Color Image Compression Using Unsupervised Learning (Clustering)

Pattern Recognition/ECE 471

William Willie Wells
wwells4@utk.edu

Abstract

Many supervised learning algorithms require massive amounts of training samples. Collecting and classifying the necessary training samples can be costly in time and money spent doing so. In such cases as data mining the classes may not be initially known and thus may depend on the groupings correlated. Features of individual classes may in some instances change overtime. These are some reasons why unsupervised learning is utilized. Clustering data points is one of the more natural, intuitive, grouping techniques especially when not dealing with distributions of continuous signals. Data points nearest each other in an n-dimension feature space are grouped into a cluster with a cluster center. Two algorithms used to implement the clustering procedure are k-means and Kohonen Maps (Self-Organizing Maps). A generalized version of Kohonen Maps is the Winner-Take-All algorithm. These algorithms can be used to lower the number of bits necessary to represent a single pixel in an image. Starting with an image that used $2^8*2^8*2^8$ possible color combinations and 8+8+8=24 bits to represent the image, k-means and Winner-Take-All algorithms were implemented reducing the (possible color combinations, bit count) per pixel to (512, 9), (256, 8), (128, 7), (64, 6), (32, 5) and (16, 4). Common gain/loss performance metrics applied to images are the RMSE and difference images. A difference image was produced for each k (possible color combinations). Using k = 16 for the k-means algorithm produces a representative image whereas the Winner-Take-All algorithm produces unrepresentative images at a k value of 128. The Kohonen Maps implementation of competitive learning would produce a representative image at the same k values as k-means.

Introduction:

When the number of classes corresponding to a data set, the class that represents each data point, and the importance of the individual features of the data point are unknown, unsupervised learning methods are utilized. Unsupervised learning attempts to recognize a hidden pattern in unclassified data. One of several ways to divide data points into separate classes is clustering. Clustering essentially makes use of the Minkowski distance of norm 2 known as Euclidean distance, clusters resembling circles, spheres, and hyper spheres in an n feature space for n=2,3,>3. Algorithms that utilize clustering include k-means, Winner-Take-All, and Kohonen Maps. The k-means algorithm involves picking an arbitrary k number of cluster centers (traditionally from the sample set), assigning each data point to one of the clusters using the nearest cluster mean, recalculating the cluster mean that now has m+1 members, and then repeating steps 2 and 3 for all points until no point changes its cluster membership. The Winner-Take-All algorithm selects an arbitrary ω number of cluster centers, assigns each data point to the nearest cluster center, updates the cluster center by adding to itself the product of the learning parameter (typically around 0.01) and the difference between the point and the cluster center, and repeats steps 2 and 3 for a programmer or user defined number of iterations. (1)(2)(3)(4)

Though their limits keep expanding, hardware devices have manufacturing limits such as in the case of a visual display unit the amount of bits allocated by the respective manufacturer to display the color spectrum range. A consequence is that indexed color images must be used to represent true color images. Algorithms such as k-means and Kohonen Maps are used to compress the number of bits used to represent each pixel thus reducing storage space. This process of reducing the number of simultaneously used colors is called color quantization. (5)(6)

In the absence of a training set supervised learning performance metrics such as the accuracy or precision of a classification decision rule are not applicable. The root mean square error for images gives a result in the range [0, 255] and represents the mean error between pixels of the original image and the compressed image. A difference image uses the absolute value of the difference between pixels of the original image and the compressed image pixel by pixel and is an image in itself. (2)(7)

Technical Approach:

The variables used to implement Euclidean distance are an unsupervised data vector (x_i), a cluster center vector (ω_i), number of elements of the vectors to be summed which is 3 for a <red, blue, green> image vector, and value of the norm which is 2 for Euclidean distance. Euclidean distance calculation is computed thus:

$L_2(\mathbf{x}, \boldsymbol{\omega}) = (\sum_{i=1}^{3} |x_i - \omega_i|^2)^{(1/2)}$

The difference, $x_i - \omega_i$, is calculated first and the difference is the multiplied by itself twice. The result of that calculation is then summed. The square root of this sum is taken using the pow(x,n) function of <cmath>.

Initializing cluster centers (n) is often done using points randomly selected from the data set, but this method is inappropriate for color quantization and would result in multiple classes with similar colors e.g. (<76,80,84>, <89,76,86>). Uniformly distributed points ($n_{u)}$ within the red, blue, green cube

([0, 255], [0, 255], [0,255])

were used to initialize cluster centers. Each individual interval was computed using:

iteration=ceil(256/(floor($\sqrt[3]{n}$)))

and a beginning offset was used to move initial cluster centers closer to the center of the RGB cube. The offset used was the lower of the two integers closest to half the iteration. If the uniform distribution $n_u < n$ corner points of the RGB cube (white, black, yellow, magenta, cyan, red, green, blue) were used to pad the initial clusters.

n_u=(floor($\sqrt[3]{n}$))^3

After initializing cluster centers according to the procedure above, every pixel is classified using minimum Euclidean distance for k-means. The mean (μ) of each cluster is computed. The cluster center is updated if the new cluster mean (μ) is different from the previous cluster center. All pixels ($\mathbf{x}$) are classified using minimum Euclidean distance. Cluster means are recomputed, cluster centers are update, and pixels are reclassified using minimum Euclidean distance until an epoch, cycle, when no cluster mean has changed. Throughout the mean update procedure the cluster centers are left as irrational numbers after they stop updating they are converted to the nearest integer. The original pixel data is replaced by the respective cluster center class data.

For Winner-Take-All cluster centers (ω) are initialized using uniformClustersRGB(ω) described above. The first pixel (x_0) is classified using minimum Euclidean distance. Initially every pixel's class is set to the first (0) class, thus on the first loop iteration (k) and whenever the classification for a pixel changes its new cluster center (ω_i) is updated using:

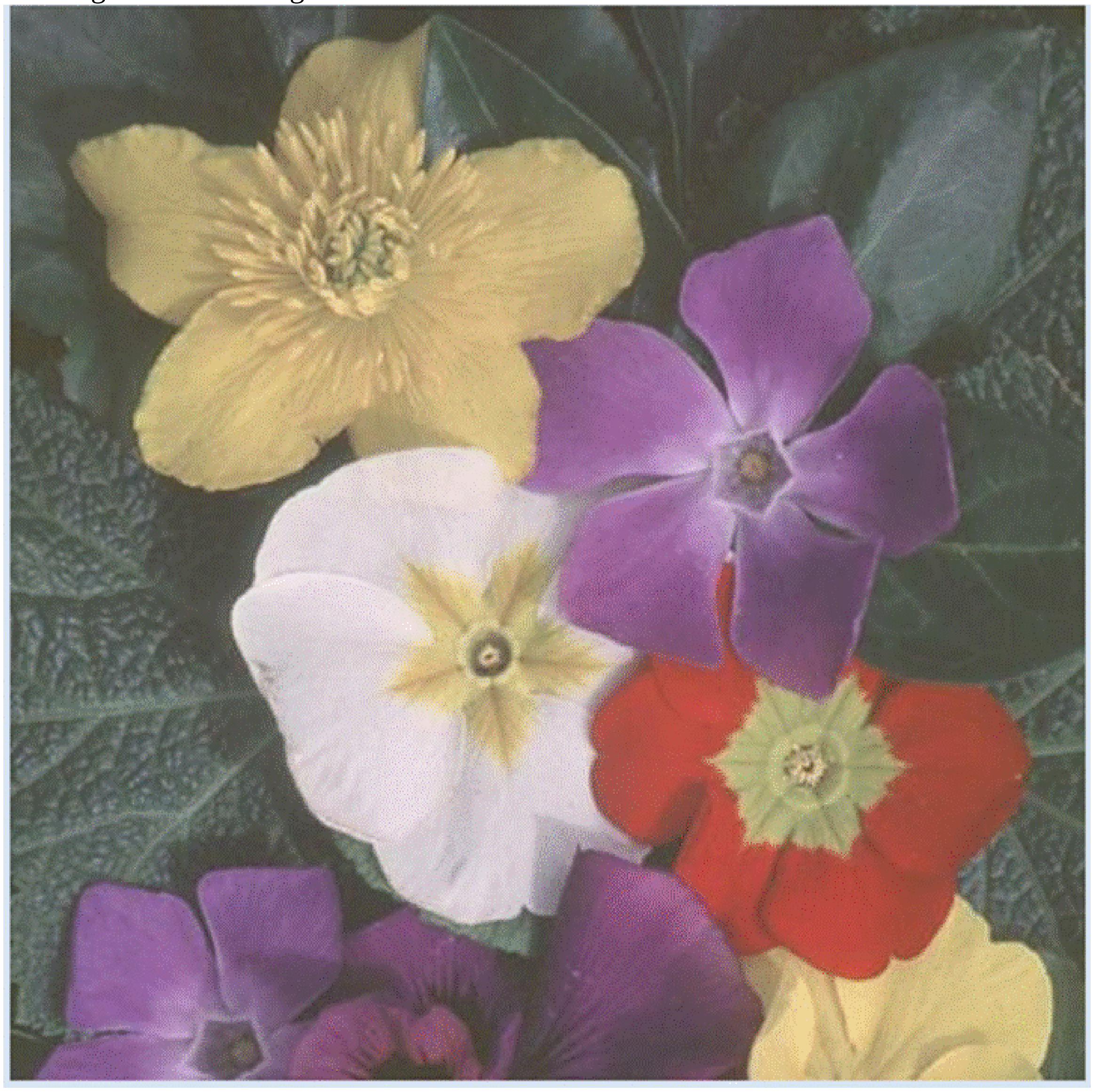
$\omega_i^{k+1} = \omega_i^{k} + \varepsilon * (x_j - \omega_i^{k})$

The learning parameter (ε) is a constant value of 0.01. A restriction of a hundred iterations was used. Throughout the cluster center update procedure the cluster centers are left as irrational numbers after the max iteration is reached they are converted to the nearest integer. The original pixel data is replaced by the respective cluster center class data.

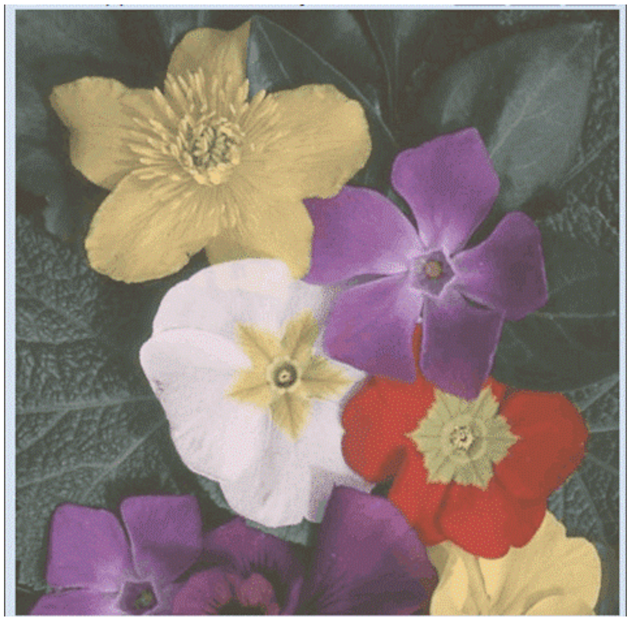
The final classified pixel matrices for k-means and Winner-Take-All are then subtracted from the original matrix, the absolute value of the resultant matrix is computed, and then written to a .ppm file.

Experiments and Results:

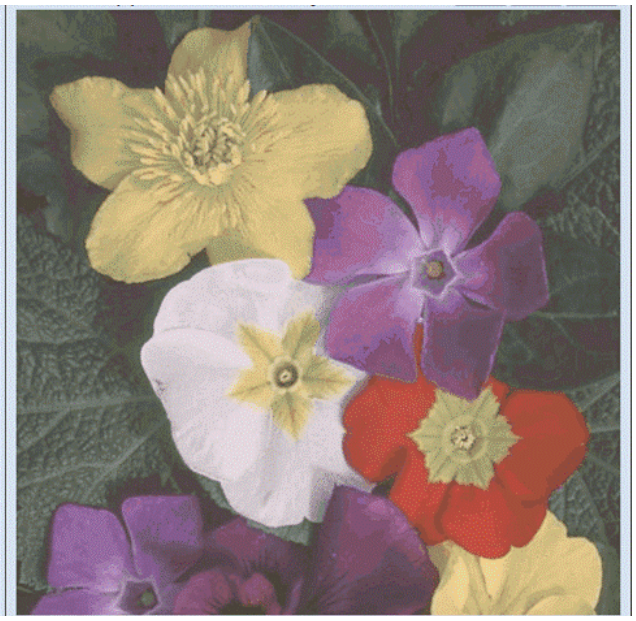
Values of n (number of possible color combinations) were 512, 256, 128, 64, 32, and 16. The original flower image that has n = 16777216:



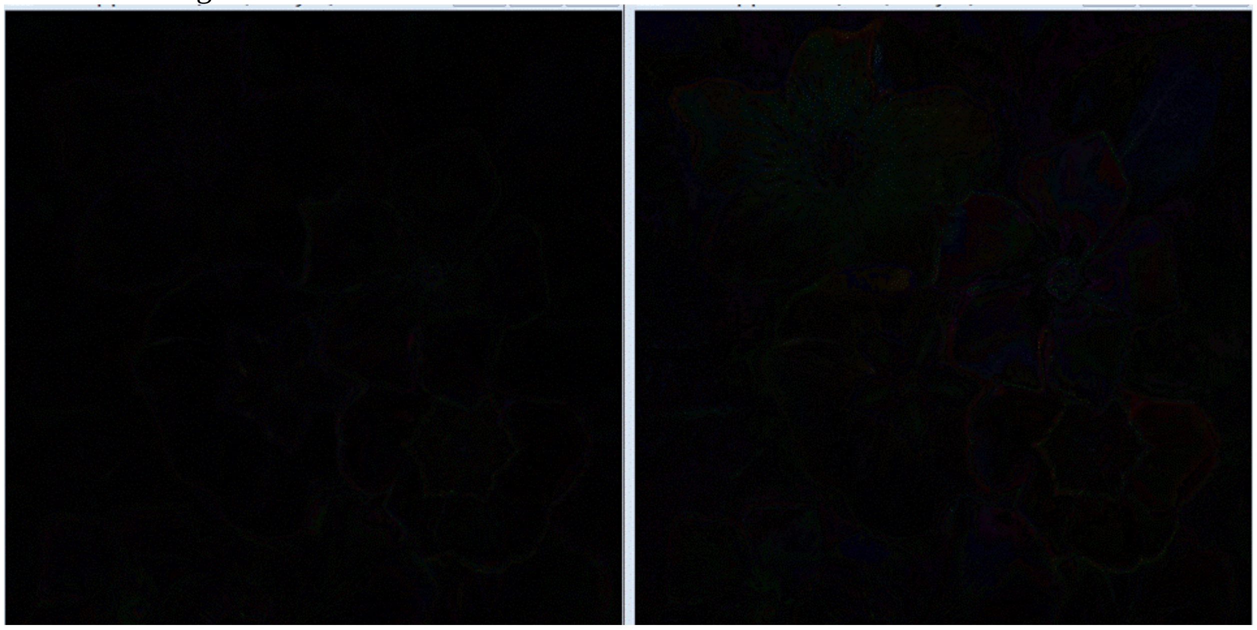
k-means with n = 512:



