, 37/59 for random forests, and 41/59 for boosting). Compare this to the number of misclassified "Black" observations, or the number observations correctly predicted as belonging to "Black" against the total number of "Black" observations (7/187 for SVMs, 22/187 for random forests, and 11/187 for boosting). The characteristics of the API class allow it to be more easily predicted than other classes.

If I had to choose a "winner" in these comparisons, based on the results of this paper, I would say random forests are the best algorithm to start an initial exploration into classification. It did not perform the best overall, in fact it had the lowest accuracy among our final three methods, but it made predictions to the smaller-sized classes more often - it was the method least overwhelmed by the class imbalance. In addition, we found random forests performed quicker, in training and in prediction; so it's possible to try many different models and parameters in a short span of time.

SVMs, as powerful as they are, their flexibility require a lot of time and effort to tune

the many parameters. Sometimes, it can take a while to train an SVM depending on the chosen parameters preventing us from creating as many models as we could with random forests in a short time span. It might be possible to define an SVM that does a great job of prediction, but the time required to do so (compared to training a random forest), and the number and range of parameters ensure that it is no trivial task. It's possible they could overcome the class imbalance issue, and we simply missed the correct parameters, but they are too time-intensive to be my go-to method, I would try random forests first.

The code for this section and the section "SVMs applied to the Adult Dataset" are included in the appendix as "Pardede_533_Midterm.R".

Appendix: svm_examples.R

```
# Midterm
# Math 533
# 2020-10-19
# include e1071, contains svm function
library(e1071)

x1 = rnorm(100, 7, 1)
x2 = rnorm(100, 2, 2)
y1 = rnorm(100, 7, 1)
y2 = rnorm(100, 2, 1)

data1 = data.frame(x = x1, y = y1, class = "1")
data2 = data.frame(x = x2, y = y2, class = "2")
data = rbind(data1, data2)

plot(data1$x, data1$y, col = 2, pch = 20, xlim = c(0, 10), ylim
    = c(0, 10), xlab = 'x', ylab = 'y', main = 'Randomly-
   generated Data')
points(data2$x, data2$y, col = 4, pch = 20)
legend('topleft', c('class 1', 'class 2'), pch = c(20, 20), col
    = c(2, 4))

linear_svm = svm(class ~ y + x, data = data, kernel = 'linear')
polynom_svm = svm(class ~ y + x, data = data, kernel = '
   polynomial')
radial_svm = svm(class ~ y + x, data = data, kernel = 'radial')
sigmoid_svm = svm(class ~ y + x, data = data, kernel = 'sigmoid
   ')

plot(linear_svm, data = data)
plot(polynom_svm, data = data)
plot(radial_svm, data = data)
plot(sigmoid_svm, data = data)

linear_svm1 = svm(class ~ y + x, data = data, kernel = 'linear
   ', cost = 10)
linear_svm1 = svm(class ~ y + x, data = data, kernel = 'linear
   ', cost = 1)
linear_svm3 = svm(class ~ y + x, data = data, kernel = 'linear
   ', cost = 0.1)
linear_svm4 = svm(class ~ y + x, data = data, kernel = 'linear
   ', cost = 0.01)
```

```
plot(linear_svm1, data = data)
plot(linear_svm2, data = data)
plot(linear_svm3, data = data)
plot(linear_svm4, data = data)

x3 = rnorm(100, 9, 1)
y3 = rnorm(100, 2, 0.5)
data3 = data.frame(x = x3, y = y3, class = "3")
data = rbind(data, data3)

plot(data1$x, data1$y, col = 2, pch = 20, xlim = c(0, 10), ylim
    = c(0, 10), xlab = 'x', ylab = 'y', main = 'Randomly-
   generated Data')
points(data2$x, data2$y, col = 4, pch = 20)
points(data3$x, data3$y, col = 1, pch = 20)
legend('topleft', c('class 1', 'class 2', 'class 3'), pch = c
   (20, 20, 20), col = c(2, 4, 1))

linear_svm5 = svm(class ~ y + x, data = data, kernel = 'linear
   ', cost = 1)
polynom_svm2 = svm(class ~ y + x, data = data, kernel = '
   polynomial')
radial_svm2 = svm(class ~ y + x, data = data, kernel = 'radial
   ')
sigmoid_svm2 = svm(class ~ y + x, data = data, kernel = '
   sigmoid')

plot(linear_svm5, data = data)
plot(polynom_svm2, data = data)
plot(radial_svm2, data = data)
plot(sigmoid_svm2, data = data)
```

