report test

Martin Duggan

28/01/2020

# Introduction to Machine Learning: MNIST written character classification

1.1

We can use machine learning to help solve many different types of problems. One problem machine learning can be useful in solving is the classification problem. This is the problem of identifying which set of categories a new observation belongs to. This has many useful applications in the real world such as classifying emails as either spam, or not spam. Classification is a problem in supervised learning, this is where we have training dataset **x** which corresponds to a set of outputs **t** (our classifications). Our dataset **x** is usually made up of many variables/ dimensions D, which is used to map to our outputs. We then learn a function to go from our data **x** to our target y(**x**)=**t**. We can then use this function with new test data to predict our new output. You wouldn’t normally use unsupervised learning to help solve a classification problem, as unsupervised learning techniques clusters similar observations. In some circumstances you could manually classify the clusters but this isn’t always recommonded as the algorithm may be clustering on different features of the data than what you are expecting.

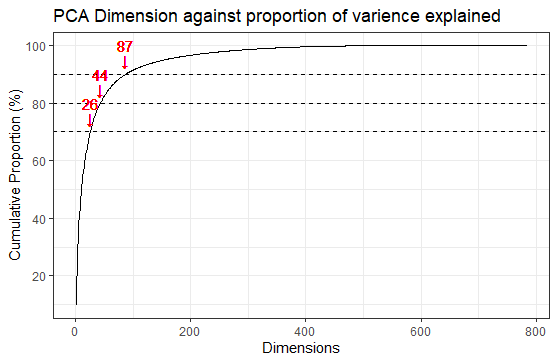
1.2

When it comes to creating our model it is standard practice to split our data up into a training and testing datasets. The training dataset is used to train our model, then we use our test dataset to see how good are model is. Splitting the data allows us fairly test our model and help reduce the risk of overfitting. This is where we fit the model exactly to the data instead of the features in the data. Using a training and test dataset is used in both supervised and unsupervised machine learning. and there are a few keys things to be aware of when making our datasets. Firstly it’s important that we don’t train our model on test data as this will cause bias when testing our model and may cause overfitting and artifially increase the accuracy of our test. It is also good practice to make sure that the training dataset covers the edges of each dimension, this improves our model as we aren’t extrapolating outside of our known model domain. In practise we would normally split our data up into 70% training and 30% test data. This is to ensure we have a large dataset to train our model and still have a sufficient amount data left to test our model. When it comes to classification and supervised learning we use the training dataset to learn the relationship between variables (**x**) and our labels (**t**), then we test how accurate the model on the test data. It is a good idea to check the distribution of the labels (**t**) with the training and test data, this is to ensure that your are modelling on different data and that you’re testing all classifications.

1.3

The MNIST dataset contains information on handwritten digits from 0-9. Each digit is stored as an individual 28x28 matrix where each element is scaled between 0 and 255, where 0 indicates a blank square (no hand writing) and 255 is a filled square (hand writing). This is a useful dataset where the classification problem comes into play. Here we have 784 (28^2) variables which we can classify as one of 10 labels (0-9). The dataset is also quite large at 60,000 observations which means we can have a good sized training and test dataset which allows us to build and sufficiently test our model.

