Matthew Mcwaters (matt.mcwaters@mail.utoronto.ca)
Ilan Karasik (ilan.karasik@mail.utoronto.ca)
12/10/2015

# CSC411H1 Assignment 3

## Cross Validation

Before we started testing any models, we decided that it was essential that we have implement proper cross validation. We first inspected the dataset and noticed that it was ordered by identity which led to our first intuition; we felt that simply taking a slice sized at most one third the size of the labeled data, from anywhere in the set would suffice. Our results with testing an out of the box Knn model showed us that this would not suffice; we noticed each slice was giving us largely different success rates.

Our next intuition was that ordering by identity might not be enough, so we decided to split based on labels, keeping an even spread of the labels in both the validation and training set. We used a function in Scikit-learn which lets us spread based on given labels in a balanced manner, just as we wanted. This however led to equally uneven classification rates amongst slices. Thus, we decided to compromise and attempt to order both by identity and label as best we could. We simply used a number of matrix manipulations on the labels, identities and image arrays (which were reshaped from (32x32) to (1x1024)) to give us one matrix per label (7) with every face and identity for that face.

We ordered each label array by identity and then looped one by one over every array putting one augmented face back into a master array. The resulting master array contained one angry face, followed by one disgusted face, a face displaying fear, a happy face, a sad face, a surprised face and a neutral face over and over. Because we ordered each label array by identity prior to putting it all back together, we are guaranteed that identities are as close as they can possibly be while ordering by labels. The goal was simply to order the array of faces by labels while also keeping faces by the same identity close to one another.

Master Array Structure Example (Label , identity, Image(1024)):

*[[ 1, 1001, ...],[ 2, 10006, ...],[ 3, 1007, ...],[ 4, 1008, ...],[ 5, 1009, ...]*
*,[ 6, 10010, ...],[ 7, 1002, ...],[ 1, 1003, ...],[ 2, 1004, ...],[ 3, 10011, ...]*
*,[ 4, 10012, ...],[ 5, 1008, ...],[6, 10012, ...],[7, 10012, ...] ... ]*

To further guarantee that any results we obtained were meaningful during the validation stage, we wrote a K-Fold function which takes our newly ordered labeled data and return an array of tuples where the tuple contains training data, training labels, validation data and validation labels for the out of K folds. We then average out the classification rate over all folds every time we cross-validated a model. Due to processing time restrictions, we used 13 folds in all cases.

## Pre-Processing

Prior to classifying our data, we ran a series of pre-processing techniques in order to increase the classification success rate of our models. These preprocessing steps applied a variety of changes onto the initial data, changing it to define important features, in turn increasing our classification rates.

**Feature Scaling/Normalization**
The pre-processing technique that we applied was feature scaling, otherwise known as normalization. We used normalization so that the data features have an equality opportunity to influence the weights, ultimately resulting in better accuracy.

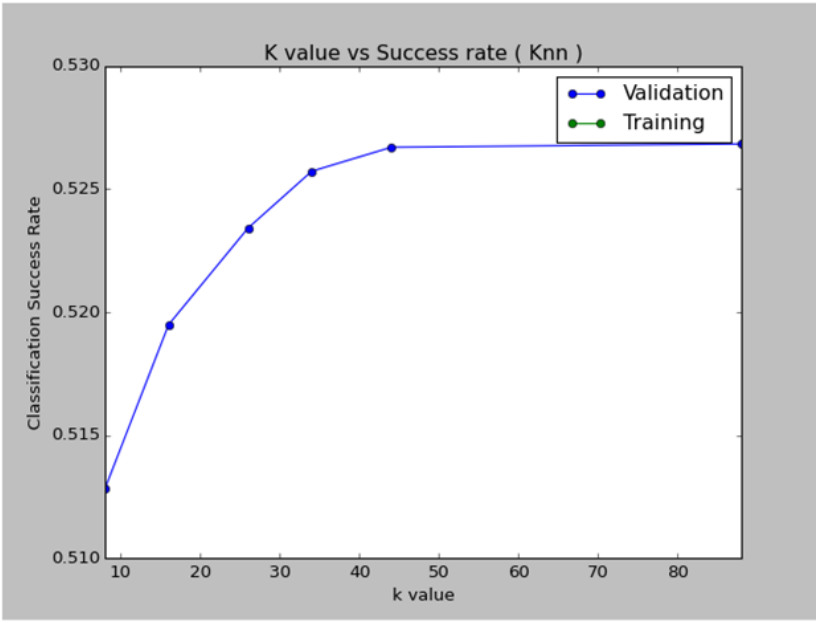
**Gamma correction**
After feature scaling our data, we apply gamma correction on the images. The human eye perceives light differently than the way that cameras do. Light is not perceived in a linear fashion. Rather, our eyes follow the gamma function which increases the sensitivity of darker tones more than bright ones. This pre-processing technique enhances the darker features of the image, allowing for more accurate classification due to areas such as the upper lip and eyebrows having less shade.