Winner-Take-All with n = 512:

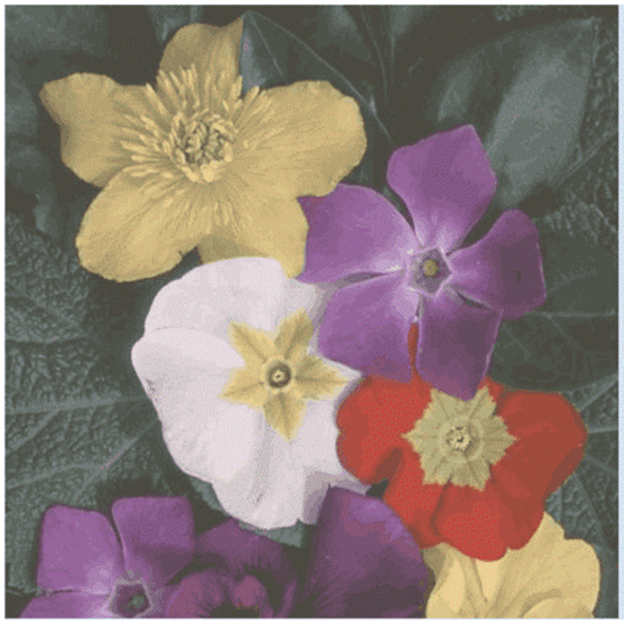


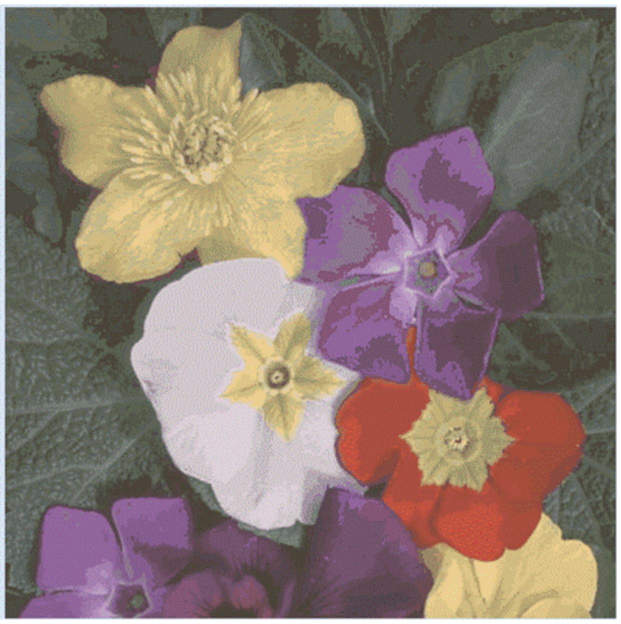
Difference images with n = 512:



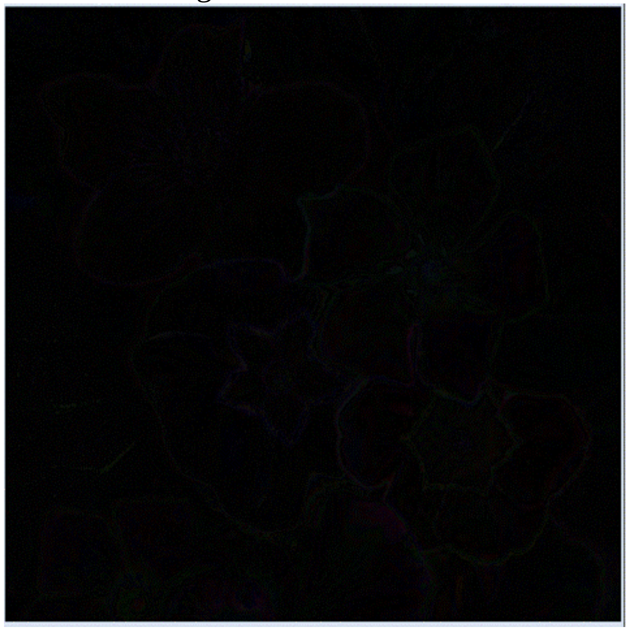
k-means with n = 256:

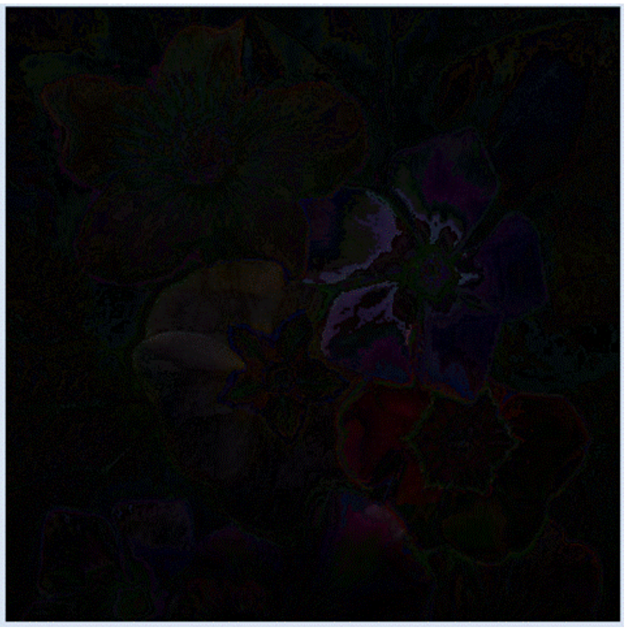
Winner-Take-All with n = 256:



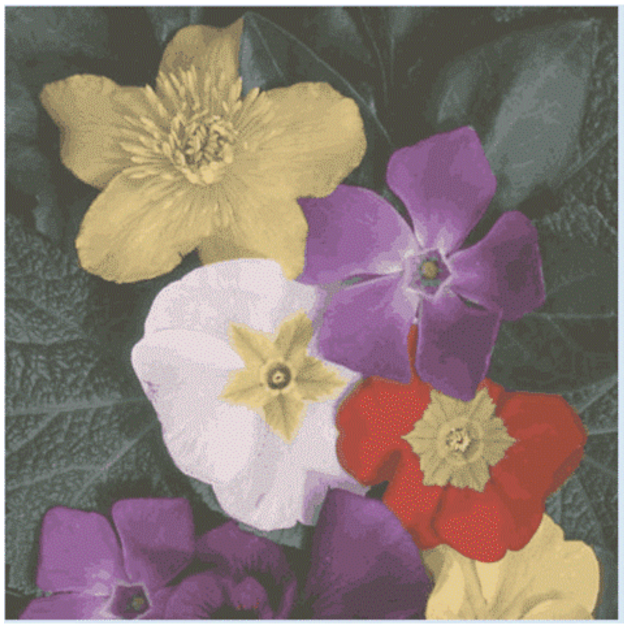


Difference images with n = 256:

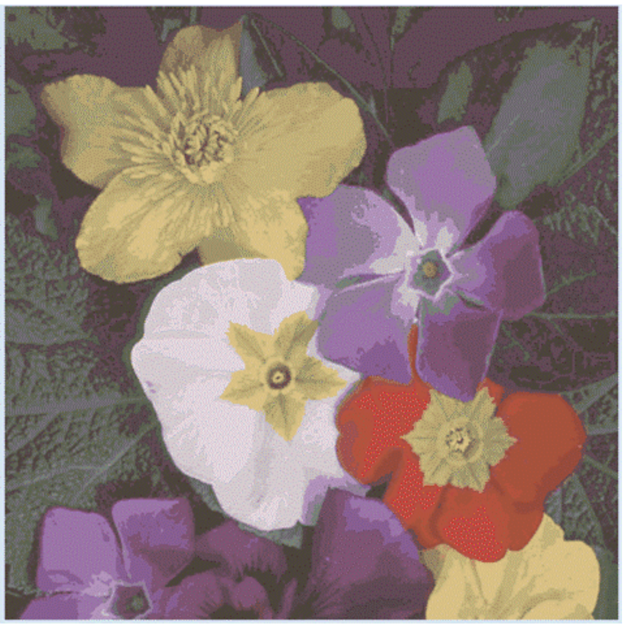




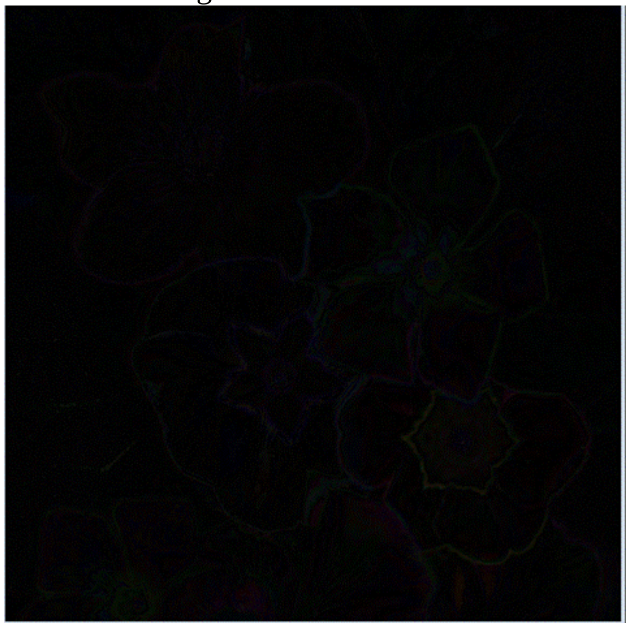
k-means with n = 128:



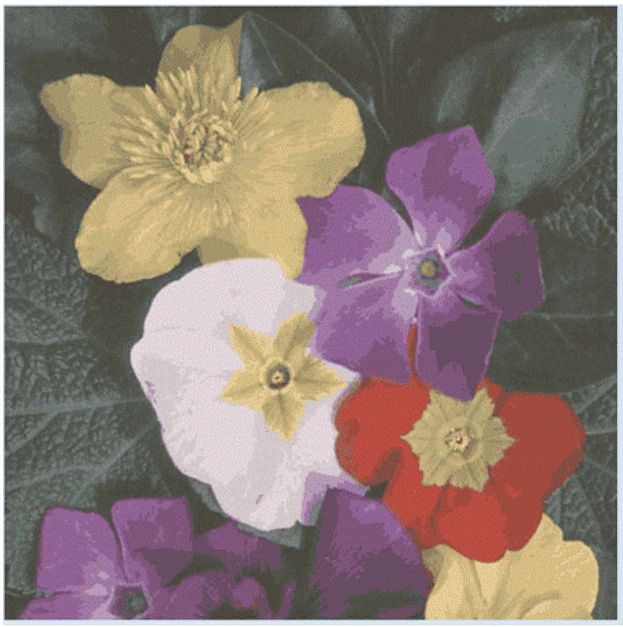
Winner-Take-All with n = 128:



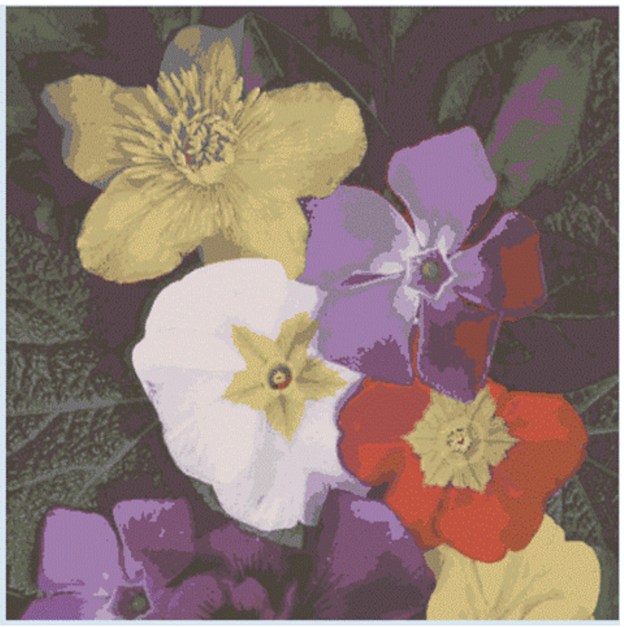
Difference images with n = 128:



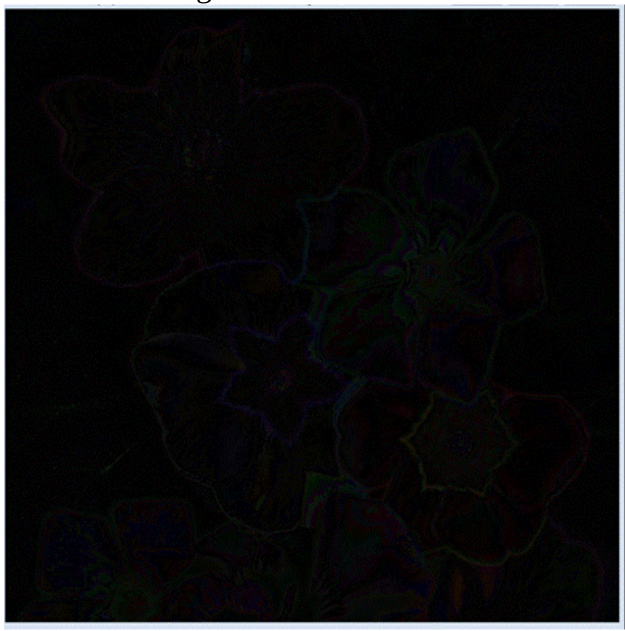
k-means with n = 64:



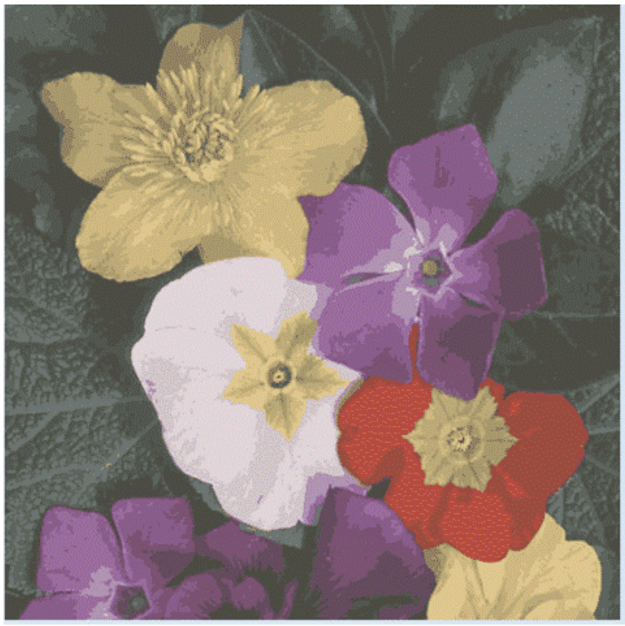
Winner-Take-All with n = 64:



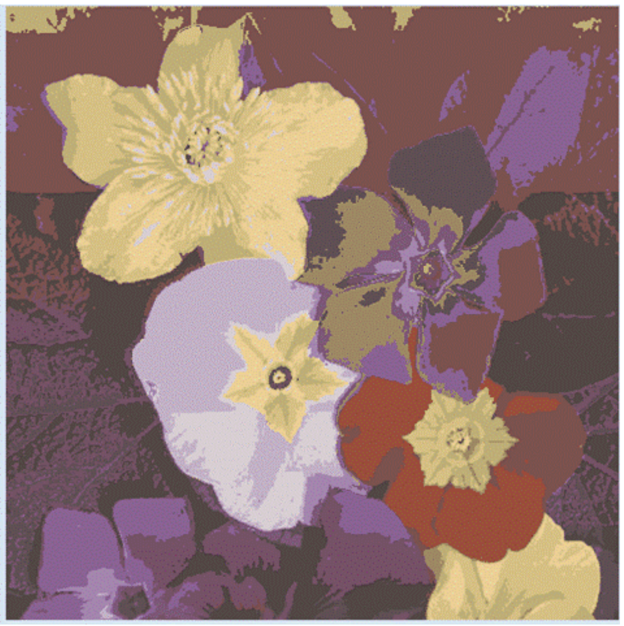
Difference images with n = 64:



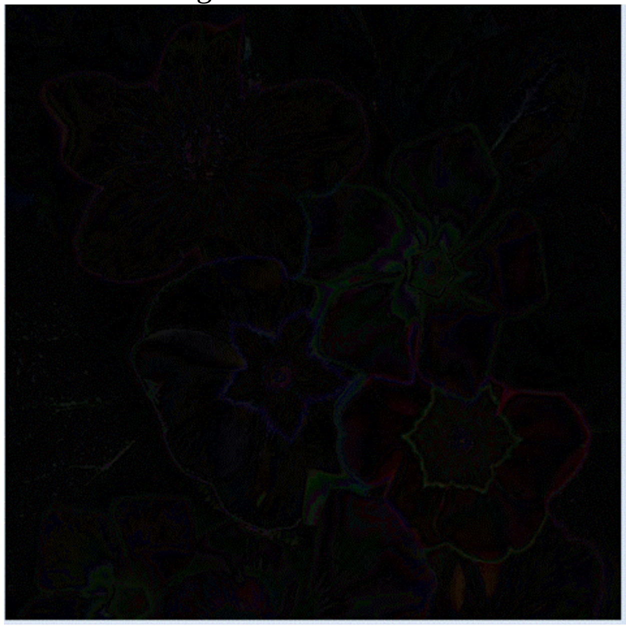
k-means with n = 32:



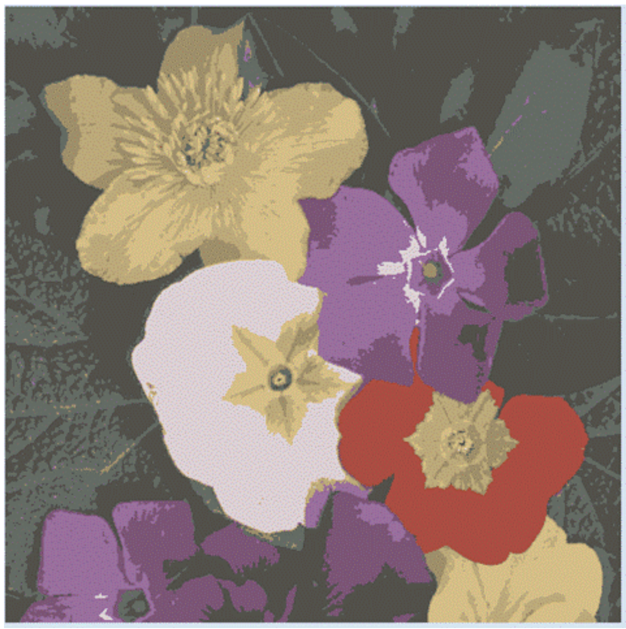
Winner-Take-All with n = 32:



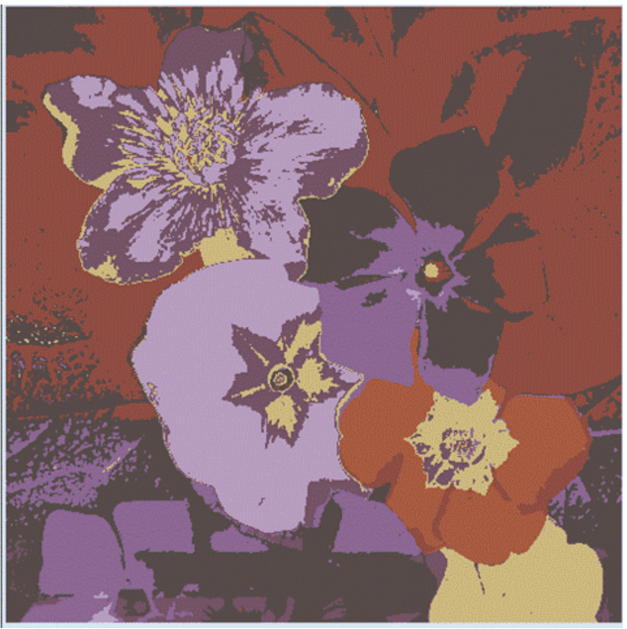
Difference images with n = 32:



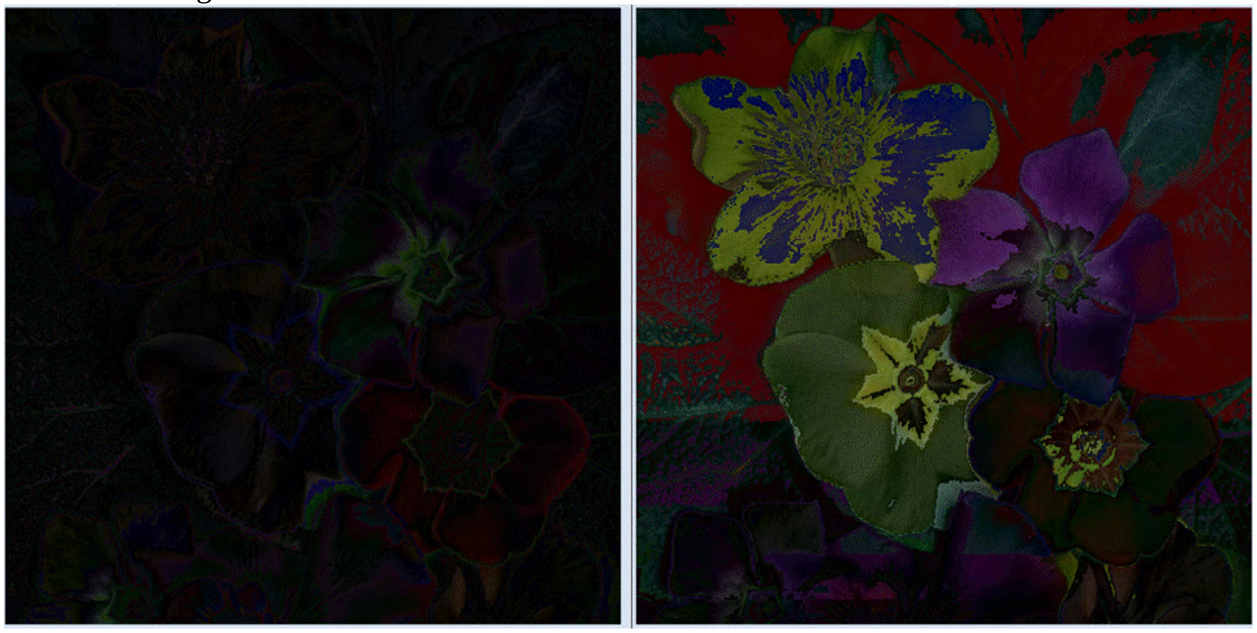
k-means with n = 16:



Winner-Take-All with n = 16:



Difference images with n = 16:



Conclusion:

The k-means images become more blurry as n is reduced. The Winner-Take-All images start producing clusters that do not represent the original data at n = 128. For all values of n k-means performs better than Winner-Take-All. The difference between their performance is magnified as n is reduced. Even at n = 16 the k-means image is representative of the original image, a person with impaired (near-sighted) vision versus a person with perfect sight. The

Winner-Take-All implementation of competitive learning requires more iterations and thus more computation time at lower values of n to return a representative image, increased iterations at all values of n would improve the respective images. Kohonen Maps with its shrinking neighborhood weighting system would return more accurate results for the same value of n as Winner-Take-All. Clustering is an effective instrument to implement color quantization if k-means or Kohonen Maps is used.