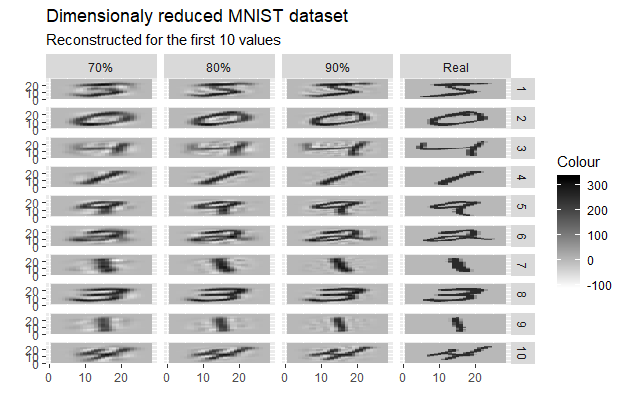
Due to the size of the dataset it can take a long time to perform analysis on the dataset. This is partly due to the large number of variables. To speed up and make analysis on this dataset viable we can apply dimension reduction. This is where we reduce the number of variables (dimensions) to increase the speed of computation. The problem is as we reduce the number of variables we are loosing valueable information, which is the variance observered in the dataset. It’s important to keep as much of the varience in the data as possible because we can loose features of the data which are important for classification as we reduce the varience. Rather than just removing variables/dimensions we are going to reduce the dimensions using Principal Component Analysis (PCA). This is where we orthogonally transform our data using eigen-vectors. This means converting the dimensional planes then the first dimension contains the most variation present in the orginal dataset, this slowly reduces with each dimension until we have the same number of variables and can exactly reproduce the orginal data. This allows us to reduce the dimensions to the first n variables (dimensions) without losing much accuracy in the data. When performing PCA we have options to scale and center our PC coordinates. Centering moves your coordinates from the origin to the center of the centroid, this doesn’t change the analysis but it can make it easier to understand your new PCA dataset. For this reason we have centered ourare PCA on the MNIST dataset. Scaling can alter your analysis quite a lot. Scaling normalises the distances between the dimensions, by stretching the dimensions. This can be useful when your dimensions vary wildly. For instance if one dimension varies between 0-5 and another dimension varies between 0-100, if you don’t scale the axises the smaller axis will be underepresented and it will put more emphasis on the larger dimension for your categorisation. But by scaling the axis you can give equal importance to each dimension. The general concensus is that you want your principal components to contain at least 80% of the variation in the data but this may be higher.



Above we can see when we plot the cumulative proportion of variance explained against dimensions that we need 44 dimensions to explain 80% of the variance. To get a balance between number of dimensions and proportion of variance explained, it is often a good balance to go for the “elbow” of the curve. We can see on the plot above this is around 90% or 87 dimensions (see chart below).

|  |  |  |  |
| --- | --- | --- | --- |
| % of variance explained | 70% | 80% | 90% |
| Number of dimensions | 26 | 44 | 87 |

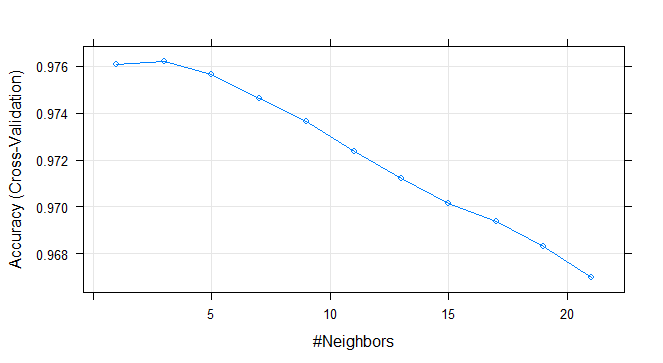
PCA is reversible (through matrix multiplication see R code) which allows us to go from our reduced dimension dataset back to our original matrix. This can be very useful in seeing the affects reducing our dimensions has. As you can see below we have tested 3 different dimension reduced datasets (70%, 80% and 90% of variance explained) and compared them to the original.



As you can see from the plot above the pictures hold up well as we reduce the amount of variance explained. At 70% the digits start becoming too blurred to tell digits apart, but at 80% the digits look similar to the original. When using the 80% dataset we have reduced the number of variables from 784 to 44 whilst getting a similar level of detail. This massively reduces the amount of computational power and storage needed for us to perform our analysis. For our analysis we are going to use 87 variables (90%) as it is a good balance between performance and accuarcy (as stated in the cumulative proportion plot).

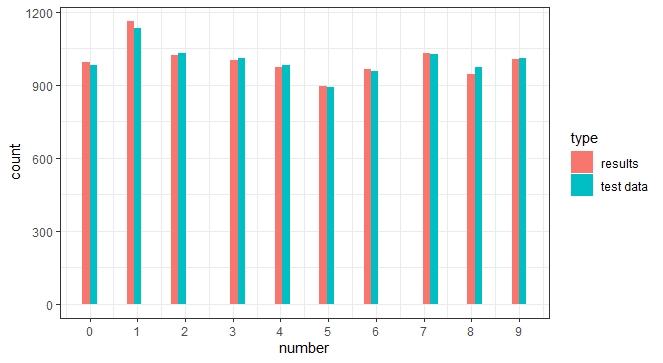
## Need to add graph about overfitting vs

