Assignment 3
Daniel Crawford
CS 6375 - Machine Learnings

PART I

Algorithm:

The collaborative filtering algorithm is very expensive, and thus it demands many optimizations to produce good computational performance. Once fully implemented, the algorithm does X things:

1. Finds the average ratings from user A: O(n)
2. Find all users B which voted on target movie I: O(n)
3. Find all users C which have common ratings with user A: O(n^2)
4. Compute Pearson Coefficient for each user C times their vote on movie I minus their average vote. O(|users C| * n)
5. Compute absolute sum of Pearson Coefficients found for each user C and find inverse O(n)
6. Now return computation results. O(1)

That leaves our algorithm at O(n^2) in the worst case. With a data set of size 3 million, n^2 will take far too long. By observation, we can see that we can best improve our algorithm by looking at step 3 and 4. For step 3, we can improve our times from O(n^2) to O(|Target user Movies| * |Users that voted on Movie I|) by using a memory tradeoff. To do this, we generate a set of MovieIDs that each UserID has voted on, then index this using a dictionary. Set intersection is O(n) time, so we can find common movies between each other user faster than O(n^2) on average. This small optimization still leaves our algorithm at O(n^2) worst case, but on average it will produce more efficient results.

Evaluation:

The results of this algorithm show that it has gained a decent understanding of what a user's preferences are. Here is a quick list of the results from our test set.

Test sample size	100478
Train sample size	3255352
Root Mean Square Error	0.9494
Mean Absolute Error	0.7492

First of all, one thing to consider here is that our results are on a scale of five (i.e. one out of 5 stars). These results show that our error is usually about a star off from the actual rating. Our mean absolute error shows that our algorithm was usually 0.7492 stars off from a user's true preference. To be general, this performance allows us to to decide whether a user will like a movie or not. Where it struggles is determining at what extent will a user like a movie, so it may underestimate or overestimate how much a user actually enjoys a movie. However, there are issues in this algorithm that can be causing our errors to be higher. One assumption that is made here is that a user's average vote will be applied to a preference if it can't find any other users that our target user correlates with. An example of when this can happen if a user has only one vote on an obscure movie. One method to avoid these errors is to use default voting. With default voting, we can assume a default vote for other users, and compute a more likely rating for a target user instead of canceling values out to 0. This will increase complexity on the algorithm but improve the performance on the test set. Root mean square error(RMSE) shows us similar to mean absolute error what our differences are. Our RMSE also gives how skewed our residuals from our algorithm. Our RMSE confirms that our algorithm is at least able to accomplish whether a user will like or dislike a movie, because the points are on average 0.9494 stars off from the actual rating.

PART II

SVC using GridSearchCV:
For this table, 3-fold cross validation was used and the error rate is the average of all the errors trained on each fold. For each SVC classifier gamma was set to scale, max_iter was set to 500 and shrinking was so to True.

Kernel	Penalty(C)	Error rate
linear	1	0.18773
rbf	1	0.02374
poly	1	0.02747
sigmoid	1	0.25661
linear	5	0.18772
rbf	5	0.01857
poly	5	0.02385
sigmoid	5	0.2572
linear	10	0.18772
rbf	10	**0.01849**
poly	10	0.02442
sigmoid	10	0.25812

MLPCLassifier using GridSearchCV:
These estimates were made using 3-fold cross validation on GridSearchCV. In our MLPClassifier, alpha=1 for all classifiers.

activation	hidden_layer_sizes	solver	Error rate
identity	50	lbfgs	0.11001
identity	50	sgd	0.90138
identity	50	adam	0.09322
identity	100	lbfgs	0.1159
identity	100	sgd	0.9013
identity	100	adam	0.0917
identity	150	lbfgs	0.11699
identity	150	sgd	0.90138
identity	150	adam	0.09792
logistic	50	lbfgs	0.07535
logistic	50	sgd	0.05545
logistic	50	adam	0.06857
logistic	100	lbfgs	0.05335
logistic	100	sgd	0.04704
logistic	100	adam	0.06278
logistic	150	lbfgs	0.04790
logistic	150	sgd	**0.04357**
logistic	150	adam	0.05906

