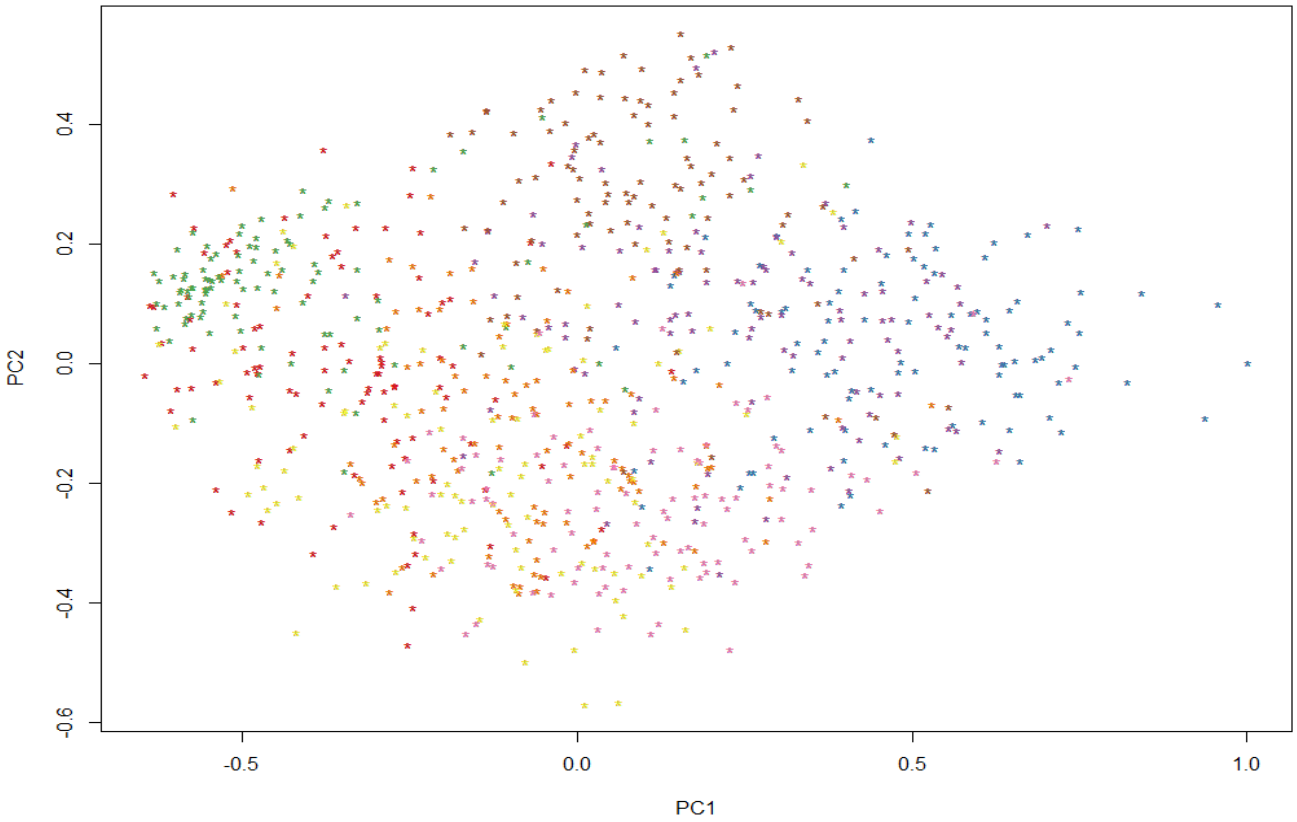
Statistics 154 - Final Project

Team C.I.A.: Audrey **C**hou
Matthew **I**annone
Arif **A**li

Our goal for this project was to accurately predict photograph region (mountain, highway, coast, etc) on unlabeled data using a training dataset of 800 photographs. To begin the analysis, we performed principal component analysis on the training dataset, to see if we could spot any obvious trends between the various classifications. The results, color coded by region, were as follows:



While overall this looks like a mess, there are many regions that are clearly dominated by one type of landscape. We also note right away that the red, orange, and yellow data, corresponding to classes 1, 5, and 6, seem to be very mixed together without any clear separation, while other classes seem easily distinguishable. We kept this in mind for later as we tried to improve our fit.