Appendix: Pardede_533_Midterm.R

```
#### Cesar Pardede ####
# Midterm
# Math 533
# 2020-10-19
# include e1071, contains svm function
library(e1071) # for SVM
library(gbm) # for boosting
library(randomForest) # for random forests

#### VARIABLE SELECTION AND DATA CLEANING ####
# read in data. we'll use 10000 samples for training, and
   randomly sample
# another 2000 samples for testing
n = 10000
train_data <- data.frame(read.csv('Fall 2020/Math 533/Midterm/
   adult.data')[1:n, ])
test_data <- data.frame(read.csv('Fall 2020/Math 533/Midterm/
   adult.data')[n:(n+2000),])
predictors = c('age', 'workclass', 'fnlwgt', 'education', '
   education_num',
'marital_status', 'occupation', 'relationship', 'race',
'sex', 'capital_gain', 'capital_loss', 'hours_per_week',
'native_country', 'income')
names(train_data) = predictors
names(test_data) = predictors

# we don't need fnlwgt or education
train_data <- train_data[, names(train_data) != 'education' &
   names(train_data) != 'fnlwgt']
test_data <- test_data[, names(test_data) != 'education' &
   names(test_data) != 'fnlwgt']
predictors = predictors[predictors != 'education' & predictors
   != 'fnlwgt']


#### EDA ####
# we can get quick summaries of each variable if we want
summary(train_data) # imbalanced race
str(train_data)

# check for missing values
for (names in predictors){
```

```
        print(length(which(is.na(train_data[names]) == F)))
} # no missing values

for (names in predictors){
        print(length(train_data[train_data[names] == '?']))
} # no missing values marked as '?'

# select response and predictors
# we choose race here, but leave the response as variable and
   write code
# for a general response so we can make explore other responses
    later
response = 'race'
predictors = predictors[predictors != response]

#### CLASS IMBALANCE HANDLING ####
# create training sets
# we'll draw from the remaining data to make test data later.
   there are major
# issues with class imbalances, but lets try balancing it
   ourselves

# approach #1: nothing - just read in the data as is
train_data1 = train_data

# approach #2: downsample/upsample combo - take equal number of
    samples
# and resamples for each class
num_classes = dim(unique(train_data[response]))
n_k = n/num_classes

ob_wh = sample(which(train_data['race'] == ' White'), n_k,
   replace = F)
ob_bl = sample(which(train_data['race'] == ' Black'),  n_k,
   replace = T)
ob_as = sample(which(train_data['race'] == ' Asian-Pac-Islander
   '),  n_k, replace = T)
ob_am = sample(which(train_data['race'] == ' Amer-Indian-Eskimo
   '),  n_k, replace = T)
ob_ot = sample(which(train_data['race'] == ' Other'),  n_k,
   replace = T)

train_data2 = train_data[c(ob_wh, ob_bl, ob_as, ob_am, ob_ot),
   ]
### NOTE: After considerable testing, it has been discovered
```

```
   that this uniform
# sampling method does not work. Every time we have generated
   SVMs from the
# raw training data, and the uniformly sampled training data,
   the SVM generated
# by the uniformly sampled training data has ALWAYS performed
   worse as measured
# by accurate predictions using each SVM model and the same
   test set. We will
# proceed with the rest of the code using train_data1 - the raw
    training data.
# A sample of some of the testing follows below.

# some testing comparing the effect of training on the raw
   training data and
# the combo training data.
lin_svm = svm(formula, data = train_data1, kernel = 'linear',
   cross = 5, cost = 1); lin_svm1$accuracies
lin_svm2 = svm(formula, data = train_data2, kernel = 'linear',
   cross = 5, cost = 1); lin_svm2$accuracies
# combo sampling performed worse  with a linear kernel

lsvm1_pred = predict(lin_svm1, test_data)
table(lsvm1_pred, test_data[, 'race'])
lsvm2_pred = predict(lin_svm2, test_data)
table(lsvm2_pred, test_data[, 'race'])
# interesting results

poly_svm1 = svm(formula, data = train_data1, kernel = '
   polynomial', cross = 5); poly_svm1$accuracies
poly_svm2 = svm(formula, data = train_data2, kernel = '
   polynomial', cross = 5); poly_svm2$accuracies

# combo sampling performed worse  with a polynomial kernel
psvm1_pred = predict(poly_svm1, test_data)
# table(psvm1_pred, test_data[, 'race'])
psvm2_pred = predict(poly_svm2, test_data)
length(which(psvm2_pred == test_data[, response]))/2000
# table(psvm2_pred, test_data[, 'race'])
# interesting results again

### NOTE: based on these results we decide to stop training
   with the
# combo/uniform training data. Instead, just use the raw
   training data.
```

```
# base acc. if we guess white every time
length(which(train_data[, 'race'] == ' White'))/n # 0.8556


#### SVM KERNEL SELECTION AND PARAMETER TUNING ####
# svm; go through the process of finding best parameters for
   each kernel
formula = as.formula(paste(response, ' ~ ', paste(predictors,
   collapse = ' + ')))

cost_acc = list()
i = 1
for (c in c(10, 1, 0.1, 0.01)){
        lin_svm = svm(formula, data = train_data1, kernel = '
