CPSC 340 Assignment 3 (due October 23rd)

Clustering, Item Recommendation, Linear Regression

- You can work in groups on the assignments. However, please hand in your own assignments and state the group members that you worked (as well as other sources of help like online material).
- Place your name and student number on the first page, and submit all answers as a single PDF file to handin.
- For questions that ask for code, you should include the relevant parts of the code in the appropriate place in the PDF file.
- Please organize your submission sequentially according to the sections used in this document.
- All Sections (1-5) are equally weighted.
- There may be updates/clarifications to the assignment after the first version is put online. Any modifications will be marked in red.

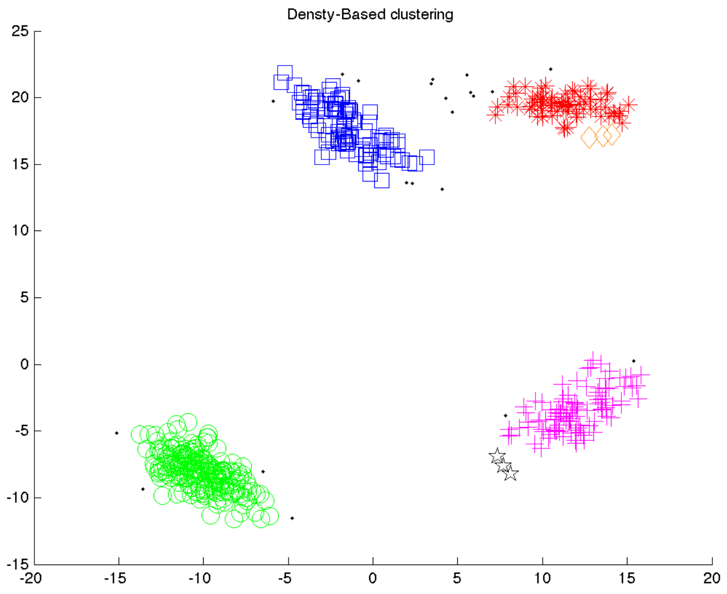
0 Unnoffiial Course Evaluation

To help improve the course as we go along, or to suggest of how things could be done differently, please fill out the survey here:
`https://survey.ubc.ca/surveys/37-7d0090012c11ea5c07f0bca610f/cpsc340-asst-3`

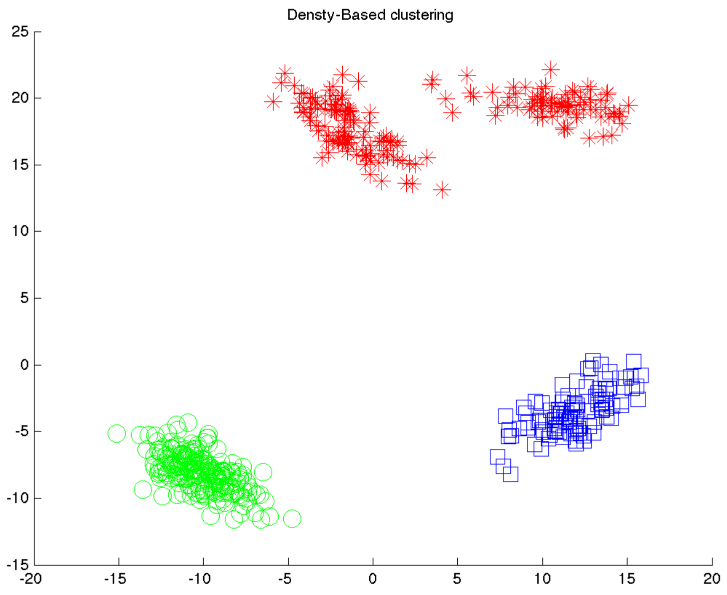
1 Non-Parametric Clustering

1.1 Effect of Parameters on DBSCAN

If you run the function *example_DBscan*, it will apply the basic density-based clustering algorithm to the dataset from the previous assignment. The final output should look like this:



Even though we know that each object was generated from one of four clusters, 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 of the density-based clustering algorithm such that it finds the correct 4 clusters and assigns all points to their appropriate cluster. Further, find and report values for the two parameters such that the top two clusters are merged into one cluster, so that we have 3 clusters as in the figure below:



