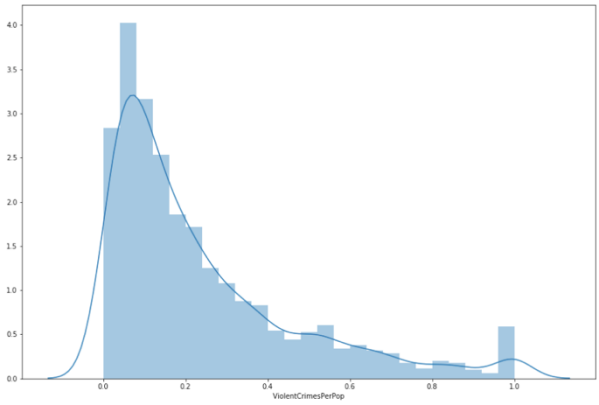
Catherine Xu
n10684069

# Assignment 1A

## Problem 1

### Introduction

In this problem, we are given a data set that contains socio-economic data from the 1990 US census. We are given the number of violent crimes per capita, and will proceed with exploring the link between socio-economic factor and crime rates. The same data set is trained with three different regression models. After training the models, in which we find the best values for the hyper parameter in lasso and ridge regression, we will calculate the RMSE for each and compare. RMSE puts more weight on large errors and is a simple method for studying the error in our trained model on the test set.
RMSE calculates the root of the mean square error between the predicted and the actual value, and the closer to zero the better.




Just to get an understanding of how the violent crime rates per population are distributed we plot the values. On the figure to the right y-axis doesn't really make sense, but we only need to look at the x-axis. As we can see the crime rate is between 0 and 1, and mostly around 0.0-0.3.

Most of the code in this problem is borrowed from Week 2 Example 2 as the example trains similar models.

### Data cleaning and removal

Before building our model, we need to clean out some data. As given in the exercise description we disregard the first five columns (state, county, community, communityname string and fold) as we don't wish that they have an effect on our models. There are many entries with a question mark, indicating that the given factor is unknown for the given crime rate.
In order to determine whether to remove all columns or rows that contain a question mark, I trained a Linear Regression model for both cases and looked at the RMSE.
When removing the rows that contain a "?", train the model, and calculate the RMSE we get a value of 0.2086, while removing the columns by the same criteria yields an RMSE of 0.1429. Therefore, the last step in the data cleaning process is to remove all columns with a question mark. Further, the covariates are isolated in a variable X and the response variable isolated in the variable Y. Scikit.learn provide the function train_test_split, and we divide the data set into training, validation and testing; $X_{train}$, $Y_{train}$, $X_{val}$, $Y_{val}$, $X_{test}$, $Y_{test}$ with the ratios 0.7, 0.15 and 0.15. To be able to compare the three models, we work with the same split, but the validation set isn't used for the linear model as no parameters need to be tuned.

## Linear Model

The Linear model is trained by using Scikit.learn's LinearRegression with X_train and Y_train. The regression will represent a model with the best fit for the training set.

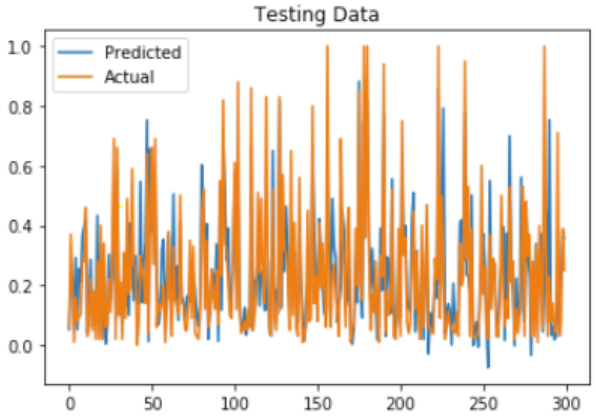
| | Coefficient |
|---|---|
| **population** | 0.204987 |
| **householdsize** | 0.015681 |
| **racepctblack** | 0.215170 |
| **racePctWhite** | -0.022614 |
| **racePctAsian** | -0.013943 |
| **...** | ... |
| **PctSameState85** | 0.010188 |
| **LandArea** | 0.041337 |
| **PopDens** | -0.010055 |
| **PctUsePubTrans** | -0.053982 |
| **LemasPctOfficDrugUn** | 0.015722 |

99 rows × 1 columns

| | Actual | Predicted |
|---|---|---|
| **1746** | 0.06 | 0.052482 |
| **1556** | 0.37 | 0.239081 |
| **56** | 0.22 | 0.230586 |
| **1740** | 0.01 | 0.052856 |
| **913** | 0.16 | 0.292754 |
| **...** | ... | ... |
| **428** | 0.71 | 0.181835 |
| **1933** | 0.03 | 0.064259 |
| **453** | 0.07 | 0.225187 |
| **1389** | 0.39 | 0.381687 |
| **1351** | 0.25 | 0.358451 |

300 rows × 2 columns

Now let's study the tables above. The left table are the coefficients for the covariates the regression model is defined by. The preview values are fairly small, which is a good sign as no factor will yield a huge change in the y-value with a small change in the x-value. The right table compares the actual response variable with the response variables predicted by our model for the training set. The result seems fairly okay.




In order to study the predicted value vs the actual value, we calculate the predicted value of X_test and compare with Y_test. It's difficult to see much from the plot, but the blue and orange lines seem quite overlapping.