**Contrast Equalization**
The final step in our pre-processing equalizes the variance of the contrast of each image. This causes the intensities of the images to be distributed more extremely, causing the low contrast portions of the image to gain a significant amount of contrast. In practice, this method often leads to better definition in cheek bone areas and underexposed areas that require more detail.

# Bootstrap Aggregating
Before submitting any test results, we decided to use a bootstrap aggregator to increase our testing classification rate. Due to slow processing time, we observed that it wouldn't be practical to cross-validate with bootstrap aggregating so we only used it during the testing phase. We noticed a significant increase in testing classification rate when using Bootstrap Aggregating in all cases. As for hyperparameters, we noticed an increase in classification rate when the number of estimators was increased up until 15, in which case the classification rate reached a plateau.

# Model 1 : K-Nearest-Neighbours
In this project, a python base model was not given out. For this reason we used the Scikit-learn K-nn as our base model. The three hyperparameters which have the most effect on the classification rate are the number of neighbors, the weights and the algorithm type. We initialized our algorithm with an array of K values: K = [8, 16, 26, 34, 44, 88]. Furthermore, we set the weight function to predict classes by 'distance' in which closer neighbors have a greater influence on the classification than neighbors which are further away because we wanted faces with similar facial structure to affect our learning more strongly than those whose structure differs more. Lastly, we chose the algorithm setting to 'auto' which finds the most appropriate algorithm (Ball Tree or K-dimensional tree) to fit the data.

**Hyperparameter optimisation ( K ):**




The K-nn classifier was run with 6 different K values:

K = [8, 16, 26, 34, 44, 88]

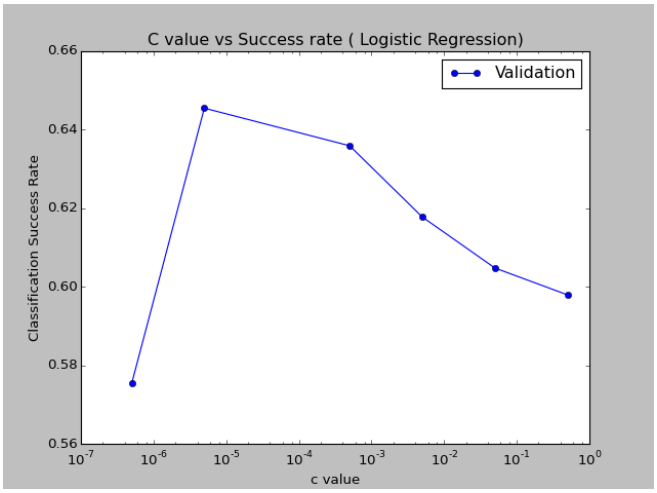
Over all runs, the training set classification rate was 100% for each fold no matter the value of K. Due to there being 7 different available classes for classification,we chose to test on values of K that were not multiples of 7. If we had not done so, a tie could have been possible in certain odd circumstances. The lowest value of K: 8 yielded a validation classification rate of approximately 0.513. As you can see in the above graph, as we increase the number of neighbors (K ), the algorithm classifies with better accuracy. Thus, we instinctively increased the value of K until the classification rate reached a plateau. From K: 44 and onward, we noticed that the validation classification rate began to plateau. Thus, despite K: 88 yielding a slightly higher validation classification rate,we chose a final K value of 44, as we prefered looking at a smaller amount of neighbours so as not to overfit the label spread of the labeled data set.

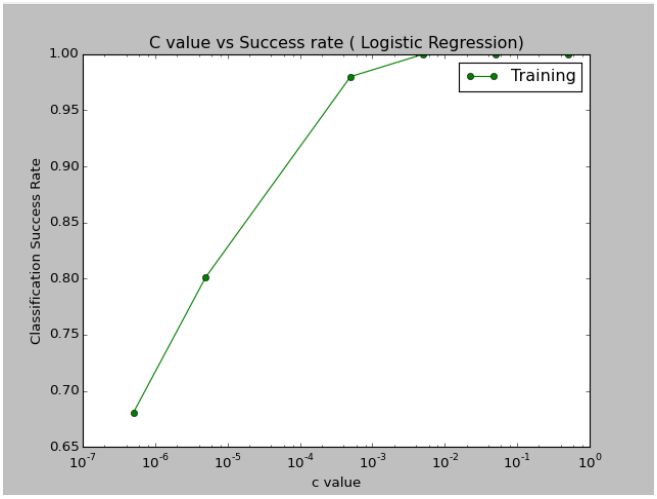
**Final Validation results without Pre-Processing:** ~ 0.354
**Final Validation results with Pre-Processing:** ~ 0.526
**Final Model Public Faces results:** ~0.552

# Model 2 : Logistic Regression

For our second model we used Logistic Regression taken from Scikit-learn's linear_model module. Due to this being a multi-class problem, the solver was limited to being either 'lbfgs' or 'sag'; we used the default solver 'lbfgs'. The lbfgs solver is limited to using only l2 penalties so we chose l2 penalties and optimized over an array of possible C values where C represents the "inverse of regularization strength".