Now we have reduced the dimensions of our dataset, we are now able to start analying/creating our model. We are going to be performing K-nearest neighbour (Knn) on this dataset. This is where you classify the an unknown observation based on the k nearest points. First we need to determine a suitable value for k. The best value will be the one which will give us the lowest error measurement. We need to be careful testing for this because we can overfit our test data, this means are model is to flexible and is modelling the noise in the data. When this happens the error on our training data will improve (decrease) but the actual error rate (on the test data) will be increasing. We start by testing different values of k on our training dataset. To get the most out of our training data (to help reduce the error rate) we use a method called k-fold cross validation. This is where we partition our training dataset into k segments, and use k-1 segments as training data and 1 segment as test data to make our model. We repeat this for all possible combinations (k-1) and then summerise these models to determine the error rate. The higher your k (folds) value is the more accurate your model will be. The only problem is that it will need more computing power. As a trade off for this analysis we have used 10 fold cross valiation.



From the graph above we can see the value of k (neighbours not folds) which gives the highest accuracy (lowest error) is when k=3. Hence we will use this for our model. Note that we have only been testing odd numbers for K, this is to help avoid draws when assigning new values.

When we tested our results we got an accuracy of 97.49%, this is slightly lower than we expected so there could be an element of over fitting when training our model. We can break these down further to see where these error our occuring.



Above we can see that the histogram of the test data and our predicted model is very similar, which gives us further confidence that this model is a good fit. But we can still dive deeper into this data.

Predicted Class Actual Class 0 1 2 3 4 5 6 7 8 9

0 974 1 1 0 0 1 2 1 0 0  
   
 1 0 1131 3 0 0 0 1 0 0 0  
   
 2 6 2 1004 2 1 0 0 15 2 0  
   
 3 0 1 3 975 1 14 0 8 5 3  
   
 4 0 5 1 0 953 0 5 2 0 16  
   
 5 3 0 0 7 2 865 5 1 5 4  
   
 6 4 3 0 0 2 3 946 0 0 0  
   
 7 0 15 6 0 2 0 0 999 0 6  
   
 8 3 1 2 12 4 11 5 3 929 4  
   
 9 5 4 1 7 9 3 1 3 3 973

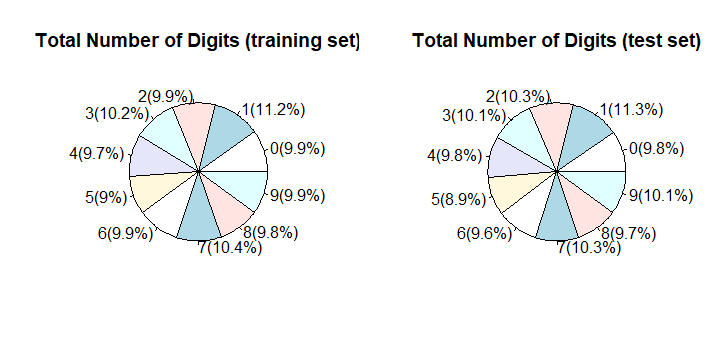
The table above breaks the values down even further to see where our model has been wrongly classifying variables. Unsuprisingly when our models wrongly classifies a observation its a digit which looks similar to another. such as 8 and 3, or 9 and 4

We are now going to use a second method to classify our MNIST dataset. Here we are going go use NaiveBayes. This is a probabilistic approach based on Bayes Theorem

P(C|D) = P(C) \* P(D|C) / P(D)

Where P(C|D) is the probability of the class (C) based on the data (D), which is what we can to find out. P(C) is the *prior*, which is the proportion of data that have class C. P(D|C) is the likelihood, this is the probability of observating data point D given the class C. P(D) is the probability of the data occuring. This is a constant across classes and is not needed to calculate the relative probabilites.

For this forumla to hold we make the assumption that features in the data are independent of each other given the class. This means each variable/dimension in the data is independent. This is often not the case, but the model can still give good predictions. In our model we know that this independence assumption isn’t true because with handwriting if one pixel has a value, the pixels next door are more likely to have handwriting in. For our model to be accurate it’s also important that training data is represenative of the overall class distribution, otherwise are P(C) (prior estimates) will be incorrect. In the graph below we can see that the prior estimates from the training dataset is close to the test data so this assumption holds.