Linear $RMSE = 0.14289$

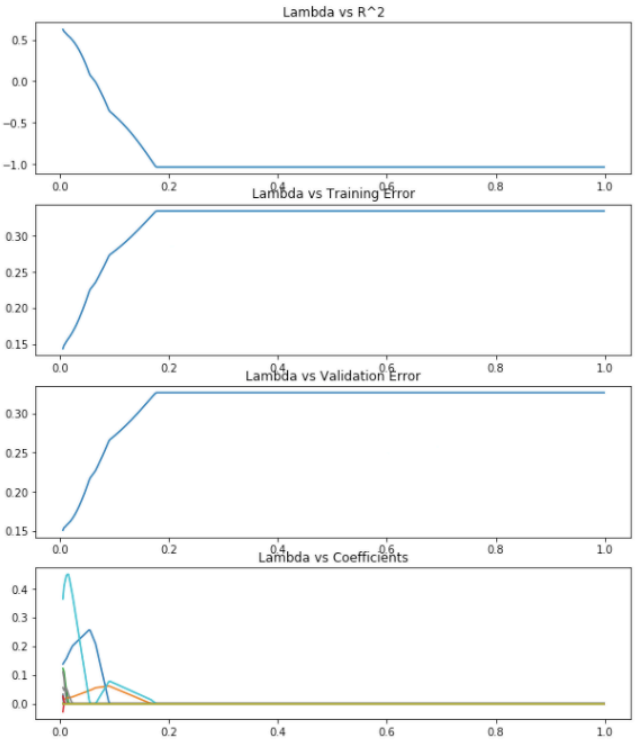
## Lasso regression

Lasso regression is essentially a linear regression with a L1 normalization. This penalty term is a sum of the absolute values of the coefficients.

To train a Lasso model, we need a value for $\lambda$. To find the optimal one we loop through values between 0.005 and 1, with a step size of 0.002, For each value of $\lambda$ we fit a Lasso regression model (scikit.learn) and calculate and store the r^2, the coefficient, training accuracy and validation accuracy in arrays. For this we use the training set for training the model and the validation set for tuning the model by identifying the $\lambda$ that would yield the smallest errors between predicted and actual values.

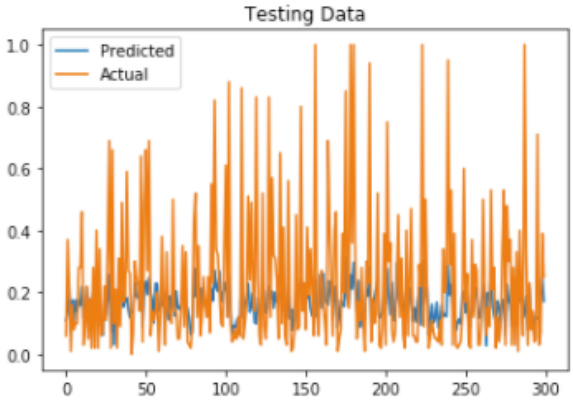
Catherine Xu
n10684069

In the plots to the right we can see how R^2, training error, validation error and the coefficients react on different λ. In general, a high R^2 value is good, and we observe that it decreases when λ grows. Both training and validation error increase as λ grows. Most of the coefficients are pushed to zero pretty fast, but there are four that take a little longer.
All this speaks in favor that a lower value of λ would yield less error.




We can extract the best λ by finding the lowest validation error and then look up the corresponding alpha, by index, in the arrays we saved. For this model, the best λ = 0.05.

Similarly, to the Linear model we plot the predicted values of X_test and the actual values; Y_test. This graph looks a bit odd as it to me seem like the predicted value isn't very overlapping with the actual.

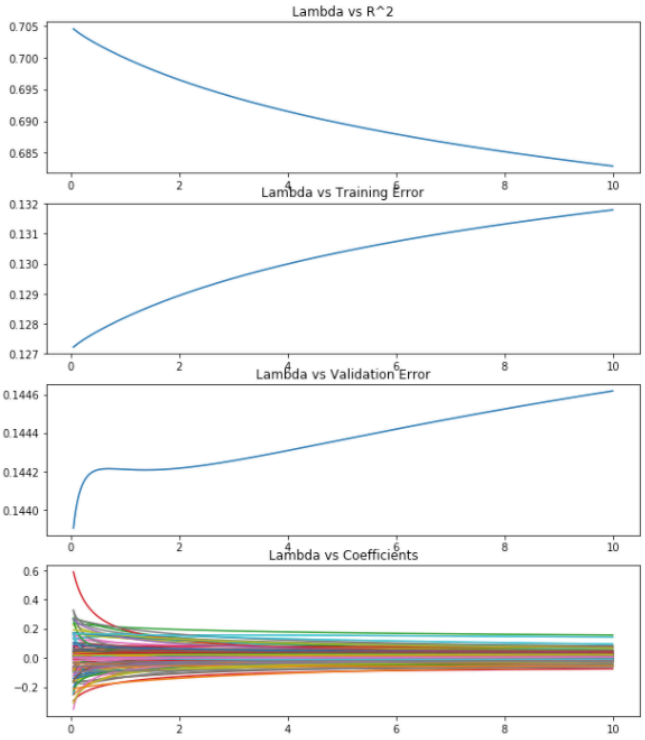



Lasso $RMSE = 0.15056$, which is surprisingly lowe.

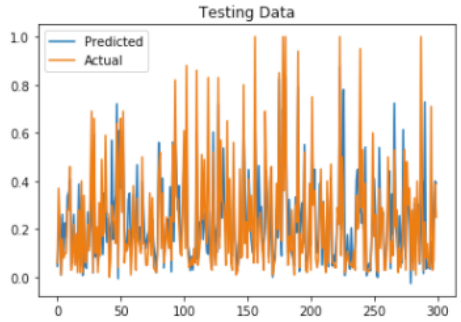
## Ridge Regression

Ridge regression is pretty much the same as Lasso regression only the penalty term is an L2 norm, which takes the sum of all the squared coefficients.
The value of λ is found in the same way as in Lasso regression, and we plot the calculated values of R^2, coefficients, training and validation error. The interval of λ we are using now is from 0.05 to 10 with a step size of 0.02. It is bigger because Ridge regression will not push coefficients to zero, but it will eventually push them close to zero, and we wish to see this graphically.

To the right we observe that changes are generally less dramatic compared to Lasso regression when the epochs increase. The trend is smaller R^2, higher training and validation error when $\lambda$ grows. The coefficients are slower than Lasso regression to approximate 0, but it's important to know that the coefficients in ridge regression never actually becomes zero.




