CPSC 340 Assignment 2 (due Sunday, February 5 at 11:59pm)

K-Nearest Neighbours, Random Forests, K-Means, Density-Based Clustering

Instructions

Rubric: {mechanics:3}

IMPORTANT!!!!! Before proceeding, please carefully read the general homework instructions at `https://github.ubc.ca/cpsc340/home/blob/master/homework_instructions.md`. You need to be signed in to github.ubc.ca in order to view this file. If you can't sign in, email Mike.

Other notes:

- We use blue to highlight the deliverables that you must answer/do/submit with the assignment.
- As requested, the number of points per questions are given below. if you see something like "Rubric: {code:3, reasoning:3}" you can interpret it as "this part is worth 6 points: 3 for code and 3 for the written portion." The points for "mechanics" above are for following the homework instructions.
- Some of the provided code uses `plt.show` to render images to your screen. In some circumstances, matplotlib behaves weirdly and creates the figure behind other windows in a way that makes it hard to see. It then looks like the code is running indefinitely but actually it's just waiting for you to close the (elusive) figure. If this happens and it annoys you, you can change the calls to `plt.show` to `plt.savefig`, which just saves the image to a file instead of having it pop up on your screen.
- You may want to add sections to *main.py*. To do this, add an `elif` statement in the same format as the existing ones, but also make sure to add the new option to the set of `choices` at the top of the file. If you try to run an option that's not listed in `choices` you will get an error.
- If you're using Python 2.7, you'll need to grab the alternate data at `https://github.ubc.ca/cpsc340/home/tree/master/assignments/hw2/data_python2.zip?raw=true`, which is Pickled using an older Pickle protocol. You don't have to overwrite the old data; once you copy in the Python 2 data into a directory called *data_python2*, you can tell the code to use it by changing the `DATA_DIR` variable at the top of *utils.py*.

1 K-Nearest Neighbours

In this question we revisit the *citiesSmall* dataset from the previous assignment.[1] In this dataset, nearby points tend to receive the same class label because they are part of the same state. This indicates that a k-nearest neighbours classifier might be a better choice than a decision tree. The file *knn.py* has implemented the training function for a k-nearest neighbour classifier (which is to just memorize the data).

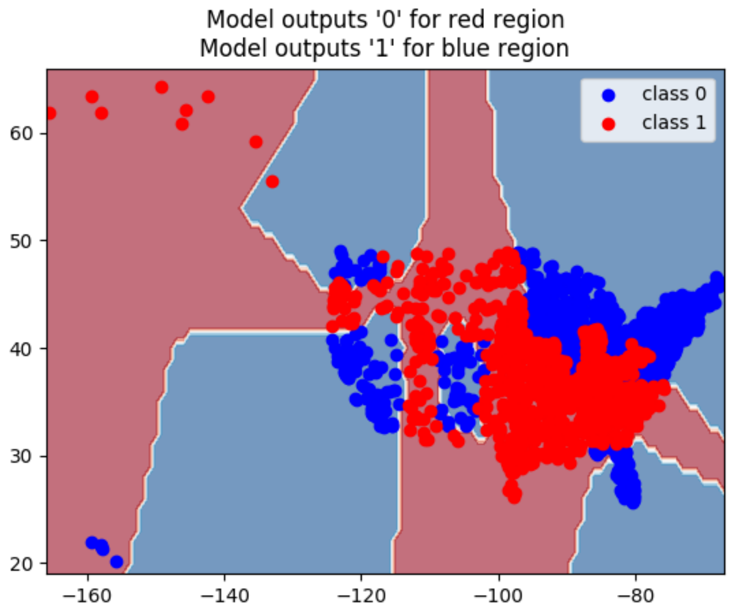
[1]For those who had trouble unpickling the data, you can use the alternate version that you used for hw1.

1.1 KNN Prediction

Rubric: {code:3, reasoning:3}

Fill in the *predict* function in *knn.py* so that the model file implements the k-nearest neighbour prediction rule. You should Euclidean distance, and may numpy's *sort* and/or *argsort* functions useful. You can also use *utils.euclidean_dist_squared*, which computes the squared Euclidean distances between all pairs of points in two matrices.

1. Hand in your predict function.
2. Report the training and test error obtained on the *citiesSmall* dataset for $k = 1$, $k = 3$, and $k = 10$.
3. Hand in the plot generated by *utils.plot_2dclassifier2Dplot* on the *citiesSmall* dataset for $k = 1$.
4. Why is the training error 0 for $k = 1$?
5. If you didn't have an explicit test set, how would you choose k?