Before we create our model, we are going to use PCA on the data and reduce the components to 90%. This has a similar effect as KNN in terms of speeding the computation up but also has an additinal effect. By only taking the first 87 PCA dimensions, the correlation between dimensions should be lower. This is because of how PCA calculates it’s principal components (note the more PCA dimensions included higher the correlation). When we created our model we used a 10-fold cross validation to make the model more compareable to the previous KNN model. This resulted in an accuracy of 88.42% (much lower than the previous 97.49%). When we look closer into the results we see that this model also mixers up similar classes (as KNN) such as 3 and 8, or 4 and 9.

Predicted Class

Actual Class 0 1 2 3 4 5 6 7 8 9

0 935 0 14 0 1 13 15 1 1 0  
   
 1 1 1080 7 16 3 6 10 3 7 2  
   
 2 16 1 887 40 9 7 12 13 44 3  
   
 3 3 0 34 870 4 35 5 16 27 16  
   
 4 1 0 15 1 871 6 10 4 8 66  
   
 5 7 0 8 58 1 765 11 4 30 8  
   
 6 8 2 22 4 10 36 874 0 1 1  
   
 7 4 16 32 4 26 13 5 868 11 49  
   
 8 14 0 18 34 9 31 5 5 846 12  
   
 9 14 3 23 9 59 19 1 31 4 846

From our analysis we can conclude from our two tested models that K nearest neighbour with value of k=3 is our best model with an accuracy of 97.49% on our test data (compared to 88.42% from Naives Bayes). When comparing the accuracy to our cross validation plot it looks like we may have overfitted this model. It may be worth trying k=5+ in future models to check this hypothesis. It is worth mentioning that we performed PCA and reduced our data to 87 principal components.

When we compare these error rates with ones already published (<http://yann.lecun.com/exdb/mnist/>) we can see the hightest accuracy rate is for a convolutional net with 99.77%, and the highest accuracy rate for a KNN method (with additional preprocessing) is 99.37%. The accuracy rate of these are a lot higher than the methods we have used, but they also have their cons. In the case of the convolution net, the computation needed to compute the model is expensive and the method used is complex and harder to explain to the laymen user. This is also similar for the KNN method, as the preprocessing used is shape context matching. Whilst the KNN method is simple (although computationally expensive) the method used to preprocess the data is complex and adds further computation.

To determine which method is best suited for our problem we have to consider a range of factors. This includes error rate (accuracy), cost (computation time) and complexity. The cost and complexity is also important becuase some models can take an extremely long time and make the models infeisable to run. Complexity is also important because depending on the method used you may need to defend and explain the method you are using to the laymend user, this can be difficult for models such as neural networks.

Some of the limits of the models we have implimented are in how we’ve trained our model. To improve this we should use the leave one out technique, but this also has the con that it is extremely computationally expensive to run. Another limitation with the naives bayes approach is the number of principal components used. This has been conditioned for the KNN rather than naives bayes, to look further into this we would want to look at correlation between each variable to calculate the best number of principal components for our analysis.

#MNIST  
source("load\_data.R")  
library(ggplot2)  
  
# Dimension reduction -----------------------------------------------------  
  
# No Scaling, this has issues as some dimensions don't have any variation  
train.pca = prcomp(train$x)  
# Following have too many dimensions to get any useful information out of  
  
eigs = train.pca$sdev^2  
#calculate the cumulative probs  
Cumulative=data.frame(  
 `Cumulative Proportion`=cumsum(eigs)/sum(eigs),  
 Dimensions=1:28^2  
 )  
# 80  
plot(x=seq(1,28^2,1),y=Cumulative$Cumulative.Proportion,type="l")  
ggplot(Cumulative,aes(x=Dimensions,y=100\*`Cumulative.Proportion`))+  
 geom\_line()+  
 geom\_hline(yintercept = 70,linetype="dashed")+  
 geom\_hline(yintercept = 80,linetype="dashed")+  
 geom\_hline(yintercept = 90,linetype="dashed")+  
 geom\_text(aes(x=26,y=80,label=paste("26")),col="red")+  
 geom\_text(aes(x=26,y=75,label=paste("↓")),col="red")+  
 geom\_text(aes(x=44,y=90,label=paste("44")),col="red")+  
 geom\_text(aes(x=44,y=85,label=paste("↓")),col="red")+  
 geom\_text(aes(x=87,y=100,label=paste("87")),col="red")+  
 geom\_text(aes(x=87,y=95,label=paste("↓")),col="red")+  
 scale\_y\_continuous(breaks = seq(0,100,20))+  
 labs(title = "PCA Dimension against proportion of varience explained",y="Cumulative Proportion (%)")+  
 theme\_bw()  
  