Once again we take a look at the predicted values for X_test compared to the actual; Y_test. In this model it seems like the predicted values follow fairly close to the actual values.




Ridge $RMSE = 0.14462$

## Comments

```
Linear Model, Test RMSE: 0.14289323350782915
Lasso Model, Test RMSE:  0.1505618424614521
Ridge Model, Test RMSE:  0.14461805417027723

Linear Model, Test R^2: 0.6019091036783852
Lasso Model, Test R^2:  0.5580341029606346
Ridge Model, Test R^2:  0.5922406334524399
```

It seems like the RMSE of the linear model is lower than both Lasso and Ridge regression. This contradicts general trends as Lasso regression push several terms to zero. However the the $R^2$ value confirms that the Linear model has a higher accuracy.
I wasn't very pleased with this result, and therefore I ran the entire code on a data set that removes the rows that contain "?" instead of the columns. The RMSE we'd get from that is:

```
Linear Model, Test RMSE: 0.20863372760585916
Lasso Model, Test RMSE:  0.1597096931157633
Ridge Model, Test RMSE:  0.16123613489166816
```

In this case we do see that Lasso and Ridge regression performs better than the Linear. However, the overall RMSE is lower if we remove the columns. With this observation, it could be that the model I have presented above (removing the columns) discard important columns for the set, and we might lose some important factors. As we are looking at statistical trends, it's not that important to keep all entries and represent all individuals, but we can lose important information if we delete factors that affect the crime rate. The

decision to remove columns just based on the RMSE values for linear regression is maybe not good enough. On a second thought it could be better to check some correlations before choosing one. I could also have added a threshold and only deleted columns that have more than a threshold of "?", in that way I could've saved some important values and cleaned the set at the same time.

## Problem 2

### Introduction

Problem 2 introduces a multivariable classification problem, where different factors classify land into different classes. The data set can be classified into one out of Sugi forest, Hinoki forest, Mixed deciduous forest and Other non-forest land. This problem is solved with two different types of classifications; K-Nearest Neighbor Classifier and an ensemble of binary classifiers. We use scikit.learn's KNeighborsClassifier, Suppor Vector Machine and OneVsRestClassifier. To evaluate findings and the models, we use the function eval_model from week 3 example 2. This provides us with information about model accuracy and plots a confusion matrix, which are useful tools to evaluate the performance.

### Data split

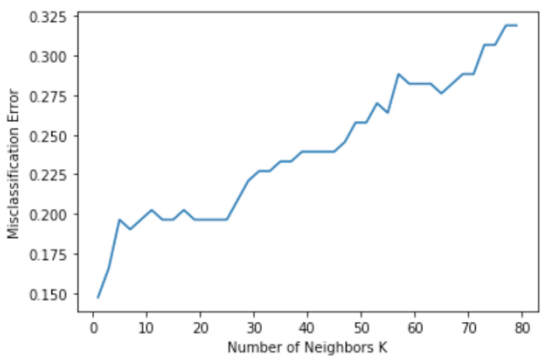
We are provided predefined set of training and testing data. However, in order to tune hyperparameters, we need a validation set. The training set has 198 entries while the testing set has 325. Therefore, we split the testing set such that the first 162 entries become the validation set and the remaining becomes our testing set. The ratios do not comply with general conventions, but as this is given by the exercise, we'll let it be. The column 'class' is defined as the response variable. This data split is used through all three classifiers.

### K-Nearest Neighbor Classifier

When training a K-Nearest Neighbor Classifier, the number of neighbors (k) is a hyper parameter that needs to be attributed. A large value of k is more resilient to outliers and results in lower variance but higher bias. Some classes will be classified as something else than what it really is because we lack enough points to represent all. A small value of k gives a very flexible fit, low bias but high variance because small changes in the training set has a big effect on the test set.

An option to find the value of k which would yield the highest accuracy is to iterate through a range of potential values, build the model and extract the value of k that gives highest accuracy. We loop through the range 1-80 with a step size of 2, build a KNeighborsClassifier, calculate the accuracy, and save the value of k and the value of the accuracy in two independent arrays. From there we can calculate the misclassification error (1-accuracy), and we plot the k values on the x-axis and the MSE on the y-axis by extracting values based on indices. The lowest MSE reveals the best k in this plot. The function yields k=1 in order to reach the highest accuracy.

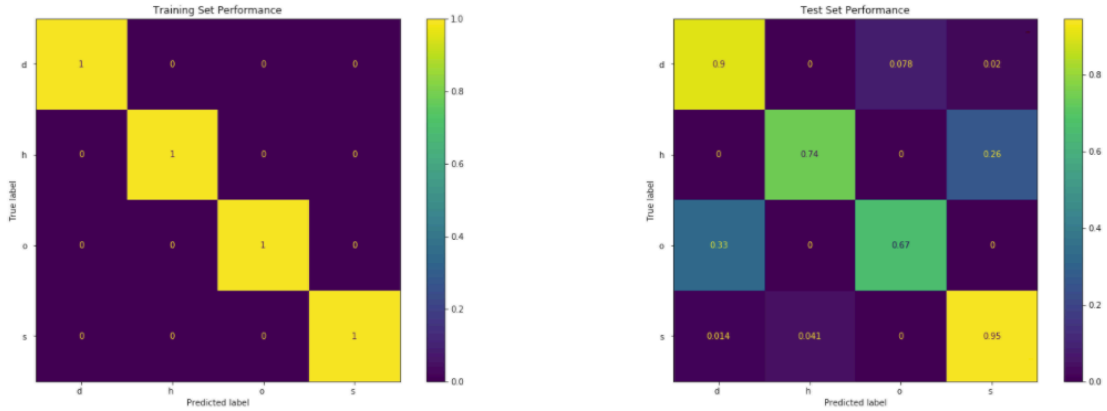
```
The optimal number of neighbors is 1
```



The accuracy is calculated by using the model trained by a training set to calculate predicted values of the validation set, and compare the predicted values with the actual response variable of the validation set. We had to use a validation set to tune the parameter k. If we had used the training set, we would've ended up with k=1 because there is always one that is nearest and has the correct class label, the example itself.
This is an effective way of finding the optimal number of k as we build the model several times and evaluate it on a validation set without performing each iteration manually. However, which k we end up with can depend on the range we choose to try k values in. If this range doesn't cover the actual value, the globally optimal k will not be found. As we can see in our plot, it seems like the MSE is generally increasing when k increases.
There are of course other ways to find the optimal k, either by trying manually or perform a k-crossfold validation.