GradientBoostingClassifier:
These were all fitted individually without cross validation, and they all tested on the last 10000 elements of the dataset. For all classifiers loss=deviance and max_features=sqrt.

criterion	max_depth	learning_rate	Error rate
mse	5	0.1	0.011349
mse	4	0.1	0.02915
mse	3	0.1	0.05078
mse	2	0.1	0.08538
mse	1	0.1	0.1471
friedman_mse	5	0.1	**0.01043**
friedman_mse	4	0.1	0.02753
friedman_mse	3	0.1	0.05091
friedman_mse	2	0.1	0.08520
friedman_mse	5	0.01	0.07448

Out of all the classifiers, here are the top performers from each dataset:
SVC

Kernel=rbf	Penalty(C)=10	**0.01849**

MLPClassifier

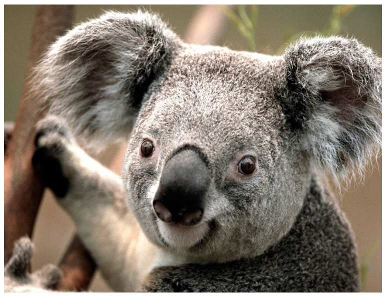
activation=logistic	hidden_layer_sizes=15 0	solver=sgd	**0.04357**

GradientBoostingClassifier

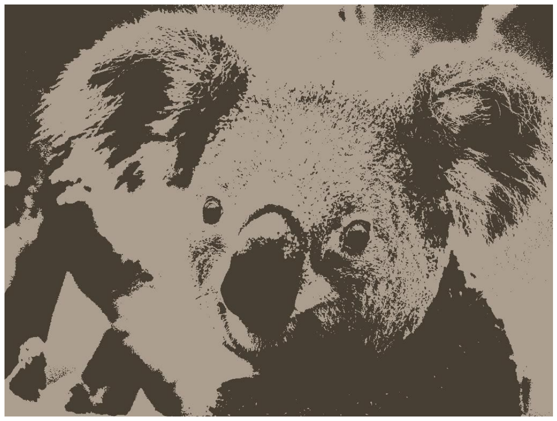
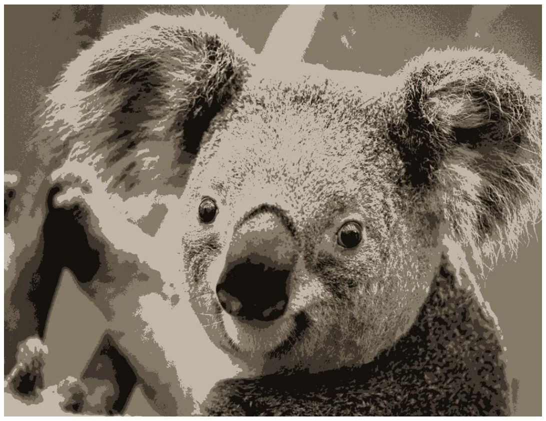
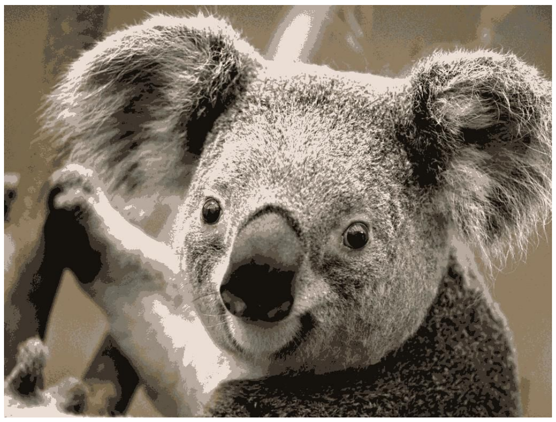
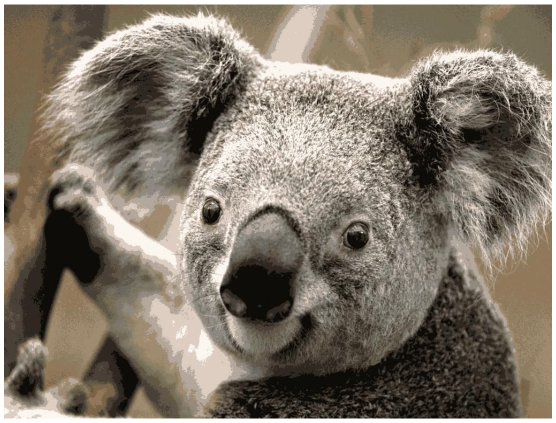
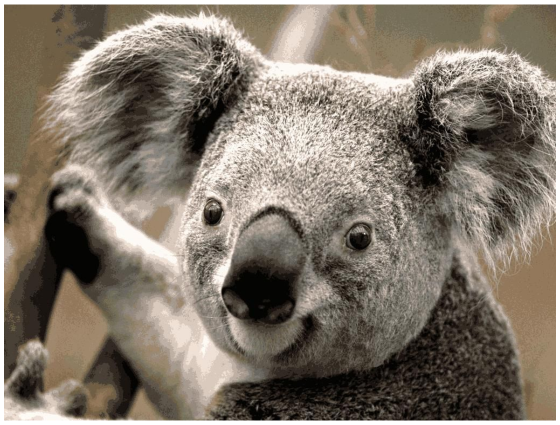
criterion=friedman_mse	max_depth=5	learning_rate=0.1	**0.01043**