References:

1 R. O. Duda, P. E. Hart, D. G. Stork, *Pattern Classification*, 2nd Edition, John Wiley, 2001.
2 http://en.wikipedia.org/wiki/Unsupervised_learning
3 http://web.eecs.utk.edu/~qi/ece471-571/lecture19_clustering.pdf
4 http://en.wikipedia.org/wiki/Self-organising_map
5 http://web.eecs.utk.edu/~qi/ece471-571/project/proj4.htm
6 http://en.wikipedia.org/wiki/Color_quantization
7 http://people.duke.edu/~rnau/compare.htm

Appendix:

```
/*****************************************************************
 * proj4.cpp - implement color quantization using clustering
 *
 * Purpose: Color image compression using unsupervised learning
 *
 * Author: William Willie Wells
 *
 * Created: October 2013
 *
 *
 *****************************************************************/

#include <fstream>
#include <cstdlib>
#include <cmath>
#include "Pr.h"
#include "Matrix.h"
#include "clusters.h"

using namespace std;

// Color quantization with unsupervised learning using clusters
int main(int argc, char **argv){

                                        /*Preprocess data*/
            // error check input format
            if(argc<5){
        cout << "image file.ppm k-means output file.ppm ";
                              cout << "Winner-Take-All output file.ppm ";
                              cout << "k-means difference file.ppm ";
                              cout << "Winner-Take-All difference file.ppm endl"<<endl;
                         exit(1);
   }

            // read in data
            int nr, nc, n=256;
   Matrix X = readImage(argv[1], &nr, &nc);
            cout<<"input number of desired clusters:\n";
            cin>>n;
            if(n<1 | n >X.getRow()){n=256;}

                         /*k-means clustering*/
            // implement k-means algorithm
        Matrix kX=wk_means(X,n,255);
        Matrix kY=X-kX;
        absMat(kY);

                /*Winner-take-all clustering*/
        // implement winner-take-all algorithm
        Matrix wX=wWinnerTakeAll(X,n,0.01,255);
        Matrix wY=X-wX;
        absMat(wY);

                /*Write output to file*/
        writeImage(argv[2],kX,nr,nc);
        writeImage(argv[3],wX,nr,nc);
        writeImage(argv[4],kY,nr,nc);
        writeImage(argv[5],wY,nr,nc);

        return 0;
}

/***************************************************************
 * clusters.cpp - Unsupervised Learning using clusters
 *
 * Purpose: implement color quantization with unsupervised
 *          learning using clusters.
 *
 *       - absMat: absolute value of Matrix entries
 *       - uniformClustersRGB: initial clusters equally spaced in
 *                          a RGB cube
 *       - minDistPt: pixel classification using minimum
 *                    Euclidean distance
 *       - minDistUns: unsupervised classification of a vector of
 *                     pixels using minimum Euclidean distance
 *       - m_dtoi: nearest integer conversion of Matrix entries
 *       - indexColor: conversion of original pixel to cluster
 *                     center pixel value
 *       - wk_means: classification of pixels using k cluster means
 *       - wWinnerTakeAll: classification of pixels using
 *                         competitive learning
 *
 * Author: William Willie Wells
 *
 * Created: October 2013
 *
 *
 ***************************************************************/

#include <cstdlib>
#include <cmath>
#include "nPDE.h"

// Calculate the absolute value of all entries of a Matrix
void absMat(Matrix &x){
        for(int i=0;i<x.getRow();i++){
                for(int j=0;j<x.getCol();j++){
                        x(i,j)=abs(x(i,j));
                }
        }
}

// Create uniformly distributed initial cluster centers in an RGB cube space
Matrix uniformClustersRGB(int n, int dMax){
        Matrix wc(n,4);
        int r,g,b,count=0;

        // calculate iteration step size and initial offset
        double m = pow(n,(1/((double)3)));
        double mi=w_dtoi(m);
        if(mi*mi*mi>n){m=floor(m);}
        else{m=mi;}
        int iter = (int)ceil((dMax+1)/m);
        int init = (int)floor(iter/2);

        // create a vector of uniformly distributed points
        // in 3-space with an identifier
        for(int i=init;i<dMax;i+=iter){
                for(int j=init;j<dMax;j+=iter){
                        for(int k=init;k<dMax;k+=iter){
                                wc(count,0)=i;
                                wc(count,1)=j;
                                wc(count,2)=k;
                                wc(count,3)=count;
                                count++;
                        }
                }
        }

        // if number of uniform central clusters < n, pad with boundary colors
        if(count<n){
                Matrix extremeColor(8,3);

                // organize boundary colors with a designed weight of importance
                // white, black, yellow, magenta, cyan, red, green, blue
                for(int e=0;e<8;e++){
                        r=dMax;
                        g=dMax;
                        b=dMax;
                        if(e==1 | e==4 | e==6 | e==7){
                                r=0;
                        }
                        if(e==1 | e==3 | e==5 | e==7){
                                g=0;
                        }
                        if(e==1 | e==2 | e==5 | e==6){
                                b=0;
                        }
                        extremeColor(e,0)=r;
                        extremeColor(e,1)=g;
                        extremeColor(e,2)=b;
                }

                // pad with boundary colors
                for(int c=0;c<8;c++){
                        if(count==n){break;}
                        for(int rgb=0;rgb<3;rgb++){
                                wc(count,rgb)=extremeColor(c,rgb);
                        }
                        wc(count,3)=count;
                        count++;
                }
        }

        return subMatrix(wc,0,0,count-1,wc.getCol()-1);
}

// Classify a point given initial cluster centers
double minDistPt(Matrix &xc, Matrix &pt, float dMin){
        double cn=-1;

        // compute unsorted Euclidean distances from a point to a vector of points
        Matrix uDist=wMinkowski(xc,pt,2,pt.getCol());

        // classify according to minimum distance
        for(int i=0;i<uDist.getRow();i++){
                if(uDist(i,0)<dMin){
                        dMin=uDist(i,0);
                        cn=xc(i,pt.getCol());
                }
        }

        return cn;
```

```
}

// Classify a vector of points given initial cluster centers (Unsupervised),
// using minimum Euclidean distance
Matrix minDistUns(Matrix &xc, Matrix &y, float dMax){
        Matrix txp(1,y.getCol());
        Matrix yc(y.getRow(),y.getCol()+1);
        float dMin;

        copyMat(y,yc);
        for(int i=0;i<y.getRow();i++){
                for(int j=0;j<y.getCol();j++){
                        txp(0,j)=y(i,j);
                }

                dMin=y.getCol()*dMax;
                yc(i,y.getCol())=minDistPt(xc,txp,dMin);
        }

        return yc;
}

// Convert a Matrix of doubles to the nearest integer values