We run eval_model on our test set to evaluate the performance of our model with k = 1. It predicts farily well with an accuracy of 0.87654.

```
Test Accuracy: 0.8765432098765432
```



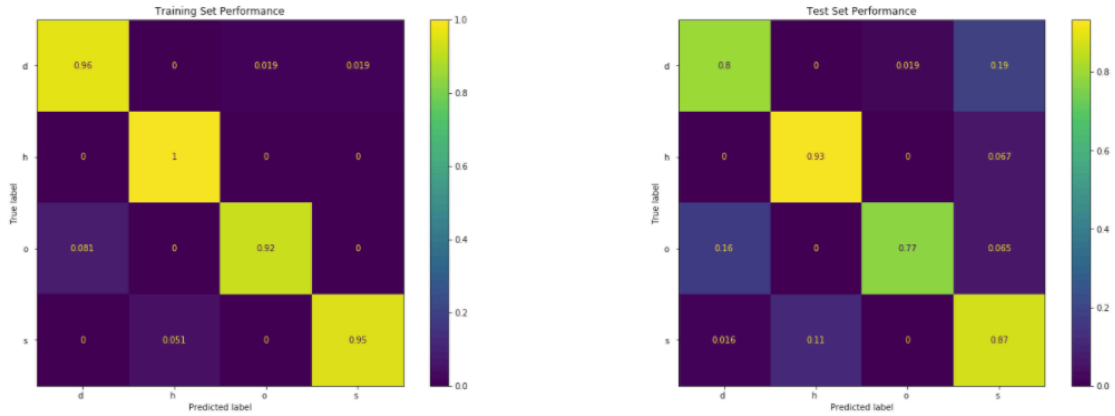

## Ensemble of binary classifiers

**One vs One**

Scikit.learn's Support Vector Machine automatically runs One vs One Classifier if it's exposed to a multiclass dataset. When initializing an SVM we can define a parameter, C, that decides how much we wish to avoid misclassifying the training set. By default this value is set to 1, but a higher value of C gives the hyperplane a smaller margin, while a lower value of C gives the hyperplane bigger margin. When creating a hyperplane we wish to define it to have the largest margin and the highest prediction accuracy, but this is a trade-off.

The SVM is built with the training set, and evaluated to the validation set in order to find a value for C that predicts the validation set the best. For an arbitrary C = 3, this is the result of eval_model with the validation set.

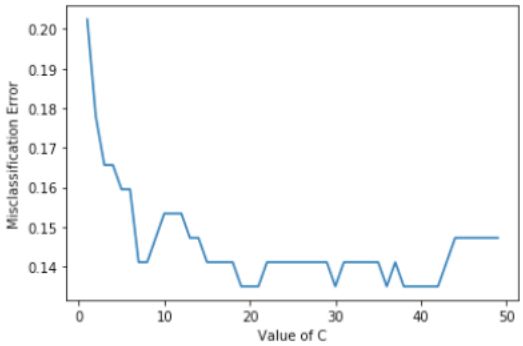
```
Test Accuracy: 0.8343558282208589
```




In the same manner as we found the optimal k for the K-Nearest Neighbor Classifier we iterate through a range of C values (1-50) and extract the C value that yields the highest accuracy. Kernel is defined as RBF because it increases accuracy a little bit.

From the following plot we get the optimal C, which is 19.

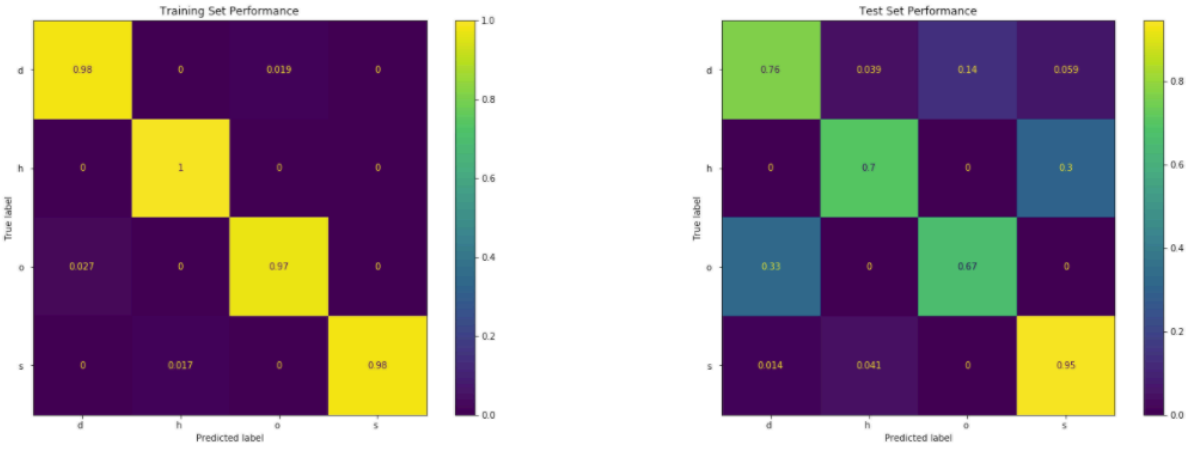
```
The optimal value of C is 19
```




An observation from running eval_model with C = 19 on the validation set once more is that it gives a model accuracy of 0.85276, which is around 0.02 better than using the arbitrary value of C = 3. It can therefore be discussed whether using such a big value of C and hence reducing the margins of the hyperplane is worth getting a better performance of 2%.