1.2 K-Means vs. DBSCAN Clustering

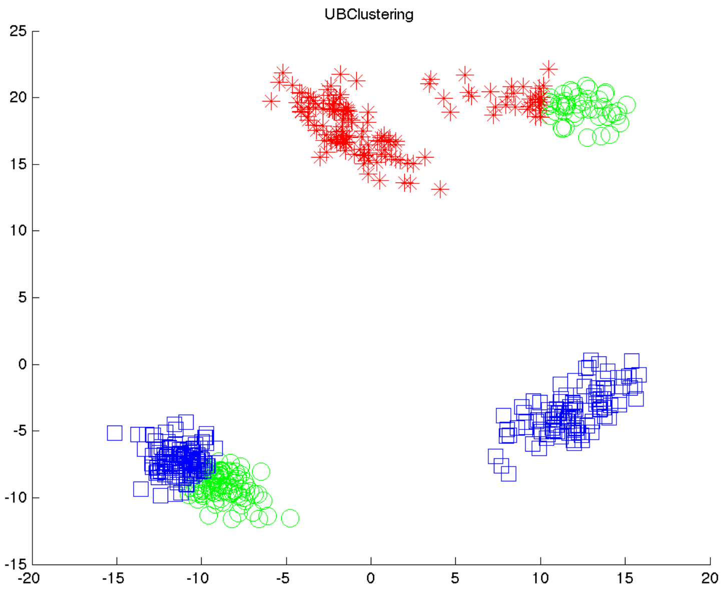
If you run the function *example_animals*, it will load a dataset containing 85 attribute values for 50 animals. It will then apply a k-means clustering to the animals, and report the resulting clusters. The exact clustering will depend on the initialization of k-means and the value of k, but below is 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.

1.3 UBClustering Algorithm

If you run the function *example_UBClustering.m*, it will load our usual synthetic clustering data. It will then apply k-means clustering 20 times, and use the mode of these clusterings as the final result. Due to the 'label switching' problem, this typically produces non-sensical clusterings like this one:



Modify this demo so that it puts two points in the same cluster if they appear together in at least 50% of the clusterings (merging the clusters of points that satisfy this property). Hand in your code and the plot that you obtain with this strategy.

Hint: Try to see if you can adapt the structure of the 'for' loop and the 'expand' functions from the DBSCAN algorithm in order to implement this new algorithm.

2 Item Recommendation

The file *newsgroups.mat* contains our usual newsgroups dataset, except that all the objects are placed together in a big matrix X, and (since we are focusing on unsupervised learning) there is no label y. In this question, we're going to build a 'word' recommendation system. That is, given a word, return a list of other words that are relevant (this could be used to direct a user to other relevant topics).

2.1 Amazon Recommendation Algorithm

In class, we discussed the recommendation algorithm used by Amazon. For each product/word j, it works with a vector x_j containing the users/posts that contain j. For the newsgroups data, this is simply $X(:,j)$. To make recommendations for a product/word i, it returns the K words that have the highest *cosine similarity*,

$$\cos(x_i, x_j) = \frac{x_i^T x_j}{\|x_i\|\|x_j\|}.$$

Write a function 'recommender', that has the following behaviour:

1. Calling 'model = recommender(X,K)' will do any 'training' the model requires (if necessary) and store relevant quantities used by the model to make predictions.
2. Calling 'wordNumbers = model.predict(model,j)' will return the K word numbers whose vectors have the highest cosine similarity with word number 'j'. Make sure to exclude 'j' from the output.

Hand in your implementation of the 'recommender' function. For each of the first 5 words, list the recommended words when $K = 5$.

2.2 Fast Recommendation with Sparse Data

Let n be the number of rows in X and d be the number of columns. Computing a norm of a column costs $O(n)$, while computing the inner product between columns also costs $O(n)$. Thus, to make a recommendation the worst case cost of computing the cosine similarity between all d rows is $O(nd)$. This would be too slow for a company like Amazon, who has millions or users (n) and millions of products (d).

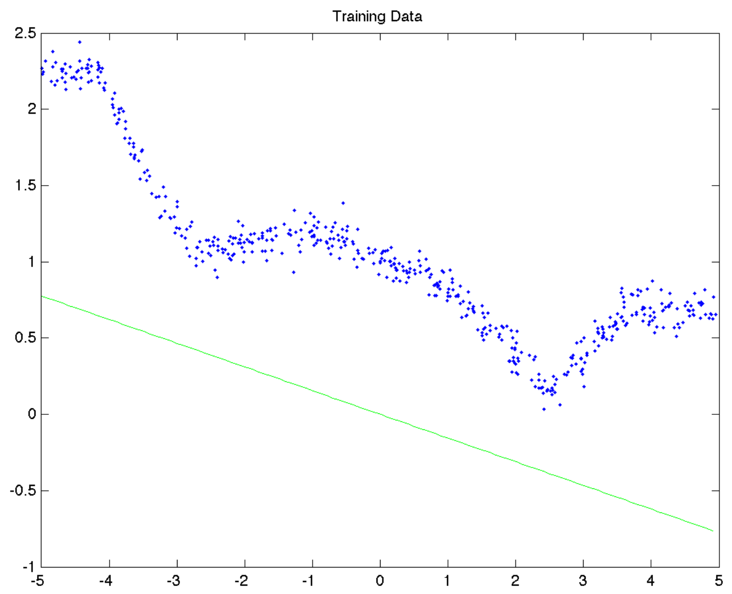
Consider an implementation that maintains the following two data structures for each product:

1. A *sorted list* of the user identification numbers for the users that bought the product.
2. An *list* of the products that have non-zero similarity with the product (i.e., have been bought by one of the same users).