1. knn.py
2. Training error for $k = 1$ is 0, Testing error for $k = 1$ is 0.0645
   Training error for $k = 3$ is 0.06, Testing error for $k = 3$ is 0.092
   Training error for $k = 10$ is 0.2, Testing error for $k = 10$ is 0.23
3. 

4. When $k = 1$ the training error is 0 as the closest cities tend to be in the same state and hence have the same classifier.
5. If we didnt have an explicit test set, we could choose k by looking for the smallest distance between points with different classifiers in the given data, and then evaluate its *position* including points with the same classifier and then using $(position - 1)$ as k.

1.2 Condensed Nearest Neighbours

Rubric: {code:3, reasoning:5}

The dataset *citiesBig1* contains a version of this dataset with more than 30 times as many cities. KNN can obtain a lower test error if it's trained on this dataset, but the prediction time will be very slow. A common strategy for applying KNN to huge datasets is called *condensed nearest neighbours*, and the main idea is to only store a *subset* of the training examples (and to only compare to these examples when making predictions). If the subset is small, then prediction will be faster.

The most common variation of this algorithm works as follows:

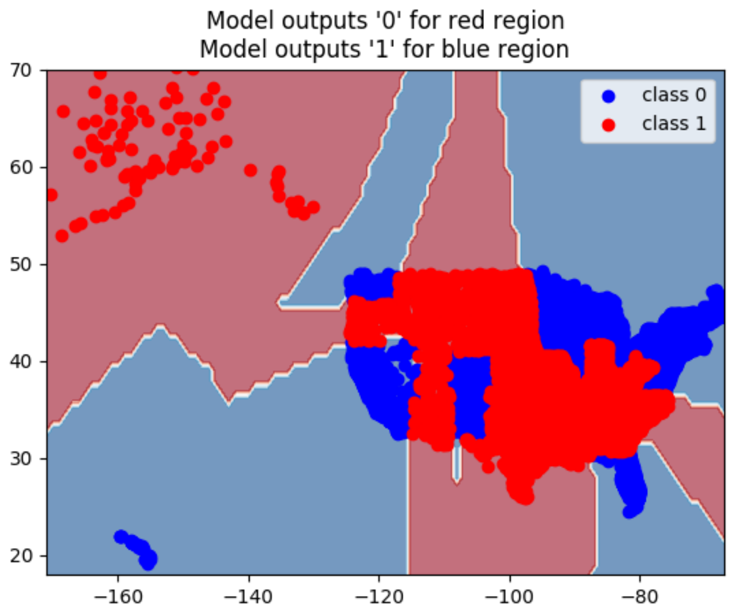
```
initialize subset with first training example;
for each subsequent training example do
    if the example is incorrectly classified by the KNN classifier using the current subset then
        add the current example to the subset;
    else
        do not add the current example to the subset (do nothing);
    end
end
```

Algorithm 1: Condensed Nearest Neighbours

Implement the *condensed nearest neighbours* algorithm as described above in a file called *cnn.py* with *fit* and *predict* functions. Note: if the size of the subset is less than k, you can take all examples in the subset as the current "neighbours".

1. Hand in your *cnn.py* code.
2. Report the training and testing errors, as well as the number of variables in the subset, on the *citiesBig1* dataset with $k = 1$.
3. Hand in the plot generated by *utils.plot_2dclassifier* on the *citiesBig1* dataset for $k = 1$.
4. Why is the training error with $k = 1$ now greater than 0?
5. If you have s examples in the subset, what is the cost of running the predict function on t test examples in terms of n, d, t, and s?
6. Try out your function on the dataset *citiesBig2*. Why are the test error *and* training error so high (even for $k = 1$) for this method on this dataset?

1. cnn.py
2. Training error = 0.00753, Testing error = 0.01757
   Number of variables = 457



3.

4. The training error is now greater than 0 since the final model dataset doesnt include all the points and hence they are further spaced apart, so the closest city when predicting is not necessarily in the same state and may have a different classifier.

5. $cost = O(st)$

6. For citiesBig2 the training and testing errors are higher since predictions made with the early subset were probably the same by chance and hence valuable data points were not

2 Random Forests

The dataset *vowels* contains a supervised learning dataset where we are trying to predict which of the 11 "steady-state" English vowels that a speaker is trying to pronounce.

2.1 Random Trees

Rubric: {code:4, reasoning:2}

The functions *decision_tree.py* and *decision_stump.py* implement a (non-random) decision tree that is built using greedy recursive splitting and information gain. The variant *random_tree.py* calls a *random_stump.py* (which is not included) to fit a random decision tree.