void m_dtoi(Matrix &x){

        for(int i=0;i<x.getRow();i++){
                for(int j=0;j<x.getCol();j++){
                        x(i,j)=w_dtoi(x(i,j));
                }
        }
}

// Convert to indexed color
void indexColor(Matrix &ic, Matrix &tc){
        for(int i=0;i<tc.getRow();i++){
                for(int j=0;j<tc.getCol()-1;j++){
                        tc(i,j)=ic(tc(i,tc.getCol()-1),j);
                }
        }
}

// Implement k-means algorithm
Matrix wk_means(Matrix &x, int n, float dMax){
        Matrix tc,mu;
        int flag=0;

        // assign n cluster centers and every point to nearest cluster
        Matrix wc=uniformClustersRGB(n,dMax);

        // initial classes - assign every point to nearest cluster
        Matrix xc=minDistUns(wc,x,dMax);

        // compute sample mean of each cluster: mu=mean(yi,yi.getCol()-1);
        while(flag<3*wc.getRow()){
                // compute mean of each cluster and update cluster centers
                for(int i=0;i<wc.getRow();i++){
                        if(i==0){flag=0;}
                        tc=splitData(xc,i);
                        mu=mean(tc,xc.getCol()-1);
                        for(int j=0;j<x.getCol();j++){
                                if(wc(i,j)==mu(j,0) | (tc.getRow()==1 & tc(0,j)!=wc(i,j))){
                                        flag++;
                                }else{
                                        wc(i,j)=mu(j,0);
                                }
                        }
                }

                // reassign each sample to cluster with the nearest mean
                xc=minDistUns(wc,x,dMax);
```

```
	}

	// double to nearest integer
	m_dtoi(wc);

	// convert members of a class to pixel values of the class for output
	indexColor(wc,xc);

	return subMatrix(xc,0,0,xc.getRow()-1,xc.getCol()-2);
}

// Implement winner-take-all
Matrix wWinnerTakeAll(Matrix &x, int n, float eps, float dMax){
	Matrix txp(1,x.getCol());
	int iter=0;
	int flag=1;
	double oldClass,newClass;

	// assign n cluster centers
	Matrix wc=uniformClustersRGB(n,dMax);

	// initiate classification matrix
	Matrix xc(x.getRow(),x.getCol()+1);
	copyMat(x,xc);

	// modify cluster center (w_a)
	// using (w_a_new)=(w_a_old)+epsilon*(x-w_a_old); epsilon=0.01
	while(flag>0 & iter<100){
		flag=0;
		for(int i=0;i<x.getRow();i++){
			// extract pixel vector
			for(int j=0;j<x.getCol();j++){
				txp(0,j)=x(i,j);
			}

			// assign pixel to winner
			oldClass=xc(i,x.getCol());//initially 0
			xc(i,x.getCol())=minDistPt(wc,txp,dMax);//winner
			newClass=xc(i,x.getCol());

			// update cluster center
			if(newClass!=oldClass | iter==0){
				for(int k=0;k<x.getCol();k++){
					wc(newClass,k)=wc(newClass,k)+eps*(xc(i,k)-wc(newClass,k));
				}
				flag=1;
			}
		}
		iter++;
	}

	// double to nearest integer
	m_dtoi(wc);

	// transform 2^24 possible true colors --> n indexed colors
	indexColor(wc,xc);

	return subMatrix(xc,0,0,xc.getRow()-1,xc.getCol()-2);
}

/*****************************************************************
 * nonParaDE.cpp - Non-Parametric Density Estimation
 *
 * Purpose: implement k-Nearest Neighbor and non-uniform size
 *		validation set Cross Validation.
 *
 *	- wMinkowski: calculate Minkowski distances of xi-p
 *	- wKNN: classify a point (p) using k-nearest neighbor
 *	- mkNN: classify a vector of points, evaluate accuracy
 *			and timing using kNN
 *          - divideSet: separate a data set into training and
 *                         validation sets
 *          - crossValidate: use cross validation to verify accuracy
 *
 * Author: William Willie Wells
 *
 * Created: October 2013
 *
 *
 ***************************************************************/

#include <cstdlib>
#include <cmath>
#include <ctime>
#include "PnE.h"

// Calculate Minkowski distances from each point
Matrix wMinkowski(Matrix &x, Matrix &pt, int d,int p){
	Matrix tmp,res;
	double t1;

	tmp.createMatrix(x.getRow(),1);
	res.createMatrix(x.getRow(),1);
	// (sum j=[1,p]: [abs(x(i,j)-pt(0,j))]^d)^1/d
	for(int i=0;i<x.getRow();i++){
		for(int j=0;j<p;j++){//partial feature
			t1=x(i,j)-pt(0,j);
			tmp(i,0)=1.0;
			for(int k=0;k<d;k++){//k distance metric
				tmp(i,0)*=abs(t1);
			}
			res(i,0)+=tmp(i,0);
		}
		// compute dth root
		res(i,0)=pow(res(i,0),(1/((double)d)));
	}

	return res;
}

// Use k-Nearest Neighbor (kNN) to clasify a point
double wKNN(Matrix &x,Matrix &pt,int k, int d, int p){
	Matrix uDist,sDist,iDist,ctc;
	int cnt=0;
	int cnt2=-1;
	int cnt3=-1;
	int cMax=-1;

	// calculate distances from each point
	uDist=wMinkowski(x,pt,d,p);
	uDist=transpose(uDist);

	// sort distances lowest-->highest
	sDist.createMatrix(1,uDist.getCol());
	iDist.createMatrix(1,uDist.getCol());
	insertsort(uDist,sDist,iDist);

		/*classification*/
	// determine number of classes
	ctc.createMatrix(x.getRow(),3);
	for(int i=0;i<x.getRow();i++){
		if(i==0){
			ctc(cnt,0)=x(i,x.getCol()-1);
			cnt++;
		}else{
			for(int j=0;j<cnt;j++){
				if(x(i,x.getCol()-1)==ctc(j,0)){
					break;
				}else if(j==cnt-1){
					ctc(cnt,0)=x(i,x.getCol()-1);
```

```
                                cnt++;
                            }
                        }
                    }
            }
            ctc=subMatrix(ctc,0,0,cnt-1,2);

            // count respective classes of first k nearest neighbors
            if(k>x.getRow()){
                    k=x.getRow();
            }
            for(int w=0;w<k;w++){
                    for(int v=0;v<ctc.getRow();v++){
                            if(x((int)iDist(0,w),x.getCol()-1)==ctc(v,0)){
                                    ctc(v,1)+=1.0;
                                    ctc(v,2)+=w;
                            }
                    }
            }

            // classify point
            for(int c=0;c<cnt;c++){
                    if(ctc(c,1)>cnt2 | (ctc(c,1)==cnt2 & ctc(c,2)<cnt3)){
                            cMax=ctc(c,0);
                            cnt2=ctc(c,1);
                            cnt3=ctc(c,2);
                    }
            }

            return cMax;
}

// Use k-Nearest Neighbor method to classify a vector of points
Matrix mkNN(Matrix &x, Matrix &y, int k, int d,int p){
        Matrix tyr,ty,oE;
        int n,o=0;
        clock_t t0;

        // initiate matrices
        tyr.createMatrix(1,y.getCol()-1);
        ty=formatResult(y,y);
        oE.createMatrix(((k+1)/2)*d,4);

        // run kNN for all points, all distances [0, d], and all odd values [1,k)
        for(int m=1;m<d+1;m++){
                n=1;
                while(n<k+1){
                        t0=clock();
                        for(int i=0;i<y.getRow();i++){
                                for(int j=0;j<y.getCol()-1;j++){
                                        tyr(0,j)=y(i,j);
                                }
                                ty(i,y.getCol())=wKNN(x,tyr,n,m,p);
                        }
                        // measure run time and accuracy for each (k,d)
                        oE(o,0)=n;
                        oE(o,1)=m;
                        oE(o,2)=execTime(t0);