 ## We want the PCA eigen vectors to account for at least 80%  
# Lowest number of dimensionsf for this is 44.  
numComponents=44 #80%  
numComponents2=87 # 90%  
numComponents3=26 # 70%  
  
Xhat = t(t(train.pca$x[,1:numComponents]  
 %\*%  
 t(train.pca$rotation[,1:numComponents]))   
 + train.pca$center)  
Xhat2 = t(t(train.pca$x[,1:numComponents2]  
 %\*%  
 t(train.pca$rotation[,1:numComponents2]))   
 + train.pca$center)  
Xhat3 = t(t(train.pca$x[,1:numComponents3]  
 %\*%  
 t(train.pca$rotation[,1:numComponents3]))   
 + train.pca$center)  
  
# compare first few numbers against predicted version  
train2=list()  
Xhat.2=list()  
Xhat2.2=list()  
Xhat3.2=list()  
for(i in 1:10){  
 train2[[i]]=matrix(train$x[i,], nrow=28)[,28:1]  
 dimnames(train2[[i]])=list(x2=1:28,y2=1:28)  
 train2[[i]]=as.data.frame(as.table(train2[[i]]))  
 train2[[i]]$number=train$y[i]  
 train2[[i]]$id=i  
   
 Xhat.2[[i]]=matrix(Xhat[i,], nrow=28)[,28:1]  
 dimnames(Xhat.2[[i]])=list(x2=1:28,y2=1:28)  
 Xhat.2[[i]]=as.data.frame(as.table(Xhat.2[[i]]))  
 Xhat.2[[i]]$number=train$y[i]  
 Xhat.2[[i]]$id=i  
   
 Xhat2.2[[i]]=matrix(Xhat2[i,], nrow=28)[,28:1]  
 dimnames(Xhat2.2[[i]])=list(x2=1:28,y2=1:28)  
 Xhat2.2[[i]]=as.data.frame(as.table(Xhat2.2[[i]]))  
 Xhat2.2[[i]]$number=train$y[i]  
 Xhat2.2[[i]]$id=i  
   
 Xhat3.2[[i]]=matrix(Xhat3[i,], nrow=28)[,28:1]  
 dimnames(Xhat3.2[[i]])=list(x2=1:28,y2=1:28)  
 Xhat3.2[[i]]=as.data.frame(as.table(Xhat3.2[[i]]))  
 Xhat3.2[[i]]$number=train$y[i]  
 Xhat3.2[[i]]$id=i  
}  
train3=do.call(rbind,train2)  
Xhat.3=do.call(rbind,Xhat.2)  
Xhat2.3=do.call(rbind,Xhat2.2)  
Xhat3.3=do.call(rbind,Xhat3.2)  
train3$x2=as.integer(train3$x2)  
train3$y2=as.integer(train3$y2)  
Xhat.3$x2=as.integer(Xhat.3$x2)  
Xhat.3$y2=as.integer(Xhat.3$y2)  
Xhat2.3$x2=as.integer(Xhat2.3$x2)  
Xhat2.3$y2=as.integer(Xhat2.3$y2)  
Xhat3.3$x2=as.integer(Xhat3.3$x2)  
Xhat3.3$y2=as.integer(Xhat3.3$y2)  
Xhat.3$pred="80%"  
Xhat2.3$pred="90%"  
Xhat3.3$pred="70%"  
train3$pred="Real"  
train4=rbind(train3,Xhat.3,Xhat2.3,Xhat3.3)  
  
ggplot(train4,aes(x=x2,y=y2,fill=Freq))+geom\_tile()+facet\_grid(id~pred)+scale\_fill\_gradient(low="white",high="black")+labs(x="",y="",fill="Colour",title="Dimensionaly reduced MNIST dataset",subtitle = "Reconstructed for the first 10 values by variance explained")  
  