1. Make a plot of the training error and the test error for the *decision_tree* model, as the *max_depth* parameter is varied from 1 to 15.

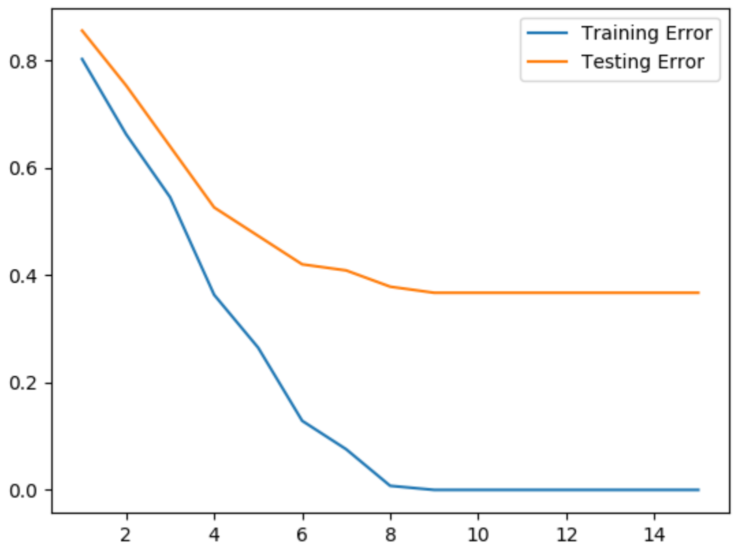
2. Why does the *decision_tree* function terminate if you set the *max_depth* parameter to ∞?

3. Copy the *decision_stump.py* file to a new file called *random_stump.py*. Modify the *random_stump.py* so that it only considers $\lfloor\sqrt{d}\rfloor$ randomly-chosen features.[2] Hand in the new training function and make a

[2]The notation $\lfloor x\rfloor$ means the "floor" of x, or "x rounded down". You can compute this with `np.floor(x)` or `math.floor(x)`.

plot of the training and test error of the now-working *random_tree* model as max_depth is varied from 1 to 15 with this method (it should not be the same every time, so just include one run of the method).

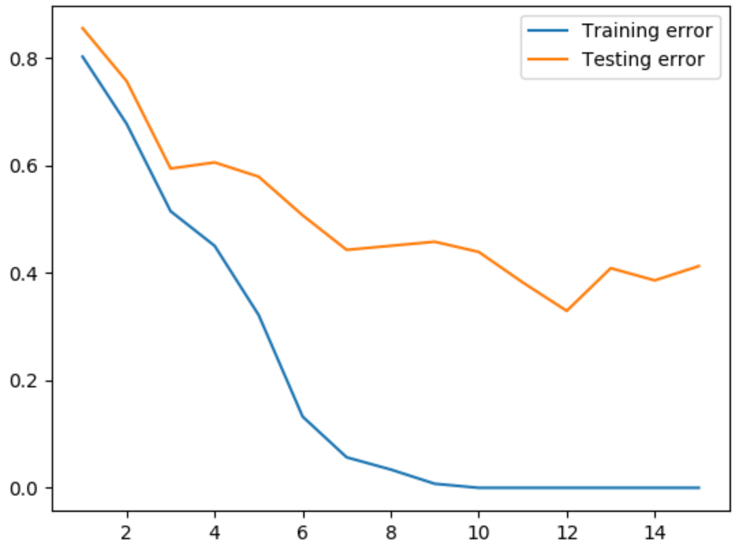
4. Make a third training/test plot where you use the *decision_tree* model but for each max_depth you train on a different bootstrap sample of the training data (but evaluate the training error on the original training data).



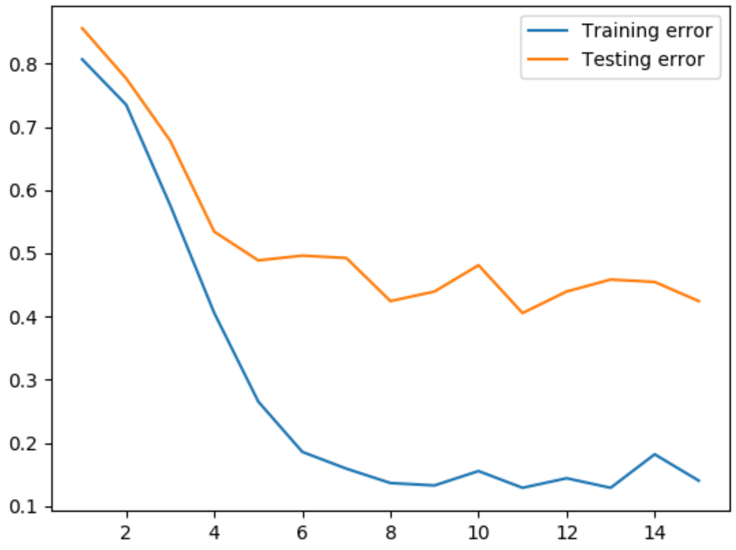
1.