           linear', cross = 5, cost = c); lin_svm$accuracies
        lsvm_pred = predict(lin_svm, test_data)
        cost_acc[[i]] = list(cost = c, acc = length(which(
           lsvm_pred == test_data[, response]))/2000)
        i = i + 1
} # best accuracy when cost = 1; 0.889

deg_acc = list()
i = 1
for (d in c(2, 3, 4, 5)){
        for (c in c(10, 1, 0.1, 0.01)){
                poly_svm = svm(formula, data = train_data1,
                   kernel = 'polynomial', cross = 5, degree = d
                   , cost = c); poly_svm$accuracies
                psvm_pred = predict(poly_svm, test_data)
                deg_acc[[i]] = c(deg = d, cost = c, acc =
                   length(which(psvm_pred == test_data[,
                   response]))/2000)
                i = i + 1
        }
} # accuracy stays unchanged 0.866 for different degrees and
   different cost values
# stay with default degrees and cost

rad_acc = list()
i = 1
for (g in c(0, 0.001, 0.01, 0.1)){
        for (c in c(10, 1, 0.1, 0.01)){
                rad_svm = svm(formula, data = train_data1,
                   kernel = 'radial', cross = 5, gamma = g,
```

```
                cost = c); rad_svm$accuracies
            rsvm_pred = predict(rad_svm, test_data)
            rad_acc[[i]] = c(gam = g, cost = c, acc =
                length(which(rsvm_pred == test_data[,
                response]))/2000)
            print(rad_acc[[i]])
            i = i + 1
        }
}

gam_acc = list()
i = 1
for (g in c(0, 0.001, 0.01, 0.1)){
        for (c in c(0, 0.1, 1, 10)){
            poly_svm = svm(formula, data = train_data1,
                kernel = 'polynomial', cross = 5, gamma = g,
                 coef0 = c); poly_svm$accuracies
            psvm_pred = predict(poly_svm, test_data)
            sig_svm = svm(formula, data = train_data1,
                kernel = 'sigmoid', cross = 5, gamma = g,
                coef0 = c)
            ssvm_pred = predict(sig_svm, test_data)
            gam_acc[[i]] = c(gam = g, coef = c, poly_acc =
                length(which(psvm_pred == test_data[,
                response]))/2000,
            sig_acc = length(which(ssvm_pred == test_data[,
                 response]))/2000)
            print(paste(g, c, i))
            i = i + 1
        }
}
# all this tuning didn't do much, accuracy remained the same as
    with default parameters
# the best performing SVM was the SVM with a linear kernel with
    default
# parameter cost = 1, and accuracy of 0.889

#### COMPARE SVMS TO RANDOM FORESTS AND DECISION TREES (
   BOOSTING) ####

p = length(predictors)
i = 1
rf_list = list()
for (ntree in c(100, 500, 800)){
        for (m in c(floor(sqrt(p)), floor(p/2), p)){
```

```
                rf_model = randomForest(formula, data =
                   train_data1, ntree = ntree, mtry = m)
                rf_pred = predict(rf_model, test_data, type = '
                   response')
                acc_rf = length(which(rf_pred == test_data[,
                   response]))/2000
                rf_list[[i]] = c(ntree = ntree, mtry = m, acc =
                    acc_rf)
                print(i)
                i = i + 1
        }
}# best performance by rf: acc = 0.8865, ntree = 500, mtry = 3

i = 1
gb_list = list()
for (ntree in c(100, 500, 800)){
        gb_model = gbm(formula, data = train_data1, cv.folds =
           5, n.trees = ntree)
        gb_pred = predict(gb_model, test_data, n.trees = 100,
           type = 'response')
        gb_pred = apply(gb_pred, 1, which.max)
        gb_pred[gb_pred == 1] = ' Amer-Indian-Eskimo'
        gb_pred[gb_pred == 2] = ' Asian-Pac-Islander'
        gb_pred[gb_pred == 3] = ' Black'
        gb_pred[gb_pred == 4] = ' Other'
        gb_pred[gb_pred == 5] = ' White'
        acc_gb = length(which(gb_pred == test_data[, response])
           )/2000
        gb_list[[i]] = c(n.trees = ntree, acc = acc_gb)
        print(i)
        i = i + 1
}# best performance by gb: acc = 0.8905, ntree = 800

# best SVM, RF, and Boosting models
svm = svm(formula, data = train_data1, kernel = 'linear', cross
    = 5)
rf_model = randomForest(formula, data = train_data1, ntree =
   500, mtry = 3)
gb_model = gbm(formula, data = train_data1, cv.folds = 5, n.
   trees = 800)

svm_pred = predict(svm, test_data)
rf_pred = predict(rf_model, test_data, type = 'response')
gb_pred = predict(gb_model, test_data, n.trees = 100, type = '
   response')
```

```
gb_pred = apply(gb_pred, 1, which.max)
gb_pred[gb_pred == 1] = ' Amer-Indian-Eskimo'
gb_pred[gb_pred == 2] = ' Asian-Pac-Islander'
gb_pred[gb_pred == 3] = ' Black'
gb_pred[gb_pred == 4] = ' Other'
gb_pred[gb_pred == 5] = ' White'

svm_acc = length(which(svm_pred == test_data[, response]))/2000
rf_acc = length(which(rf_pred == test_data[, response]))/2000
gb_acc = length(which(gb_pred == test_data[, response]))/2000

accuracy = list(svm = svm_acc, rf = rf_acc, gb = gb_acc)
accuracy

table(svm_pred, test_data[, 'race'])
table(rf_pred, test_data[, 'race'])
table(gb_pred, test_data[, 'race'])
```