We then execute evaluation of the model by running eval_model on the test set with C = 19.

```
Test Accuracy: 0.8271604938271605
```

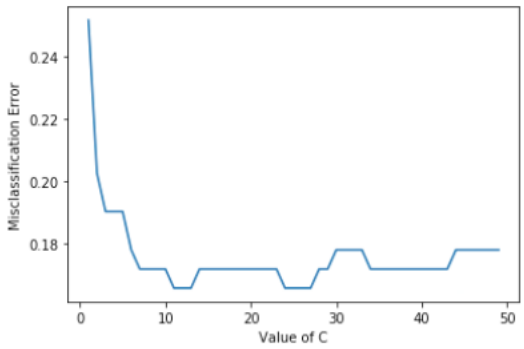



**One vs All**

We train a One vs All classifier by using scikit.learn's OneVsRestClassifier, which takes in an SVM. This is because a One vs All classifier is essentially just hyperplanes isolating individual classes from the rest. First we create the model by inputing a SVM with class_weight = 'balanced' and train it without training set. Then we evaluate the model with the validation set with eval_model. The resulting test accuracy is 0.82822 with C = default value = 1.

Further we run through the same algorithm for finding the optimal value for C in the SVM.

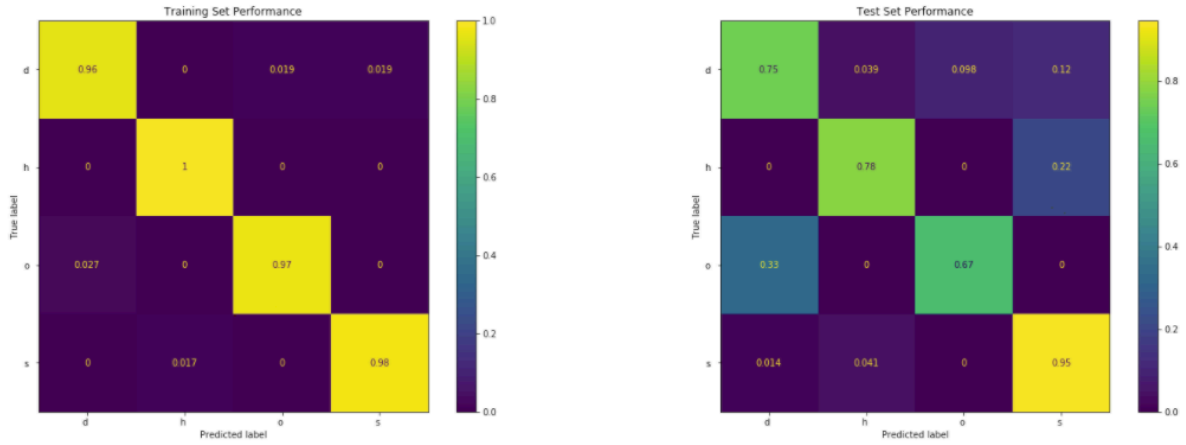
```
The optimal value of C is 11
```




As we can see, the optimal value for C is 11. If we run eval_model with the validation once more with the parameter C = 11, our model yields an accuracy of 0.83436. This is an improvement of around 0.5%. We can again raise the question if it's worth it to reduce the margin for small improvement of model accuracy.

Let's now evaluate the model by running eval_model on the test set. The accuracy is actually slightly higher than for the validation set.

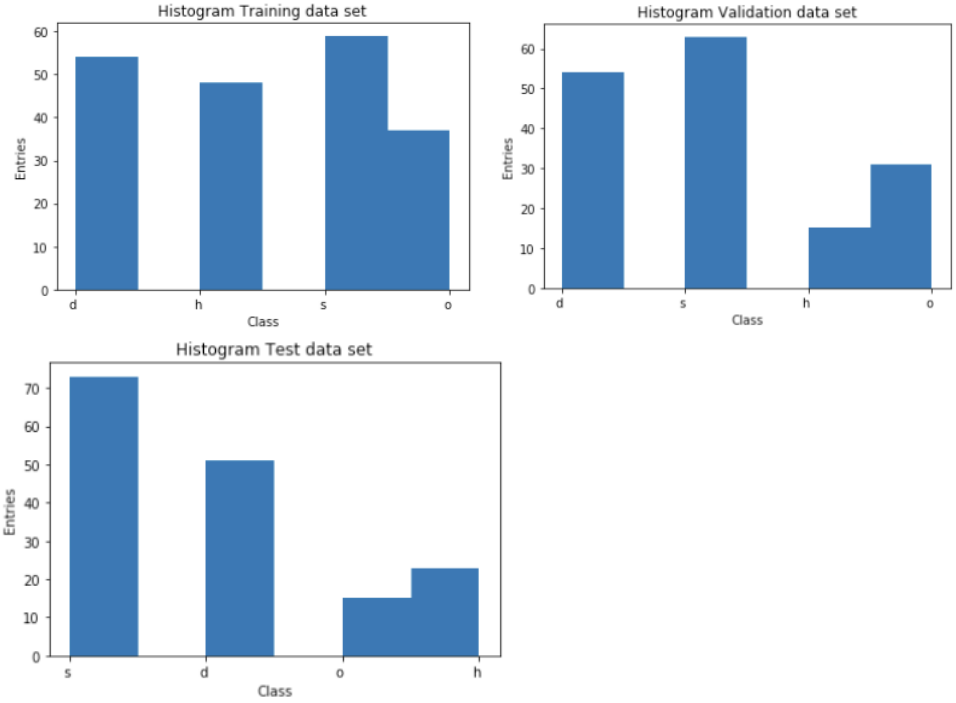
```
Test Accuracy: 0.8333333333333334
```




## Comments

The One vs One classifier and the One vs All classifier all aims towards the same, using binary classification techniques to classify multiclass variables. One vs One classifier will run a binary classification on each unique pair of the classes we have. Since we have four classes, we get 6 binary classifiers. A One vs All classifier would however isolate create a binary classifier for each isolated class from the rest of the classes. With the classes we have, we'd get 4 binary classifiers. With more classes we'd get a bigger difference between the number

of classifiers necessary. If there is a big class imbalance in the latter one, the smaller group can be completely overpowered by the others and practically not exist in the model.
As we have seen from the evaluations, the accuracy for the two models are very close to each other. This is can be explained by a very low class imbalance as we can see here:




# Problem 3

## Introduction

We are provided with two datasets already divided into training and testing of Street View House Numbers. The data is divided into 10 classes, where training set has 100 samples and the testing set has 1000 samples of each class. The solution is divided into three parts: a basic deep network, a deep network with data augmentation and then fine tuning an existing model on our dataset. We use Tensorflow and Keras to perform a classification.