From quick observation, we can see that GradientBoostingClassifier has done the best job at consistently classifying the number drawn, with only 1% of error. Upon closer inspection of GradientBoostingClassifier, the more specific the nodes are, the more accurate it is at classifying the test set. Also as expected, friemdan_mse produces slightly better results than mse.

KMEANS

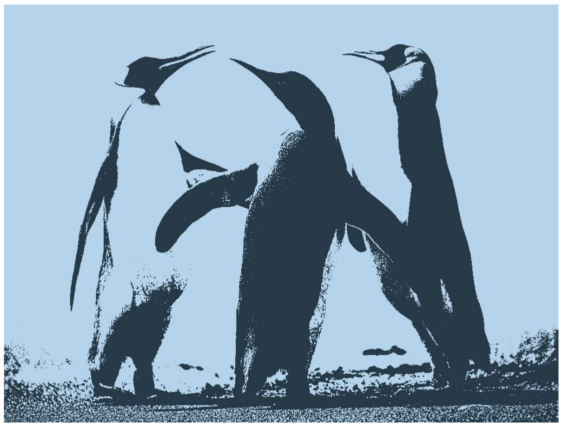
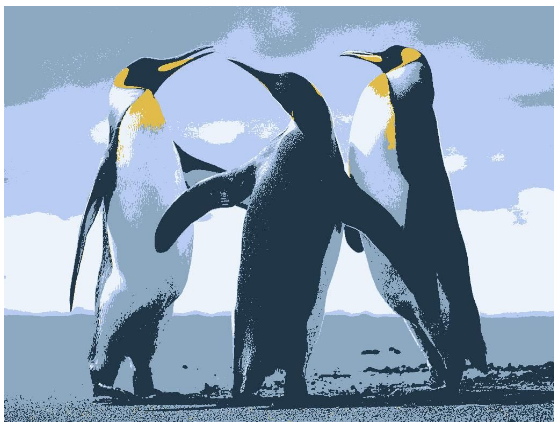
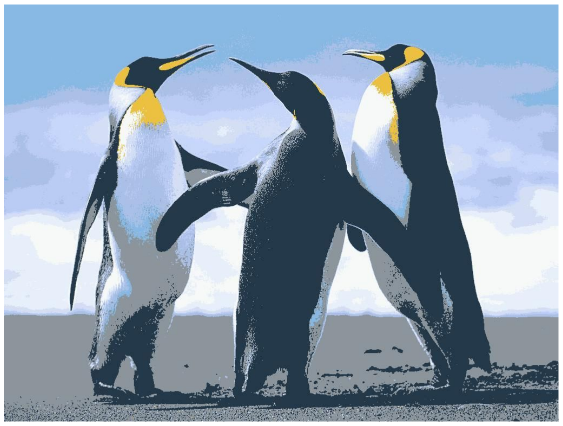
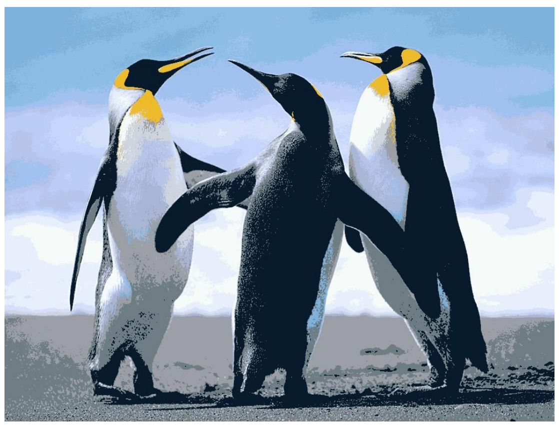
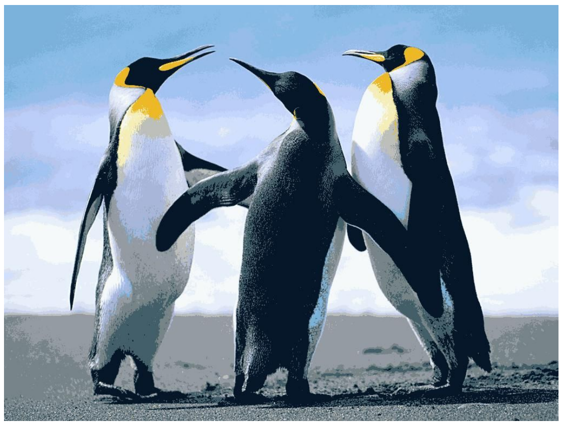
Koala Image: 

KMeans with N=2, 5, 10, 15, 20:

Penguins Image:

KMeans with N = 2,5,10,15,20:

Compression Ratios

NOTE: For this experiment, the seed on the random generator was taken off. The images generated above were generated using random seed ‘1234’. Compression was completed using the zip function in bash. Since these initializations are random, they can not be replicated but the estimates will still be close. The size of each experiment (cell) is 100.

K	Koala.jpg Compression Ratio Average	Koala.jpg Compression Ratio Variance	Penguins.jpg Compression Ratio Average	Penguins.jpg Compression Ratio Variance
2	1.041588	7.844*10^-6	1.1060266	2.779*10^-5
5	1.006996	2.555*10^-6	1.0478272	5.327*10^-5
10	1.001922	5.099*10^-7	1.0267085	2.425*10^-5
15	1.000842	1.743*10^-7	1.0169186	1.899*10^-5
20	1.000427	9.996*10^-8	1.0116690	1.159*10^-5

First of all, our results show us that our estimate of the compression ratio is fairly consistent, since the variance is fairly low. We can also see there there is a tradeoff between compression ratio and image quality: as image quality becomes more saturated, compression ratio converges to 1.
The best K for each image depends on their situation. If we look at Koala.jpg, ratio average drops heavily after the K increases to 5, meaning there is not much tradeoff. Since there is a small compression ratio, we will use K=20 for Koala.jpg since it has the highest quality and K=2 is not high quality enough. For Penguins.jpg, K=5 produces an image pretty close to the original, and it has a good compression ratio. Therefore, for Penguins.jpg we will use K=5.