2. Although the depth never decreases with $max\_depth = \infty$, the decision tree recursion does reach a point with only one element in the subset of data to be trained on, and hence with no split variable returned it satisfies the base condition eventually to end recursion.

3. • random_stump.py



•



4.

2.2 Random Decision Forests

Rubric: {code:4, reasoning:2}

The function *decision_forest.py* model repeatedly calls *decision_tree* to fit a set of models, and for prediction averages their results.

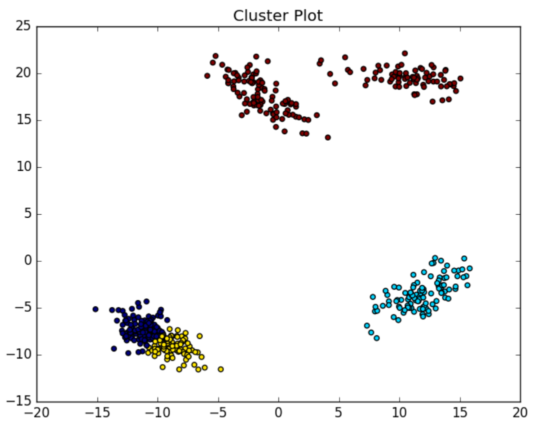
1. Report the test error of the *decision_forest* classifier with a depth of ∞ and 50 trees.

2. Report the test error of the *decision_forest* classifier with a depth of ∞ and 50 trees, if each tree is trained on a different boostrap sample.
3. Report the test error of the *decision_forest* classifier with a depth of ∞ and 50 trees, if you train on the original dataset but use *randomTree* instead *decisionTree* to fit the models.
4. Report the test error of the *decision_forest* classifier with a depth of ∞ and 50 trees, if you use *random_tree* and each tree is trained on a different bootstrap sample. Hand in your modified *decision_forest.py* file (which is now a random forest model).
5. What is the effect of the two ingredients of random forests, bootstrapping and random splits, on the performance on this dataset?

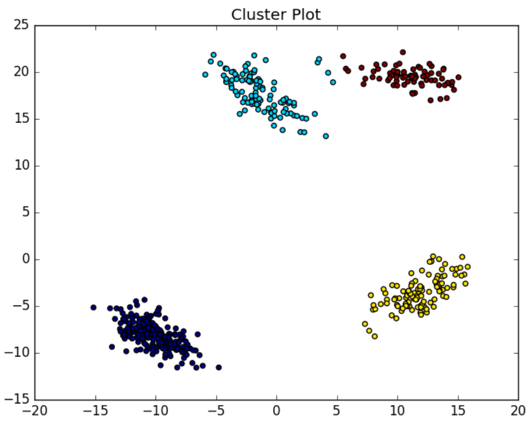
1. Testing error for $decision\_forest = 0.3674$
2. Testing error for $decision\_forest$ with bootstrapping $= 0.2614$
3. Testing error for $random\_forest = 0.1780$
4. Testing error for $random\_forest$ with bootstrapping $= 0.1629$
5. Instead of going through all features, random splits fit models quicker acting on fewer features which are randomly chosen which greatly improves performance in case there are many features. Random forests use bootstrap samples to fit and average out predictions returned by multiple trees returning a more accurate answer than just one decision tree.

3 K-Means Clustering

If you run `python main.py -q 3.1`, it will load a dataset with two features and a very obvious clustering structure. It will then apply the k-means algorithm with a random initialization. The result of applying the algorithm will thus depend on the randomization, but a typical run might look like this:



But the ‘correct’ clustering (that was used to make the data) is this:



(Note that the colours are arbitrary.)