## Data importation

The data is imported as Matlab-files and used in the predefined sets, and we split the sets into $X_{train}$, $Y_{train}$, $X_{test}$, $Y_{test}$. We check the ranges, and modify indices of Y-sets as they're adjusted to Matlab standards. The data is in a shape that does not fit keras' input, and we reshape the training and testing X-sets from (32, 32, 3, 1000) to (1000, 32, 32, 3).
Further on we divide the same two sets on 255 to get our data range in the interval [0,1] and hence our model will converge faster.

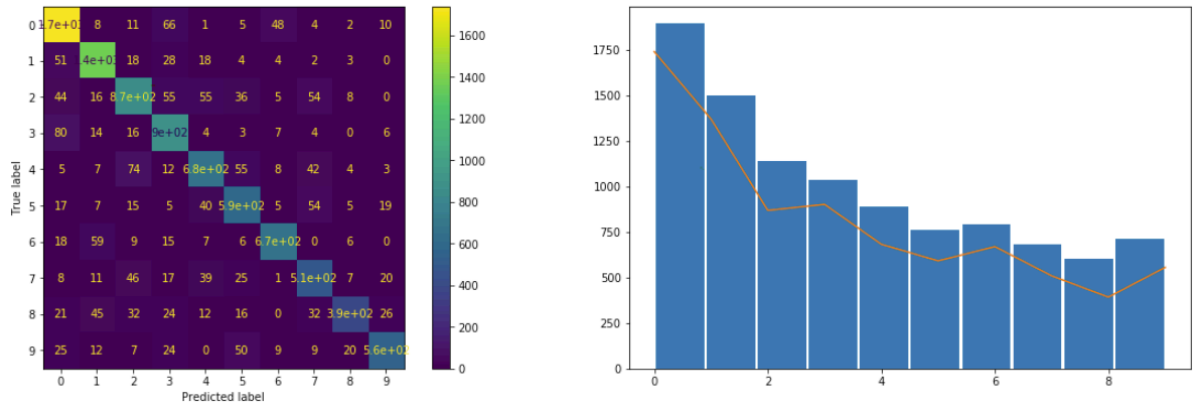
## Train model from scratch

Building our model uses the build_model function from Practical 4, and takes X_train as input with the shape (32, 32, 3). The number of classes is defined when initializing the model. We run three pairs of convolution filters (from 8 to 32), each followed by a batch normalization, an activation, spacial dropout and max pooling (except for the last one). The convolution layers learns features of the image and are therefore important. We start with smaller filters such that the model can learn the most important characteristics, and then

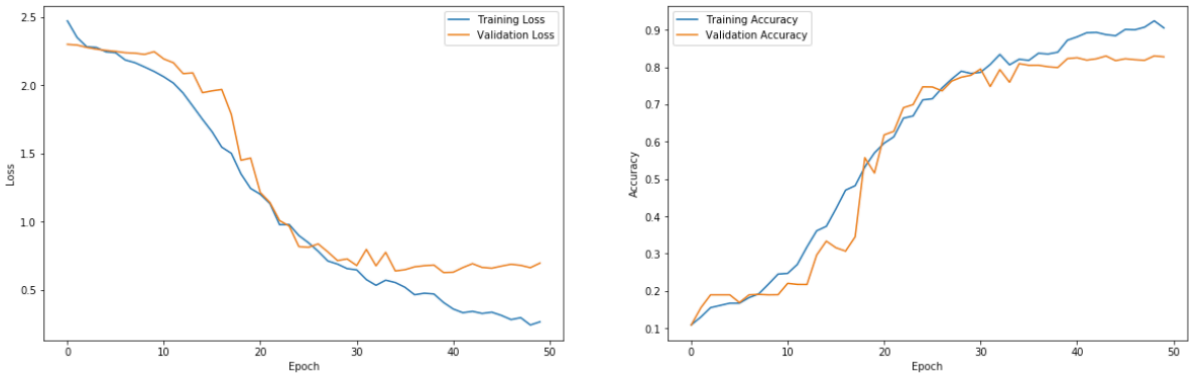
increase to learn more details. Batch normalization allows for higher learning rates, relu is our activation function to move away from only linear mappings, spacial dropout and max pooling is used to avoid overfitting.

We then train the network by choosing epoch = 50 and batch size = 50. The batch size is chosen relatively small to our data set as large batch sized aren't the best at generalizing. The number of epochs is chosen by trying a few different and see if our model converges with the batch size. It could be that these values aren't the most optimal in order to see a global convergence. However, when evaluating the model to the testing set we get an accuracy of 0.8275, which isn't too bad. On the figure to the right it also seems like the prediction follow the class distribution fairly well. One consideration is that the special dropout layer isn't necessarily needed, and the effect on the performance could be go both ways.

```
10000/10000 - 3s - loss: 0.6957 - accuracy: 0.8275
Test loss: 0.695667024731636
Test accuracy: 0.8275
```




Looking at the training and validation loss in the figure to the left below, we can see that the training loss is trending towards becoming quite a bit lower than the validation loss. On the figure to the right, it looks like the training accuracy is becoming increasingly higher than the validation accuracy. These are signs of overfitting[1].