                        oE(o,oE.getCol()-1)=wAccuracy(ty,ty.getCol()-1);
                        n+=2;
                        o++;
                }
        }

        return oE;
}

// Divide data set into a training and a validation(test) set
void divideSet(Matrix &x,Matrix &M,Matrix &tr,Matrix &te,int n){
        Matrix tTr,tTe;
```

```
	int c=0;
	int rCt=0;
	int eCt=0;
	int flag=0;

	// find number of items in test set
	for(int j=0;j<M.getRow();j++){
		if(M(j,n)>0){
			c++;
		}
	}
	tTe.createMatrix(c,x.getCol());
	tTr.createMatrix(x.getRow()-c,x.getCol());

	// separate training and testing sets
	for(int i=0;i<x.getRow();i++){
		for(int k=0;k<c;k++){
			if(i+1==M(k,n)){
				for(int p=0;p<x.getCol();p++){
					tTe(eCt,p)=x(i,p);
				}
				eCt++;
				flag=1;
			}
		}
		if(flag==0){
			for(int r=0;r<x.getCol();r++){
				tTr(rCt,r)=x(i,r);
			}
			rCt++;
		}else{ flag=0;}
	}

	// return training and test sets
	tr=tTr;
	te=tTe;
}

// Run cross Validation n times with n semi-random validation sets
Matrix crossValidate(Matrix &x, Matrix &M, int kMax, int dMax, int n){
	Matrix rpe,tr,te,cva;

	// implement cross validation n times
	for(int ni=0;ni<n;ni++){
		// separate training and test data sets
		divideSet(x,M,tr,te,ni);

		// use kNN to classify data set
		rpe=mkNN(tr,te,kMax,dMax,tr.getCol()-1);

		// return mean accuracy (1-MAE not MSE)
		if(ni==0){
			cva=rpe;
		}else{
			for(int i=0;i<rpe.getRow();i++){
				cva(i,3)+=rpe(i,3);
				cva(i,2)+=rpe(i,2);
				if(ni==n-1){
					cva(i,3)/=n;
				}
			}
		}
	}

	return cva;
}

/****************************************************************
 * prepEval.cpp - preprocessing and performance evaluation
 *
 * Purpose: preprocess data to be used with the Matrix library
 *                 and evaluate classification performance.
 *
 *          - w_dtoi: convert irrational number to nearest integer
 *          - execTime: measure execution time in seconds
 *          - strToBit: find a user defined string and assign it a
 *                           binary value and assign all else the other
 *                           binary value
 *          - xNorm: normalize a generic data set using mean and
 *                       standard deviation
 *          - splitData: split all members of a class predefined in
 *                            a training set, user defined input
 *          - copyColumn: copy a specified column in one Matrix to a
 *                             specified column in another Matrix
 *          - copyMat: copy a Matrix into another Matrix
 *          - formatResult: add original classification column to a
 *                                reduced dimension Matrix
 *          - removeSpace: remove rows of all zeros and transpose
 *          - wAccuracy: calculate accuracy of classification
 *          - ruleEval: calculate performance evaluation values
 *
 * Author: William Willie Wells
 *
 * Created: September 2013
 *
 * Modified: October 2013
 ***************************************************************/

#include <fstream>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include "Matrix.h"

// Convert irrational number to nearest integer
double w_dtoi(double d){
		double fp=floor(d);
		double cp=ceil(d);

		if(cp-d<d-fp){return cp;}
		else{return fp;}
}

// Measure execution time
float execTime(clock_t t0){
	return ((float)(clock()-t0))/CLOCKS_PER_SEC;
}

// Find indicated string in a file and assign bit values to x(:,n)
void strToBit(Matrix &x, int n, string xStr,char *xFile){
	string findS;

	// sift through xFile until the nth column, compare value to xStr,
	// and assign x(:,n) a binary value.
	ifstream fS(xFile,ios::in);
	for(int i=0;i<x.getRow();i++){
		for(int j=0;j<x.getCol();j++){
			fS >> findS;
			if(j==n){
				if(findS==xStr){
					x(i,n)=0;
				}else{
					x(i,n)=1;
				}
			}
		}
	}
}

// Normalize the data set so that features have the same scale
```

```
void xNorm(Matrix &y,Matrix &x){
        Matrix mu,S;

        // calculate the mean and covariance
        mu=mean(x,x.getCol()-1);
        S=cov(x,x.getCol()-1);

        // normalize the data set
        for(int i=0; i<y.getRow();i++){
                for(int j=0; j<y.getCol()-1;j++){
                        y(i,j)=(y(i,j)-mu(j,0))/(sqrt(S(j,j)));
                }
        }
}

// Find class identifier and copy row into Matrix
Matrix splitData(Matrix &x, int c){
        Matrix tmp;
        int count=0;

    // find class identifier and copy row
    tmp.createMatrix(x.getRow(),x.getCol());
    for (int i=0; i<x.getRow(); i++){
        if(x(i,x.getCol()-1) == c){
                        for(int j=0;j<x.getCol();j++){
                                tmp(count,j)=x(i,j);
                        }
                        count++;
                }
        }

        return subMatrix(tmp,0,0,count-1,tmp.getCol()-2);
}

// Copy a specific column in Matrix 1 to a specific column in Matrix 2
void copyColumn(Matrix &data, Matrix &rDat, int dc,int rc){

        for(int i=0;i<data.getRow();i++){
                rDat(i,rc)=data(i,dc);
        }
}

// Copy values of one Matrix into another without modifying size of target
void copyMat(Matrix &data, Matrix &rDat){

        for(int j=0;j<data.getCol();j++){
                copyColumn(data,rDat,j,j);
        }
}

// After dimensionality reduction reformat matrix for classification
Matrix formatResult(Matrix &xCopy1, Matrix &xCopy2){
        Matrix tmp;

        // append last column of Matrix 2 to Matrix 1
        tmp.createMatrix(xCopy1.getRow(),xCopy1.getCol()+1);
        copyMat(xCopy1,tmp);
        copyColumn(xCopy2,tmp,xCopy2.getCol()-1,tmp.getCol()-1);

        return tmp;
}

// Remove extra space (all zero rows) and return transpose
Matrix removeSpace(Matrix &x,int n){
        Matrix tmp;
        int j=0;

        // transpose matrix and return non-zero data
        x=transpose(x);
        tmp.createMatrix(x.getRow(),(int)((n+1)/2));
```

```
	for(int i=0;i<n;i+=2){
		copyColumn(x,tmp,i,j);
		j++;
	}

	return tmp;
}

// Calculate accuracy of classification
double wAccuracy(Matrix &rX, int nc){
	double tru=0.0;
	double tot=(double)rX.getRow();

	// sum True Positive and True Negative values
	for(int i=0;i<rX.getRow();i++){
		if(rX(i,nc-1)==rX(i,nc)){
			tru=tru+1.0;
		}
	}

	return tru/tot;// divide by total
}

// Calculate True Positive, False Negative, True Negative, False Positive,
// Sensitivity, Specificity, precision, and recall
Matrix ruleEval(Matrix &xC){
	Matrix tmp;

	// calculate total number of TP, FN, TN, and FP results
	tmp.createMatrix(1,8);
	for(int i=0;i<xC.getRow();i++){
		if(xC(i,xC.getCol()-1)==1){
			if(xC(i,xC.getCol()-2)==1){
				tmp(0,0)+=1.0;// (1,1) TP
			}else{
				tmp(0,2)+=1.0;// (0,1) FP
			}
		}else{
			if(xC(i,xC.getCol()-2)==1){
				tmp(0,3)+=1.0;// (1,0) FN
			}else{
				tmp(0,1)+=1.0;// (0,0) TN
			}
		}
	}

	// determine sensitivity--recall, specificity, and precision
	tmp(0,4)=tmp(0,0)/(tmp(0,0)+tmp(0,3));// sensitivity--recall
	tmp(0,5)=tmp(0,1)/(tmp(0,1)+tmp(0,2));// specifity
	if(tmp(0,0) >0 & tmp(0,2)>0){
		tmp(0,6)=tmp(0,0)/(tmp(0,0)+tmp(0,2));// precision
	}
	tmp(0,7)=tmp(0,4);// recall

	return tmp;
}
```