The next order of business was to fit several baseline predictions. Clustering gave us extremely low accuracy, so that was out of the question. The methods we tried next were two powerful models: random forests and support vector machines. The random forest model was fitted to the entire training dataset with 100 trees grown using the randomForest function. On

the training dataset this model had an 82% accuracy, but on the test dataset accuracy was only 70%. Examining the misclassification matrix showed that the model had particular trouble distinguishing between 1s and 6s, something we anticipated from the PCA.

The second model, support vector machines (SVM), required using the R package 'e1071'. In order to proceed with multi-classification via SVM, we specified the type in the SVM function as "C-classification". When attempting to build the SVM model, it was decided that each variable (GIST) would be used in the the multi-classification. Because of the hidden nature of the test data, we decided to split the training data up into a train and test subgroup. The idea was to see if by using a portion of the training data, we could find specific errors the model tends to make, so that if need be the model could reclassify as necessary with the use of another algorithm. The SVM model was trained on a subset of the training data set and accounted for 700 of the 800, when running the test on the remaining 100 testing rows, we had an internal accuracy rate of 86%. When looking at the individual classifications, we found a very high error rate with 6s (57%). When comparing it to the actual results, it was noted that the pictures miss-classified as 6s were actually 1s. KSVM models were also applied to see if there would be an improvement in accuracy. However, KSVM offered no observable prediction improvement on the fraction of the test dataset.

At this point we decided to modify the code to better account for the high errors with regards to misclassification of 6s as 1s and vice versa. This lead to the decision to use the SVM model as our base, and attempt to improve it by reclassifying 1s and 6s.

To better classify 1s and 6s we tried several ideas. First, we trained a 5-nearest neighbor on the full dataset, and replaced any prediction of 1 and 6 from the SVM with the prediction given by 5NN. This idea failed miserably - KNN was far from an improvement over the methods we were already using.

Next, we tried to use logistic regression to try to help isolate which points the SVM incorrectly predicted as a 1 or a 6. We first targeted the 6's, reclassifying the training data as 0-1 variables, where 1 indicated a 6 and 0 indicated anything else. After splitting the training data into training and testing subsets, we constructed a sparse logistic regression classifier based on the training subset. The sparse classifier only considers a select few predictors, to prevent overfitting, and based on the predictions made on the testing subset, it was highly accurate. For both 6's and 1's, using sparse logistic regression gave us low error. However, the predictions for 6's and 1's were not mutually exclusive, and we surmised that combining sparse classifiers for each of the 8 classes would end up with significantly larger errors.

We then attempted a variant of this method, taking the predictions based on the random forest model and subsetting the predictions that were equal to 1 or 6. Since the original model tended to mix up 1's and 6's, and sparse logistic regression appeared to reduce this error based on the simulations we ran on the training data, we thought that this would improve our accuracy. In this case, we set our subset to 0-1 data based on whether or not the prediction was a 1, and after constructing our new sparse model we replaced the 0's with 6's. We then fit this subset back into the data. Unfortunately, this did not improve our overall accuracy, even when we fit it to SVM instead (although SVM was an improvement over random forest).

After this, we tried training an advanced model on the dataset consisting purely of ones and sixes, in the hope that this more targeted model could distinguish between them better than our original SVM. The two models we tried for this were multinomial boosting and Kernel SVM, with the gbm and ksvm functions respectively. Once these models were trained on the one/six dataset, they were used to re-predict anything svm originally predicted as a one or a six. Both models seemed to improve our training fit, but neither was particularly effective at improving our fit on 50% of the test data, changing the final prediction accuracy by at most a tenth of a percent.

We also attempted neural netting, testing it first on the training data. We separated the data into a training set and a testing set, and then fit a model to the training subset using the multinom() command from the package 'nnet'. The usual nnet() command only considers one class, while multinom() accounts for multiple classes. We then used this model to predict the results of the test set, compared it to the actual results, and got roughly 80% accuracy. After this test confirmed that neural netting worked, we ran it on the actual test set, using the entire training set to fit a model. However, this only gave us roughly the same accuracy as before, so there was no improvement.