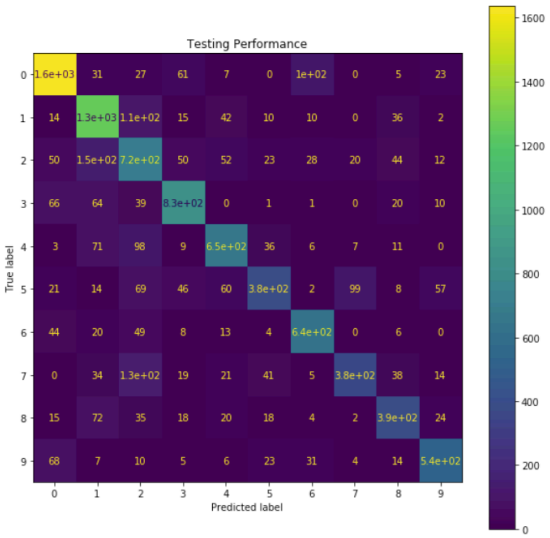

## Data Augmentation

In this part I retrained the model, but this time I used some data augmentation through Keras' library, ImageDataGenerator. This can be everything from grayscale to flipping and rotating. I chose to follow the augmentations that were done in Practical 4, which are rotate, horizontal and vertical shift, shear transform and zoom. However, I have chosen to not flip

[1] https://towardsdatascience.com/handling-overfitting-in-deep-learning-models-c760ee047c6e Downloaded 22nd April 2020.

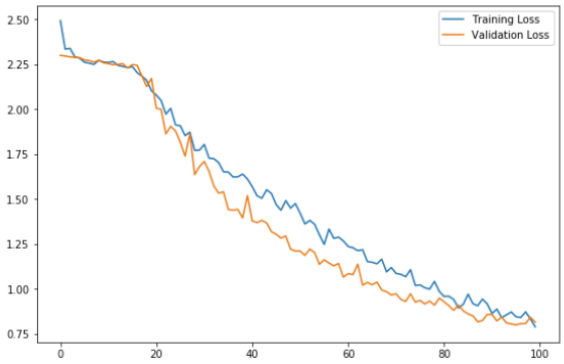
the image as there are numbers that can change meaning, for example 9 and 6. It might not be necessary to use that many different, however I went along with this, built my model and trained it with 10 classes, a batch size of 50 and 100 epochs. However, creating bigger differences in the data in the training set for the model should make the model predict better as the model is trained on a virtually bigger data set.

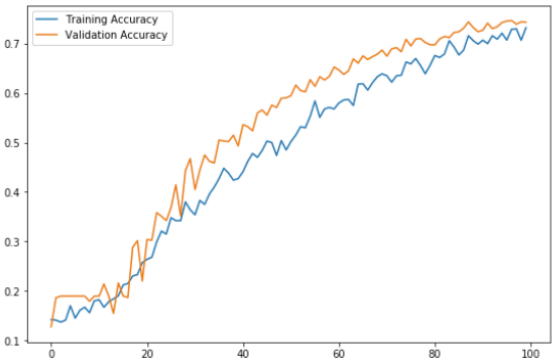
```
10000/10000 - 3s - loss: 0.8160 - accuracy: 0.7435
Test loss: 0.8160149386405945
Test accuracy: 0.7435
```




After building and training the model, we evaluate the accuracy of the model with our test set. As we can see from the figure to the right, the accuracy has actually dropped with 8%. It could be that the quality of the images in the training set sunk after using many augmentations, and testing on a set with better quality images resulted in worse accuracy. This needs further examination, but due to high running times on these models I did not try several different varieties.

Looking at the figure below, it seems like our model is catching more noise as the graphs are less smooth. It could be that the model isn't generalizing as well. However, comparing to our previous model, there's a smaller difference between the training and the validation loss, and the validation accuracy is actually better than the training accuracy after 20 epochs. The latter could be because the use of a dropout function when building and training the model, in which some information is lost. We have reduced our overfitting problem.



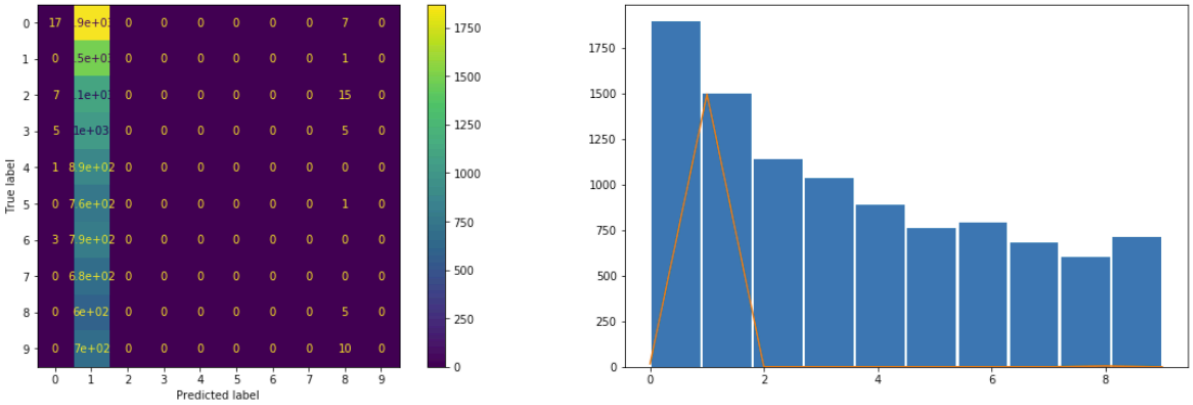

## Fine Tune Existing Model

From resources in Blackboard we import the already trained Residual Network model ResNet/resnet_v2_cifar_big.h5. This was chosen because it's based on CIFAR, which has the same dimensions on the input variables. It has a few more groups of convolution, batch normalization and activation than the previous model we trained, but it doesn't have convolution filters in pairs.