3.1 Selecting among Initializations

Rubric: {code:4}

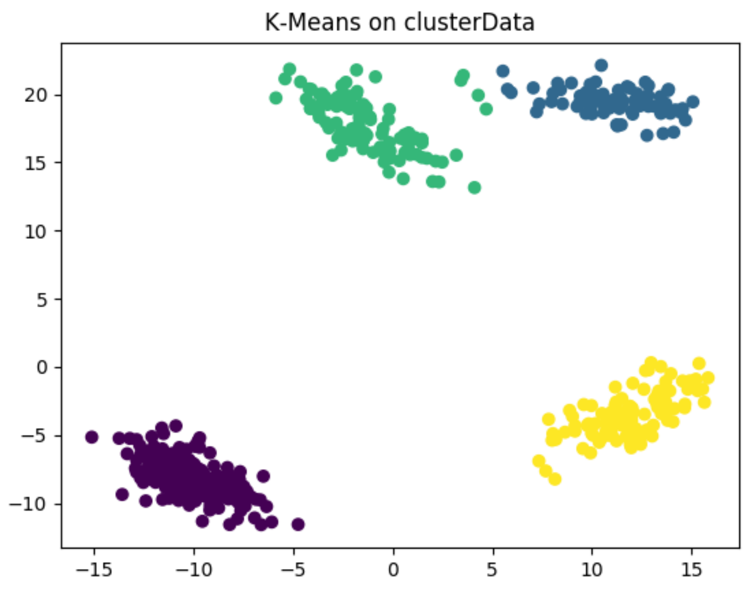
If you run the demo several times, it will find different clusterings. To select among clusterings for a *fixed* value of k, one strategy is to minimize the sum of squared distances between examples x_i and their means w_{c_i},

$$f(w_1, w_2, \dots, w_k, c_1, c_2, \dots, c_n) = \sum_{i=1}^{n} \|x_i - w_{c_i}\|_2^2 = \sum_{i=1}^{n}\sum_{j=1}^{d}(x_{ij} - w_{c_ij})^2.$$

where c_i is the *index* of the closest mean to x_i (an integer), and w_{c_i} is the *location* of the closest mean to x_i (a vector in $\mathbb{R}^d$). This is a natural criterion because the steps of k-means alternately optimize this objective function in terms of the w_c and the c_i values.

1. In the *kmeans.py* file, add a new function called *error* that takes the same input as the *predict* function but that returns the value of this above objective function. Hand in your code.
2. Using the *utils.plot_2dclustering* function, output the clustering obtained by running k-means 50 times (with $k = 4$) and taking the one with the lowest error.

1. kmeans.py



2.

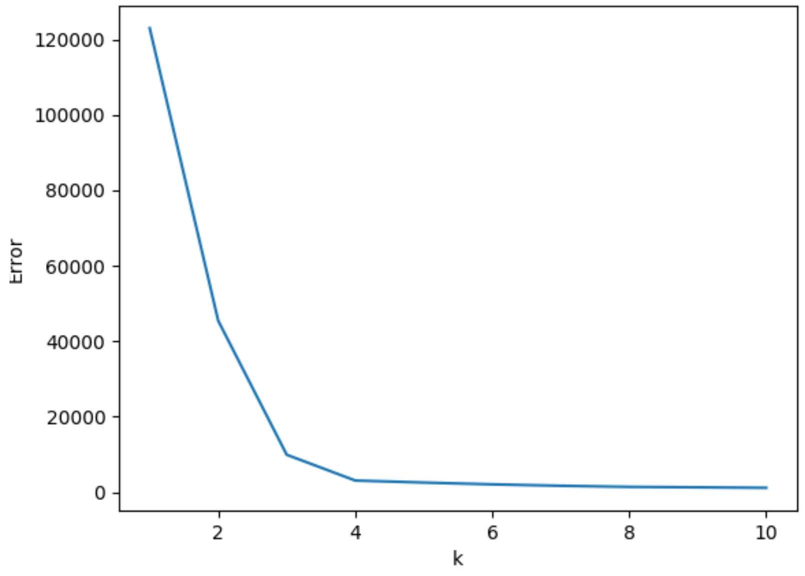
3.2 Selecting k

Rubric: {reasoning:4}

We now turn to the much-more-difficult task of choosing the number of clusters k.

1. Explain why the above objective function cannot be used to choose k.
2. Explain why even evaluating this objective function on test data still wouldn't be a suitable approach to choosing k.
3. Hand in a plot of the minimum error found across 50 random initializations, as you vary k from 1 to 10.
4. The *elbow method* for choosing k consists of looking at the above plot and visually trying to choose the k that makes the sharpest "elbow" (the biggest change in slope). What values of k might be reasonable according to this method? Note: there is not a single correct answer here; it is somewhat open to interpretation and there is a range of reasonable answers.

1. The above objective function cannot be used to choose k, since it assumes the value of k is already known.
2. Evaluating the objective function on test data still wouldnt be a suitable approach since we do not know about presence of outliers which will most likely give sub-optimal clusters.