Our final test accuracy, using SVM and boosting, was 76.589% on half of the test data. We conclude that while not amazing, our classifier is reasonably effective given the difficulty of the task, and works well enough as a solid first screening of the picture data. In the end, despite all the effort we put into properly classifying the two we were never able to change our final predictions by more than 30 numbers from the original SVM classification. If given additional funding (or unlimited access to the test dataset), this fit could be potentially improved by investigating additional kernels for KSVM or trying to improve other aspects of the fit besides the especially problematic distinction between 1s and 6s.

Code:

```
library(kernlab)
library(cluster)
library(FNN)
library(glmnet)
library(randomForest)
library(e1071)
library(RColorBrewer)


###Matt###
#initial data processing
test_im = read.csv('C://Users//Bob/Desktop//test (1).csv', header = FALSE)
train_im = read.csv('C://Users//Bob/Desktop//train (1).csv', header = FALSE)

names(train_im) = c('y', paste('V', 1:ncol(test_im), sep = ''))
train_im$y = as.factor(train_im$y)

#pca
pcomp_vals = prcomp(train_im[,-1])$x
cols = brewer.pal(8, 'Set1')
plot(pcomp_vals, pch = '*')
for(i in 1:8){
  points(pcomp_vals[train_im$y == i,], col = cols[i], pch = '*')
}

#random forest
rF = randomForest(x = train_im[,-1], y = train_im[,1], ntrees = 1000)
preds = predict(rF)
sum(preds == train_im$y)/nrow(train_im)
rF #confusing 6 and 1

###for svm see arif's code; predictions on training set → preds

#knn
pr_os = train_im[preds == 6 | preds == 1,]
pr_os$y = factor(pr_os$y)

tnn = knn(train = train_im[,-1], test = pr_os[,-1], cl = train_im[,1], k = 5)
preds[preds == 6 | preds == 1] = tnn

#audrey's addition
train2 = train_im[preds == 6 | preds == 1,]
train2$y = as.numeric(train2$y==1)
train.sparse = glmnet(x=as.matrix(train2[,-1]), y=train2[,1], family="binomial")
lambda = numeric()
for (i in 1:length(train.sparse$lambda)){
  lambda [i] = sum(coef(train.sparse, s = train.sparse$lambda[i])[,1]!=0)
}
which(lambda == 11) #26 27 give the same indices for nonzero vectors
```

```
portfolio = coef(train.sparse, s = train.sparse$lambda[27])[,1]
portfolio = portfolio[which(portfolio!=0)]
portfolio
sparsevectors = c(3, 11, 124, 130, 203, 251, 262, 283, 327, 422)
train2 = train2[, c(1, sparsevectors+1)]
pred_test = predict(rF, test_im)
train.lr=glm(y~., binomial, data=train2)
pr_os = test_im[pred_test == 1 | pred_test == 6, c(sparsevectors)]
train.predict=predict(train.lr, pr_os)
train.predict = as.numeric(train.predict>0)
train.predict[train.predict==0]=6
pred_test[pred_test == 1 | pred_test == 6] = train.predict

##sample output code to upload to kaggle, will not include every one of these
pred_test = predict(rF, newdata = test_im)
pr_os = test_im[pred_test == 1 | pred_test == 6,]
tnn = knn(train = train_im[,-1], test = pr_os, cl = train_im[,1], k = 5)
pred_test[pred_test == 1 | pred_test == 6] = tnn

write.csv(data.frame(Predictions = pred_test), 'C:\\Users\\Bob\\Desktop\\datatest1.csv')
#yes my username is Bob, long story

#Note that from this point on some variables were loaded via a
# workspace from other members, and so code won't work as is

#isolating problem points
svm_err = data.frame( train_im$y[pred_svm != train_im$y] ,pred_svm[pred_svm !=
train_im$y] )
names(svm_err) = c('ac', 'pred')

#apologies, the code really isn't clean after this point…
#hopefully you're not actually reading this :(