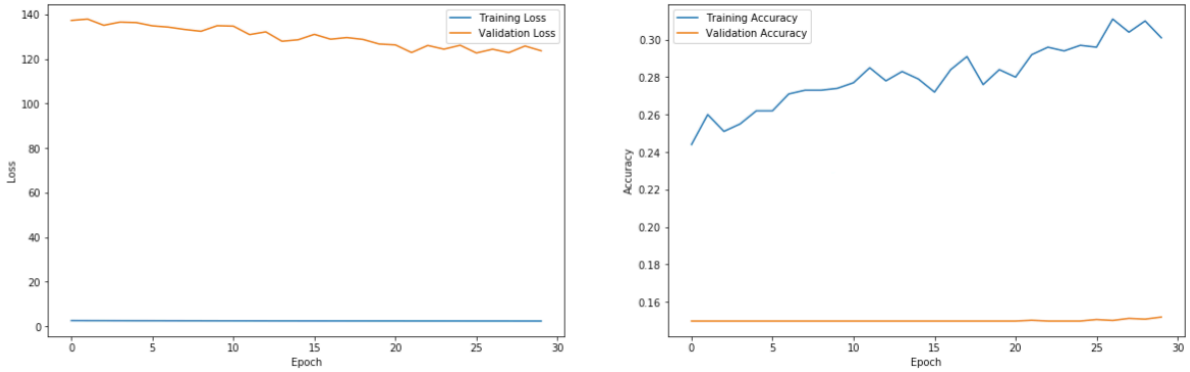
We wish to train this existing model on our data set to 10 classes. In order to retrain less, we begin with only retraining the last layer, a Dense layer. We set all the layers except for the last one to false on the Boolean parameter trainable. The loss function is set to be categorical cross entropy because it is used for multiclass classification, and the batch size

remain at 50 while epochs is set to 30 because the accuracy was very bad with lower epoch, and the model requires quite a bit of time to train.
We evaluate the model with the our test set, and as we can see, it's not very good at predicting. It essentially almost only predicts two classes and the accuracy is very low.

```
10000/10000 - 46s - loss: 123.6784 - accuracy: 0.1518
Test loss: 123.67836230468750
Test accuracy: 0.1518
```

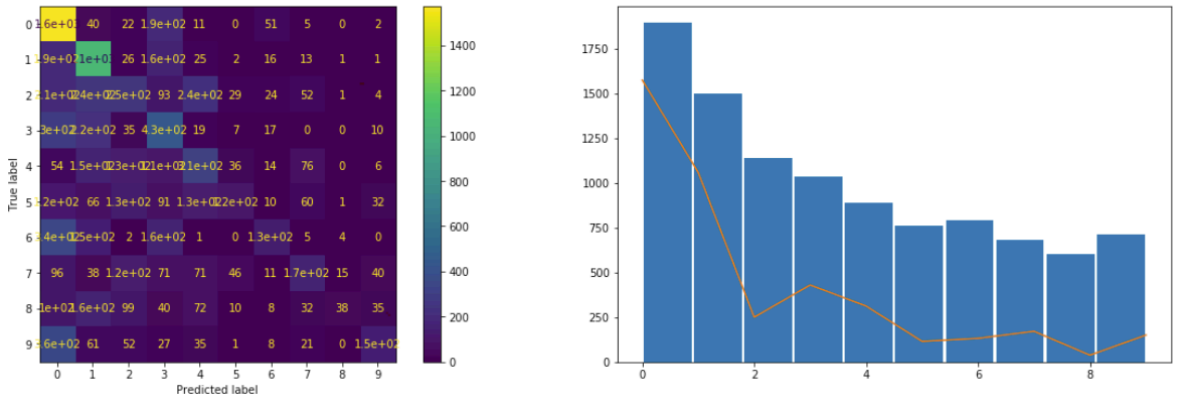



Looking at the plots below, it's obvious that the model is not performing well on the testing set with a high loss and low accuracy.

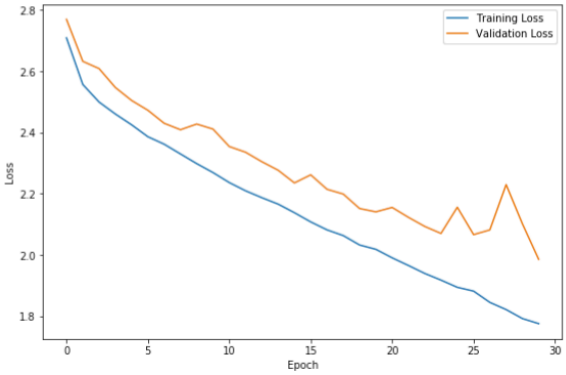


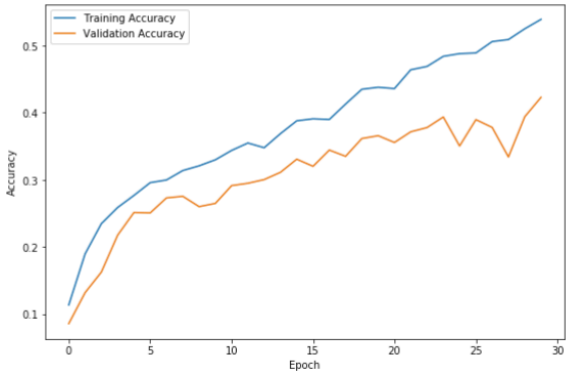

We can try to improve this by retraining all layers in our model to our data set. This should increase the accuracy. The steps and variables are set to the exact same, but we skip the part where we set the trainable parameter to False.
When evaluating the model on out testing set we see a big improvement in accuracy, and we see that more images are classified to more classes.

```
10000/10000 - 45s - loss: 1.9853 - accuracy: 0.4230
Test loss: 1.9853471782684327
Test accuracy: 0.423
```



The last thing to look at is the accuracy and the loss. From the figure below we see a big improvement, the model predicts a lot better and the validation loss is lower. It is not completely clear if the model has converged or not, but due to high running times I had to limit the number of epochs to 30. We could, however, speculate that it's trending towards validation accuracy flattening out while training accuracy is still high, and training loss getting increasingly smaller than validation loss. Further adding data augmentation could improve this potential overfitting.




As a last word, the training set was indeed small, especially compared to the size of the testing set.