  
# compare the average against the average predicted version (0-9)  
  
  
# 3) K- nearest neighbour (KNN) -------------------------------------------  
library(class)  
  
test.pca = prcomp(test$x) # Reduce test data into 44 dimensions  
  
knn\_train=as.data.frame.array(train.pca$x[,1:numComponents2]) # create train dataset with 44 dimension reduced variables from PCA   
#knn\_test=as.data.frame.array(test.pca$x[,1:numComponents]) # creat test dataset with 44 dimension reduced variables  
knn\_test=predict(train.pca,newdata = test$x)[,1:numComponents2]  
  
  
# KNN with Caret ----------------------------------------------------------  
 library(caret)  
 train\_control<- trainControl(method="cv", number=10, savePredictions = TRUE)  
   
 knn\_train$y=as.factor(train$y)  
   
 model<- train(y~., data=knn\_train, trControl=train\_control, method="knn",tuneGrid = expand.grid(k = seq(1,21,by=2)))  
   
 plot(model)  
   
 resultscv=predict(model,newdata=knn\_test,type="raw")  
 100\*sum(as.numeric(as.character(resultscv))==test$y,na.rm=T)/length(resultscv==test$y)  
 table(`Actual Class` = test$y, `Predicted Class` = resultscv)  
   
   
 # Plot to see if histogram of results and real data is similar  
 hist.knn=data.frame(  
 number=c(as.numeric(as.character(resultscv)),as.numeric(test$y)),  
 type=c(rep("results",length(resultscv)),rep("test data",length(test$y)))  
 )  
 ggplot(hist.knn,aes(number,fill=type))+geom\_histogram(position="dodge")+theme\_bw()+scale\_x\_continuous(breaks = 0:9)  
 #+facet\_grid(~type)  
   
   
  
# Next part.... -----------------------------------------------------------  
  
   
library(e1071)   
yTable=table(train$y)  
percentage=round(100\*yTable/sum(yTable),1)  
labels=paste0(row.names(yTable),"(",percentage,"%)")  
  
yTable1=table(test$y)  
percentage1=round(100\*yTable1/sum(yTable1),1)  
labels1=paste0(row.names(yTable1),"(",percentage1,"%)")  
par(mfrow=c(1,2))  
pie(yTable1,labels=labels1,main="Total Number of Digits (test set)")  
pie(yTable,labels=labels,main="Total Number of Digits (training set)")  
  
train2=train  
train2$y=as.factor(train2$y)  
train2[['n']]=NULL  
train3=as.data.frame(train2$x)  
model.naiveBayes <- naiveBayes(train2$y ~ ., data=train3)   
summary(model.naiveBayes)   
  
prediction.naiveBayes=predict(model.naiveBayes,newdata=test$x)  
table(`Actual Class`=test$y,`Predicted Class`=prediction.naiveBayes)  
  
  
Accuracy=100\*sum(as.numeric(as.character(prediction.naiveBayes))==test$y,na.rm=T)/length(prediction.naiveBayes==test$y)  
  
  
## Now with PCA  
## Need method to visualise the predictions?? If this is possible.  
  
# redoing ~NaiveBayes -----------------------------------------------------  
  
train.pca = prcomp(train$x)  
  
train2=train  
train2$y=as.factor(train2$y)  
train2[['n']]=NULL  
train3=as.data.frame(train.pca$x[,1:numComponents2])  
#test.pca = prcomp(test$x)  
test.pca=predict(train.pca,newdata = test$x)[,1:numComponents2]  
  
model = train(train3,train2$y,'nb',trControl=trainControl(method='cv',number=10))  
  
prediction.naiveBayes=predict(model$finalModel,test.pca)  
  
table(`Actual Class`=test$y,`Predicted Class`=prediction.naiveBayes$class)  
#prediction.naiveBayes  
  
Accuracy=100\*sum(as.numeric(as.character(prediction.naiveBayes$class))==test$y,na.rm=T)/length(prediction.naiveBayes$class==test$y)