pred_svm = predict(img.svm, test_im)
pred_ksvm = predict(k_svm, test_im)
pred_svm[pred_svm == 1 | pred_svm == 6] = pred_ksvm[pred_svm == 1 | pred_svm == 6]

train_pred_svm = predict(img.svm, train_im)
onesix_loc =train_pred_svm == 1 | train_pred_svm == 6
onosix = train_im[onesix_loc,]

k_svm_16 = ksvm(y~., data = onosix)
k16_pred_tr = predict(k_svm_16)
k16_pred_tr[k16_pred_tr == 5] = 6 #why is it doing this

train_pred_svm[onesix_loc] = k16_pred_tr

test_pred_svm = predict(img.svm, test_im)
onesix_loc =test_pred_svm == 1 | test_pred_svm == 6
testsix = test_im[onesix_loc,]
```

```
k16_pred_te = predict(k_svm_16, newdata = testsix)
k16_pred_te[k16_pred_te == 5] = 6 #why is it doing this

test_pred_svm[onesix_loc] = k16_pred_te


#fitting boost for ones and sixes
boost = gbm(y ~ ., data = onosix, distribution = 'multinomial', interaction.depth = 2,
shrinkage = .01)
prd = predict(boost, newdata = onosix, n.trees = 100)
prod = NULL
prd = data.frame(prd)
for( i in 1:193){
  prod[i] = (1:8)[which(prd[i,] == max(prd[i,]))]
}
prod[prod == 5] = 6
sum(prod == onosix$y)

test_pred_svm = predict(img.svm, test_im)
oneandsix = test_pred_svm == 1 | test_pred_svm == 6
boost_pred = predict(boost, newdata = test_im[oneandsix,], n.trees = 100)
prod = NULL
boost_pred = data.frame(boost_pred)
for( i in 1:sum(oneandsix)){
  prod[i] = (1:8)[which(boost_pred[i,] == max(boost_pred[i,]))]
}
prod[prod == 5] = 6
test_pred_svm[oneandsix] = prod


##Defunct code section - including for completeness, don't bother
#trying to understand this code though
preds[preds == 1 | preds == 4 | preds == 5 | preds == 7] = predicted.img.class[preds == 1
| preds == 4 | preds == 5 | preds == 7]

for(i in 1:8){
  print(i)
  print(1 - mean((predict(img.svm) == train_im$y)[predict(img.svm) == i]))
  print(1 - mean((predict(rF) == train_im$y)[predict(rF) == i]))
  print('-----')
}

##trying adaboost raw, as an alternative base to svm; it was bad
boost = gbm(y ~ ., data = train_im, distribution = 'multinomial', interaction.depth = 2,
shrinkage = .01)
prd = predict(boost, newdata = train_im, n.trees = 100)
prod = NULL
prd = data.frame(prd)
for( i in 1:800){
```

```
  prod[i] = (1:8)[which(prd[i,] == max(prd[i,]))]
}
sum(prod == train_im$y)

prd = predict(boost, newdata = test_im, n.trees = 100)
prod = NULL
prd = data.frame(prd)
for( i in 1:nrow(test_im)){
  prod[i] = (1:8)[which(prd[i,] == max(prd[i,]))]
}


###Arif###
setwd("~/Dropbox/School/Statistics/Stat 154 Spring 2014/Picture Classification")
###testing from known sample
img = read.csv("train.csv", header = F)
names(img) = c("V1", as.character(1:512))
img$V1 = as.factor(img$V1)

test.rows = sample(1:dim(img)[1], 300)
test.rows = test.rows[order(test.rows)]
training.sample = img[test.rows, ]
testing.sample = img[-test.rows, -1]

library("e1071")
img.svm = svm(V1~., data = training.sample, type = "C-classification")
predicted.img.class = predict(img.svm, testing.sample)

test = predicted.img.class == img[-test.rows, 1]
names(test) = img[-test.rows, 1]
test
mean(test)

1 - mean(test[img[-test.rows, 1]==8])
#[1] 0.109375
1 - mean(test[img[-test.rows, 1]==7])
#[1] 0.08928571
1 - mean(test[img[-test.rows, 1]==6])
#[1] 0.5671642
1 - mean(test[img[-test.rows, 1]==5])
#[1] 0.09090909
1 - mean(test[img[-test.rows, 1]==4])
```

```
#[1] 0.1578947
1 - mean(test[img[-test.rows, 1]==3])
#[1] 0.2058824
1 - mean(test[img[-test.rows, 1]==2])
#[1] 0.1323529
1 - mean(test[img[-test.rows, 1]==1])
#[1] 0.1846154
#beat Matt 1 4 5 & 7, so method will be favored in this case