Let u be the maximum length of the first (sorted) list and p be the maximum length of the second list. What is the worst-case cost of making an item recommendation in terms of u and p?

3 Linear Regression and Change of Basis

The script *example_basis* loads a one-dimensional regression dataset and displays the result of applying a simple linear regression model. Unfortunately, this is an awful model of the data:



3.1 Linear Regression with a Bias Variable

Write a new function, *simpleLeastSquares*, that has the same input/model/predict format as the *leastSquares* function, but that includes a *bias* variable w_0, and makes predictions using the model

$$y_i = wx_i + w_0.$$

Hand in your new function and the updated plot.

3.2 Polynomial Basis

Centering the target variables y_i improves the prediction substantially, but the model is still problematic because the target seems to be is a non-linear function of the input. Write a new function, *leastSquares-Basis(x,y,degree)*, that takes a data vector x (i.e., assuming we only have one feature) and the polynomial order *degree*. The function should perform a least squares fit based on a matrix *Xpoly* where each of its rows contains the values $(X_i)^j$ for $j = 0$ up to *degree*. E.g., *leastSquaresBasis(x,y,3)* should form the matrix

$$Xpoly = \begin{bmatrix} 1 & x_1 & (x_1)^2 & (x_1)^3 \\ 1 & x_2 & (x_2)^2 & (x_2)^3 \\ \vdots & & & \\ 1 & x_n & (x_n)^2 & (x_N)^3 \end{bmatrix},$$

and fit a least squares model based on it. Hand in the new function and a 3 by 3 plot showing the fit for $degree = 0$ up to $degree = 8$.

Hints: You can use the *subfigure* command to construct a figure containing multiple plots. Don't forget that you need to do the same transformation to the test data. You may want to write a new function *polyBasis* that you can use for both the training and testing data. Also, since we include a column of '1' values (for $degree = 0$) you do not have to center the y_i values.

3.3 Choosing the Basis

Using the first 250 examples as a training set and the remaining 250 examples as a validation set, report the training and validation errors for models with degrees 0 through 20. Use the *averaged squared error* as the performance measure. How does the degree of the polynomial affect the fundamental trade-off in machine learning?

Hint: use a 'for degree = 0:20' loop, and use an 'fprintf' statement to output the training and testing error for each degree.

4 Radial Basis Functions and Regularization

The function *example_rbf* loads and displays a training data set $\{X, y\}$ and a corresponding test data set $\{Xtest, ytest\}$. Subsequently, it displays the performance of least squares using non-parametric radial basis function (RBFs). The performance of the model becomes better as the RBF parameter *sigma* is decreased, but the becomes substantially worse for low values.

4.1 Regularization

Using an RBF kernel leads to a much more complicated model, so we might expect to do better through the use of regularization. Modify the function *leastSquaresRBF* to take an extra parameter *lambda*, and use an L2-regularized estimate of w. Explore whether you can achieve a lower test error by exploring different values of λ. Hand in your modified code.

4.2 Proper Training/Validation/Testing

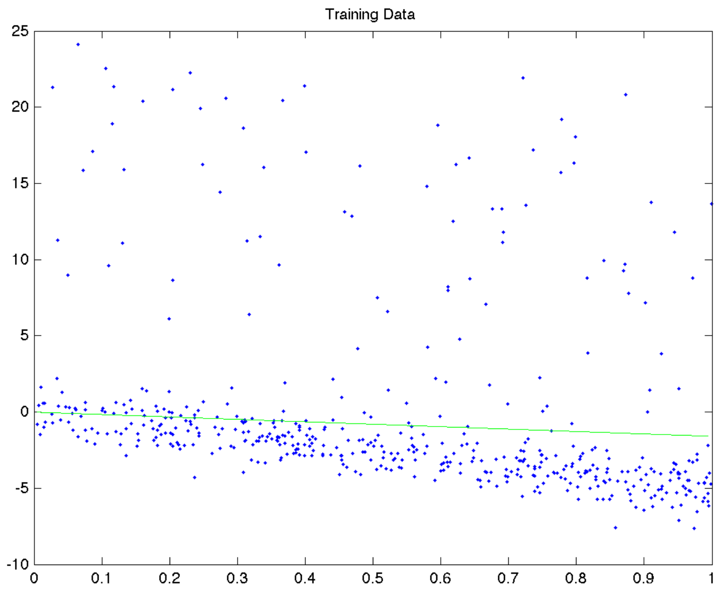
Unfortunately, this demo and exploring whether we can achieve a better test error by trying out different values of λ is violating the golden rule of machine learning, by looking at the test set multiple times. Make the following modifications to the demo:

1. Instead of training on the full training set and testing on the test set. Train on the first half of the training data and test on the second half of the training data (the 'validation' set).
2. Modify the demo to search over the 'hyper-parameters' σ and λ, and find the one that performs the best on the validation set. Measure performance based on the average absolute error. A reasonable range of λ values to search is $\{2^{-12}, 2^{-11}, 2^{-10}, \dots, 2^{2}\}$.
3. Once you have the best values of the hyper-parameters σ and λ, train on the full training set and compute the average absolute error on the test set.

Hand in your code, report the best values of σ and λ, and report the final test error estimate from this procedure.

5 Least Squares with Outliers

The script *example_outliers* loads a one-dimensional regression dataset that has a non-trivial number of 'outlier' data points. These points do not fit the general trend of the rest of the data, and pull the least squares model away from the main downward trend that most data points exhibit:



5.1 Weighted Least Squares in One Dimension

One of the most common variations on least squares is *weighted* least squares. In this formulation, we have a weight z_i for every training example. To fit the model, we minimize the weighted squared error,

$$\underset{w \in \mathbb{R}^d}{\arg\min} \frac{1}{2} \sum_{i=1}^{n} z_i (y_i - w^T x_i)^2.$$

In this formulation, the model focuses on making the error small for examples i where z_i is high. Similarly, if z_i is low then the model allows a larger error. By taking the derivative and equating it with zero, solve for the the solution w to one-dimensional weighted least squares problems.

5.2 Weighted Least Squares Fitting

Write a model function, *weightedLeastSquares(x,y,z)*, that implements the model from the previous question. Apply this model to the data containing outliers, setting $z = 1$ for the first 400 data points and $z = 0.1$ for the last 100 data points (which are the outliers). Hand in your function and the updated plot.

5.3 Smooth Approximation to the L1-Norm

Unfortunately, we typically do not know the identities of the outliers. In situations where we suspect that there are outliers, but we do not know which examples are outliers, it makes sense to use a loss function that is more robust to outliers. In class, we discussed using the sum of absolute values objective,

$$\underset{w \in \mathbb{R}^d}{\arg\min} \sum_{i=1}^{n} |y_i - w^T x_i|.$$

This is less sensitive to outliers than least squares, but it is non-differentiable and harder to optimize. Nevertheless, there are various smooth approximations to the absolute value function that are easy to optimize. One possible approximation is to use

$$|r| \approx \sqrt{r^2 + \epsilon},$$

for some small ϵ. This approximation becomes exact as ϵ goes to zero, but for any fixed ϵ the function will be differentiable. Using this approximation, we obtain an objective of the form

$$\operatorname*{arg\,min}_{w \in \mathbb{R}^d} \sum_{i=1}^{n} \sqrt{(y_i - w^T x_i)^2 + \epsilon},$$

which is smooth but less sensitive to outliers than the squared error. If we define f by

$$f(w) = \sum_{i=1}^{n} \sqrt{(y_i - w^T x_i)^2 + \epsilon},$$

derive the gradient ∇f of this function with respect to w. You should show your work but you do not have to express the final result in matrix notation.

5.4 Robust Regression

The function *example_gradient* is the same as *example_outlier*, except that it fits the least squares model using a *gradient* method. One advantage of this strategy is that it only costs $O(nd)$ for an iteration of the gradient method, which is faster than forming X^TX which costs $O(nd^2)$. Of course, we need to know the *number* of gradient iterations in order to precisely compare these two strategies, but for now we will assume that the number of gradient iterations is typically much less than d.

The usual input to a gradient method is a function that, given w, returns $f(w)$ and $\nabla f(w)$. See *funObj* in the *leastSquaresGradient* function for an example. Note that *leastSquaresGradient* also has a numerical check that the gradient code is approximately correct, since implementing gradients is often error-prone.

A second advantage of gradient-based strategies is that they are able to solve problems that do not have closed-form solutions, such as the formulation from the previous section. The function *robustRegression* has most of the implementation of a gradient-based strategy for fitting the robust regression model under the $\sqrt{r^2 + \epsilon}$ approximation. The only part missing is the function and gradient calculation inside the *funObj* code. Modify this function to implement the objective function and gradient based on the smooth approximation to the absolute value function (from the previous section). Hand in your code, as well as the plot when we set $\epsilon = 0.1$.

.