3.

4. Visually the sharpest elbows seem to be at $k = 3, 4$ which would be ideal for this clusterData.

3.3 k-Medians

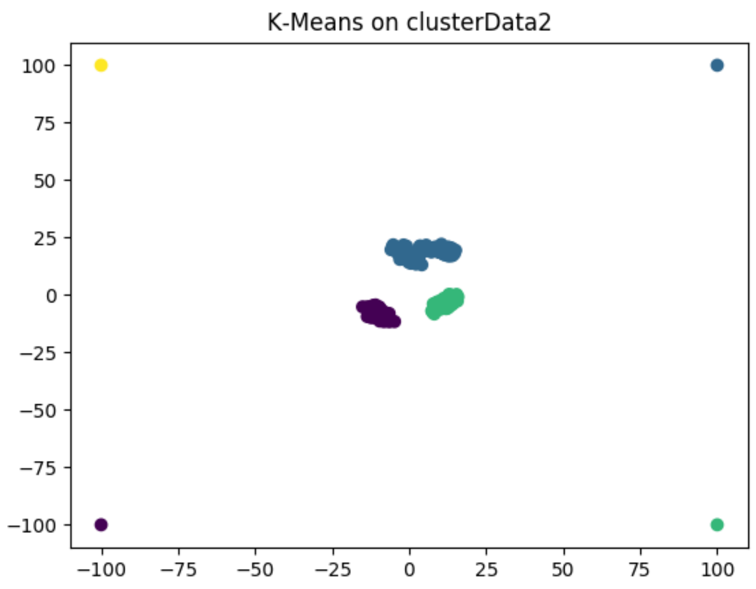
Rubric: {reasoning:3, code:3}

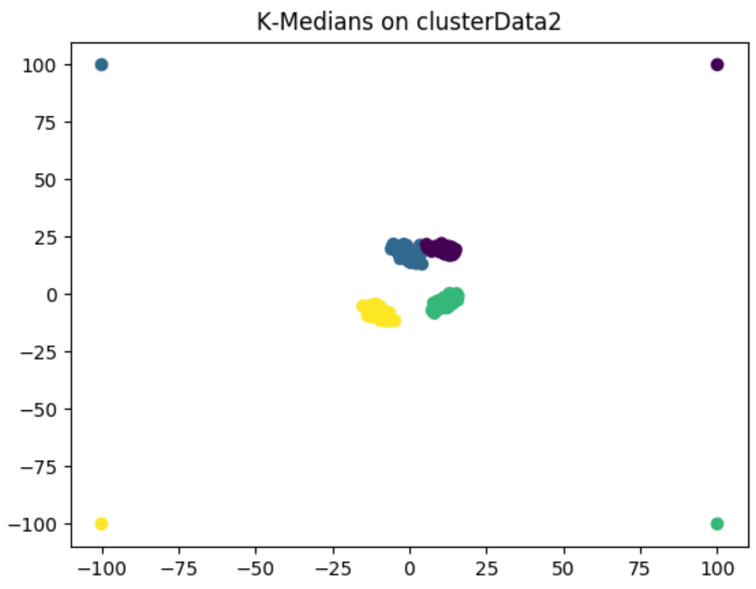
The data in *clusterData2* is the exact same as the above data, except it has 4 outliers that are very far away from the data.

1. Using the *plot_2dclustering* function, output the clustering obtained by running k-means 50 times (with $k = 4$) on *clusterData2* and taking the one with the lowest error.
2. What values of k might be chosen by the elbow method for this dataset?
3. Implement the *k-medians* algorithm, which assigns examples to the nearest w_c in the L1-norm and updates the w_c by setting them to the "median" of the points assigned to the cluster (we define the d-dimensional median as the concatenation of the median of the points along each dimension). Hand in your code.
4. Using the L1-norm version of the error (where c_i now represents the closest median in the L1-norm),

$$f(w_1, w_2, \dots, w_k, c_1, c_2, \dots, c_n) = \sum_{i=1}^{n} \|x_i - w_{c_i}\|_1 = \sum_{i=1}^{n} \sum_{j=1}^{d} |x_{ij} - w_{c_i j}|,$$

what value of k would be chosen by the elbow method under this strategy?

1. 

2. Since means are sensitive to outliers, the elbow method would probably give us $k = 7, 8$ for this dataset as there are 4 outliers.
3. - kmedians.py
   - 

4. Since medians are not sensitive to outliers, the elbow method gives us $k = 3, 4$.

4 Vector Quantization and Density-Based Clustering

In this question we'll look at vector quantization, an alternative way that k-means is commonly used. Then we'll start to explore alternative clustering methods like density-based clustering.