#[1] 0.802 this is the overall prediction accuracy based on created test data

###completing the model.
train_img = read.csv("train.csv", header = F)
names(train_img) = c("V1", as.character(1:512))
train_img$V1 = as.factor(train_img$V1)
test_img = read.csv("test.csv", header = F)
names(test_img) = as.character(1:512)
img.svm = svm(V1~., data = train_img, type = "C-classification")
predicted.img.class = predict(img.svm, test_img)

write.csv(data.frame(id = c(1:length(predicted.img.class)),Predictions =
predicted.img.class), file = "predictions.cs")
#0.76589 was the actual accuracy went tested against the Kaggle data

###Audrey###
train = read.csv("~/train.csv", header = F) #800 513
test = read.csv("~/test.csv", header = F) #1888 512
names(train) = c("y", names(train)[1:512])

#clustering (k-medoids), which was totally ineffective
library(cluster)
{
accuracy.kmeds = c()
for (j in 1:20){
comp.kmeds = pam(train[,-1], k=20) #512 500
comp.cluster = as.numeric(comp.kmeds$clustering)
classifiers = c()
for (i in 1:length(unique(comp.cluster))){
  comp.table = table(train[which(comp.cluster==i), 1])
  classifier = as.numeric(names(which(comp.table == max(comp.table))))
  classifiers = c(classifiers, classifier)
}
#1 3 2 8 4 6 8 5 7 5
comp.cluster2 = classifiers[comp.cluster]
accuracy.kmeds = c(accuracy.kmeds, sum(comp.cluster2[which(train[,1]==6)] ==
```

```
                    train[which(train[,1]==6),1])/length(train[which(train[,1]==6),1]))
}
mean(accuracy.kmeds)
#0.465 - k=8
#0.46 - k=10
#0.44 - k=15
#0.33 - k=20 #yeah these aren't very good accuracies
}

#neural netting

library(nnet)
train2 = train
samps = sample.int(800, 500)
train.train = train2[sort(samps),]
train.test = train2[-sort(samps),]

train.nnet = multinom(y~., data = train.train, MaxNWts = 4200)
mean(predict(train.nnet, train.test[,-1])==train.test[,1])

train.nnet2 = multinom(y~., data = train, MaxNWts = 4200)
testdf = data.frame(Id = 1:nrow(test), Predictions = as.numeric(predict(train.nnet2,
test)))
View(testdf)

write.table(testdf, file = "154_submission.csv", sep = ",", col.names = c("Id",
"Predictions"), row.names = F)

###sparse logistic regression

library(glmnet)
train2 = train
train2$y = as.numeric(train$y==6) #only looking at 6's for now
train.sparse = glmnet(x=as.matrix(train2[,-1]), y=train2[,1], family="binomial")
lambda = numeric()
for (i in 1:length(train.sparse$lambda)){
  lambda [i] = sum(coef(train.sparse, s = train.sparse$lambda[i])[,1]!=0)
}
which(lambda == 11) #17
portfolio = coef(train.sparse, s = train.sparse$lambda[12])[,1]
portfolio = portfolio[which(portfolio!=0)]
portfolio
sparsevectors = c(5, 9, 13, 48, 100, 108, 112, 128, 262, 281, 391, 394, 410, 415, 450,
454, 503, 506) #nonzero vectors for 6's
sparsevectors1 = c(7,11,131,247,251,282,283,370,374,379) #ran for 1's instead of 6's
later on
train2 = train
train2$y = as.numeric(train$y==1)
train2 = train2[, c(1, sparsevectors+1)]
```

```
accuracy.lr = c()
for(i in 1:1000){
samps = sample.int(800, 500)
train.train = train2[sort(samps),]
train.test = train2[-sort(samps),]

train.lr = glm(y~., binomial, data=train.train)
train.predict = predict(train.lr, train.test[,-1])
train.predict = as.numeric(train.predict>0)
prop = sum(train.predict==train.test[,1])/length(train.test[,1])
accuracy.lr = c(accuracy.lr, prop)
}
1-mean(accuracy.lr)

#0.09039333 error sparse lambda[12] for 6's -- much lower than before
#0.08377333 error for 1's

#variant of 1’s/6’s subset added to Matt’s code
```