**Hyperparameter optimisation ( C ):**




The Logistic Regression classifier was run with 6 different C values:

C = [0.0000005, 0.000005, 0.0005, 0.005, 0.05, 0.5]

Smaller values of C result in stronger regularization according to the Sklearn documentation so we stuck with low values in our optimization array. It seems that the lower the c value, the better the validation classification rate up until the value of 0.0000005. Just as well, it seems as though we are over-regularizing at the C value of 0.0000005 as the training data classification rate is extremely low for this value compared to the training classification rate for the other values. Generally, the training data classification rate was lower as we decreased the value of C, but at C=0.000005 we found a nice balance between avoiding over-fitting and avoiding over-regularization. Thus our final choice for the value of C is 0.000005.

**Final Validation Classification rate:** ~0.646
**Final Classification rate with PCA on public test faces:** ~0.699

# Model 3 : Kernel SVM

The final model we chose to try is a Kernel Support Vector Machine. The hyperparameters which we chose to modify from default were Gamma, C and the kernel used by the SVM. We used a very useful function in the scikit-learn library which lets us search for the most optimal parameters given an array of possibilities. We fed the function 5 values for both C and Gamma and it returned the most optimal values out of the ones given.

Gamma values fed into GridSearch:

[0.005, 0.05, 0.5, 5, 50]

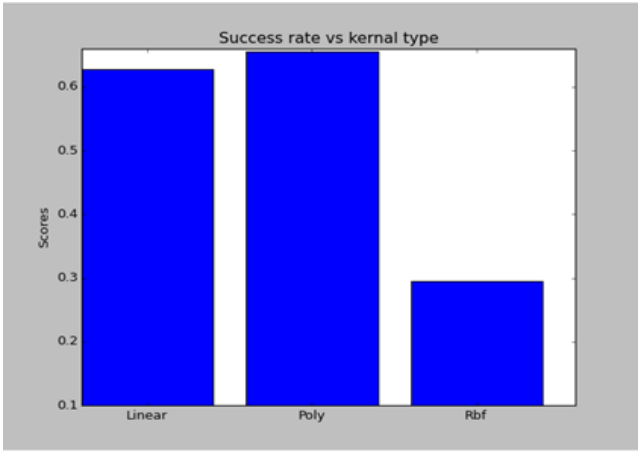
C values fed into GridSearch:

[0.001, 0.01, 0.1, 1, 10]

Optimal parameters output by Gridsearch:

Gamma: 0.5

C : 1




We then cross validated using our standard fold count (13) and optimized our kernel choice. We have the option of using a Linear kernel, a Polynomial kernel or the RBF kernel. Both the Linear and Polynomial kernels way outperformed the Rbf kernel. We would guess that this is due to the fact that we have a very large number of features where Rbf usually performs slightly worse. Based on these results, we chose a Polynomial kernel for our final version.

**PCA and SVM**

We tried using PCA to reduce the amount of features of the data input into the SVM. Using the unlabeled data given to us, we fit a PCA model and kept roughly 96.5% of the variance in the data. Using the model, we transformed the labeled data before splitting it into 13 folds and cross validated our SVM. Interestingly enough, though the model performed faster ( it had only 121 features compared to 1024 previously), it did not perform better. In fact, it classified roughly [NUMBER] less than when we did not use PCA. This is slightly intuitive to us because even though the data has a large amount of features, it makes sense that each feature would have some degree of importance because the entire face moves when difference expressions occur. In other words, it is not beneficial to reduce features if we know all features are important to some degree.

**Final Validation results without Pre-Processing:** ~ 0.630
**Final Validation results with Pre-Processing:** ~ 0.655
**SVM Test Faces classification rate without Bagging:** ~ 0.737
**SVM Test Faces classification rate with Bagging:** ~0.746 (Final Kaggle results)

## Comparison of models

The model which performed the worse was the KNN model. Even though the KNN model classified much better than if it was randomly guessing (0.14 classification rate), it only scored 52% in the 13 fold cross validation stage and not much higher on the public test faces. The model which gave us the best results was the Kernel SVM model which had a classification rate of 65% during the cross validation stage and roughly 75% on the public test faces. This is also intuitive because it is capable of classifying data that is too complex to separate in a low dimensional space. Because many faces can fit on some level into multiple categories, drawing a hyperplane in low dimension space may be impossible. Though K-nn is a non-linear classifier, it classifies non-linearly separable data by probabilistic learning whereas support vector machines assume that a hyperplane can be drawn to linearly separate the data in n-dimensional space. It is also worth noting that the SVM classifier did not perform much slower than the other two models.

## Instructions for using our code

In order to run one of our models on the given data, simply run the function with the argument “Submit” as ‘False’ if you wish to use kfold cross validation on the labeled data or ‘True’ if you wish to generate a csv.