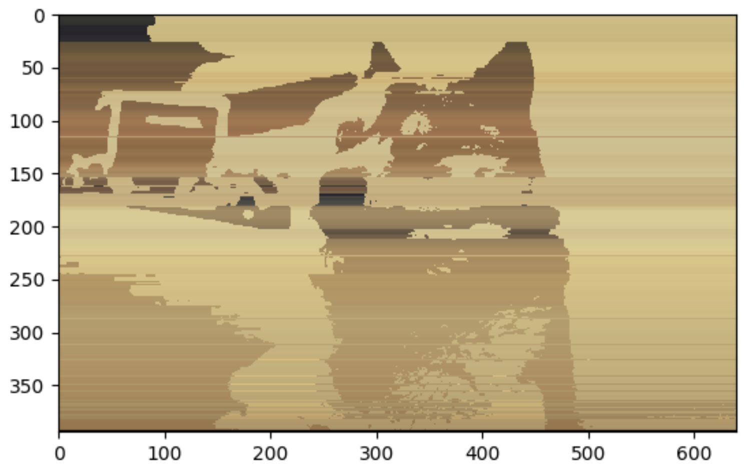
4.1 Image Colour-Space Compression

Rubric: {code:2}

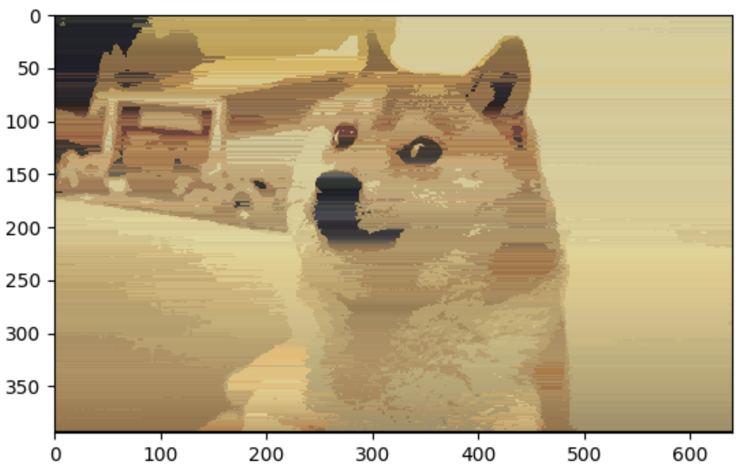
Discovering object groups is one motivation for clustering. Another motivation is vector quantization, where we find a prototype point for each cluster and replace points in the cluster by their prototype. The dataset *dog* contains a 3D-array I representing the RGB values of a picture of a dog. You can view the picture by using the *plt.imshow* function. Create a file called *quantize_image.py* with the function `def quantize_image(I,b)` where I is an image, b is the number of bits used to represent the colour-space, and returns a version of the image where each pixel's colour can be encoded using only b bits by replacing the original pixel's colour with the nearest prototype. You should find the prototypes using the provided k-means code, and recall that with b bits you can represent 2^b numbers, so the number of clusters should be $k = 2^b$. Hint: you may want to look up the *np.reshape* command.

1. Hand in your *quantizeImage* function.
2. Show the image obtained if you encode the colours using 1, 2, 4, and 6 bits.

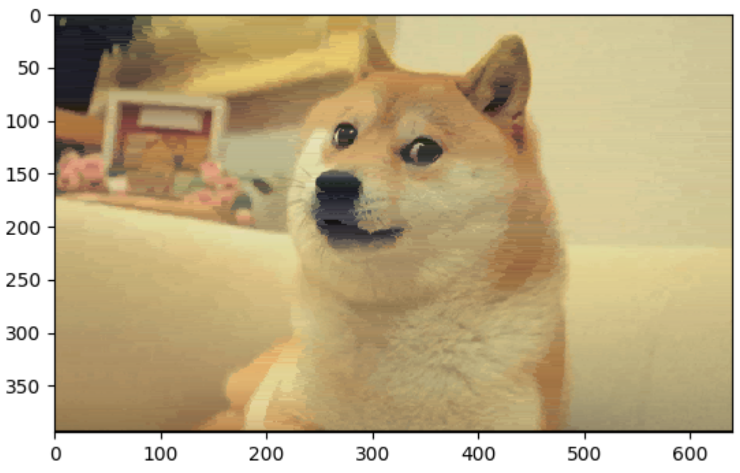
1. quantize_image.py



2. $B = 1$


0
50
100
150
200
250
300
350
0
100
200
300
400
500
600

$B = 2$


0
50
100
150
200
250
300
350
0
100
200
300
400
500
600

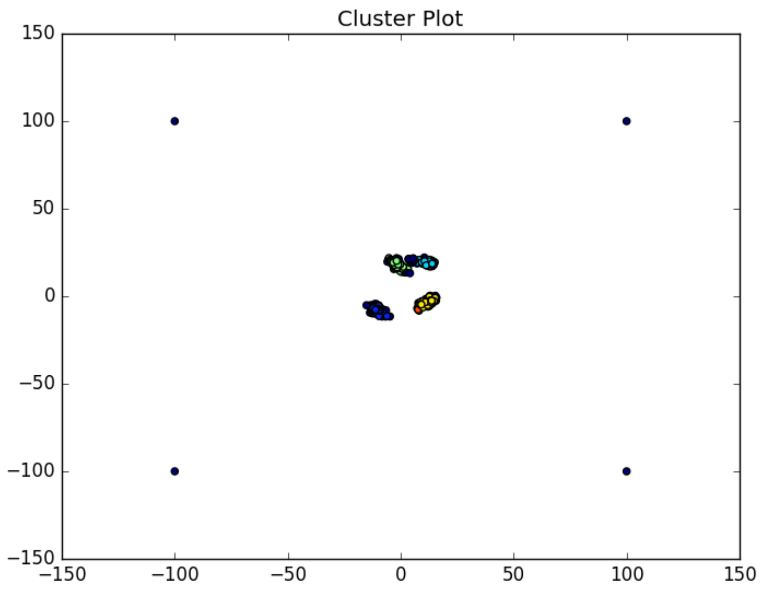
$B = 4$

$B = 6$

4.2 Effect of Parameters on DBSCAN

Rubric: {code:2}

If you run `python main -q 4.2`, it will apply the basic density-based clustering algorithm to the dataset from the previous question. The final output should look like this:



Even though we know that each object was generated from one of four clusters (and we have 4 outliers), the algorithm finds 6 clusters and does not assign some objects to any cluster. However, the assignments will change if we change the parameters of the algorithm. Find and report values for the two parameters (*radius2* and *minPts*) such that the density-based clustering method finds:

1. The 4 "true" clusters.
2. 3 clusters (merging the top two, which also seems like a reasonable interpretation).
3. 2 clusters.
4. 1 cluster.

Unimportant note: we formulated things in terms of *radius2*, the radius squared, which is why we then take its square root in *dbscan.py*.

1. $radius2 = 4$, $minPts = 3$
2. $radius2 = 16$, $minPts = 3$
3. $radius2 = 169$, $minPts = 3$
4. $radius2 = 256$, $minPts = 3$

4.3 K-Means vs. DBSCAN Clustering

Rubric: {reasoning:2, code:2}

If you run `python main -q 4.3`, it will load a dataset containing 85 attribute values for 50 animals. It will then apply a k-means clustering to the animals, and report the resulting clusters. The exact clustering will depend on the initialization of k-means and the value of k, but below is the result of one of the runs:

- Cluster 1: killer+whale blue+whale hippopotamus humpback+whale seal walrus dolphin
- Cluster 2: elephant ox sheep rhinoceros buffalo giant+panda pig cow
- Cluster 3: skunk mole hamster squirrel rabbit mouse raccoon
- Cluster 4: antelope horse moose spider+monkey gorilla chimpanzee giraffe zebra deer
- Cluster 5: grizzly+bear beaver dalmatian persian+cat german+shepherd siamese+cat tiger leopard fox bat wolf chihuahua rat weasel otter bobcat lion polar+bear collie

Some of these groupings make sense (cluster 1 contains fairly-large animals that live in or near the water) while others do not (grizzly bears and bats are both in cluster 5).

Modify this demo to use the density-based clustering method, and report the clusters obtained if you set *minPoints* to 3 and the radius such that it finds 5 clusters.

$radius2 = 15$

- Cluster 1: antelope horse moose ox sheep giraffe buffalo zebra deer pig cow
- Cluster 2: dalmatian persian+cat german+shepherd siamese+cat mole tiger leopard fox hamster squirrel rabbit wolf chihuahua rat weasel bobcat mouse collie
- Cluster 3: hippopotamus elephant rhinoceros
- Cluster 4: blue+whale humpback+whale seal walrus dolphin
- Cluster 5: spider+monkey gorilla chimpanzee