CPSC 340 Assignment 5 (due November 27)

Sparse Latent Factors, Recommender Systems, MDS, Neural Networks

- 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.

1 Sparse Latent-Factor Models

If run the function *example_faces* it will load a set of face images under different lighting conditions. It will then fit a PCA model (with $k = 10$) and display 5 figures:

- Random faces taken from the dataset (Figure 1).
- The average face (Figure 2).
- The principal components (Figure 3).
- The original examples x_i and the compressed varient z_i (Figure 4).
- The same random faces, after reconstructing them from their corresponding z_i (Figure 5).

The reconstructions tend to be reasonable given that they compress each 1024-pixel face down to just 10 numbers, although they often lack specific details and are less accurate for faces that do not look like the average face. In this model, some of the principal components are interpretable (e.g., they reflect different lighting conditions), but many of them are not. You can change the value of k in this script to see the effect of including/excluding principal components, if you increase k it will do a better job of reconstructing faces that do not look like the average face.

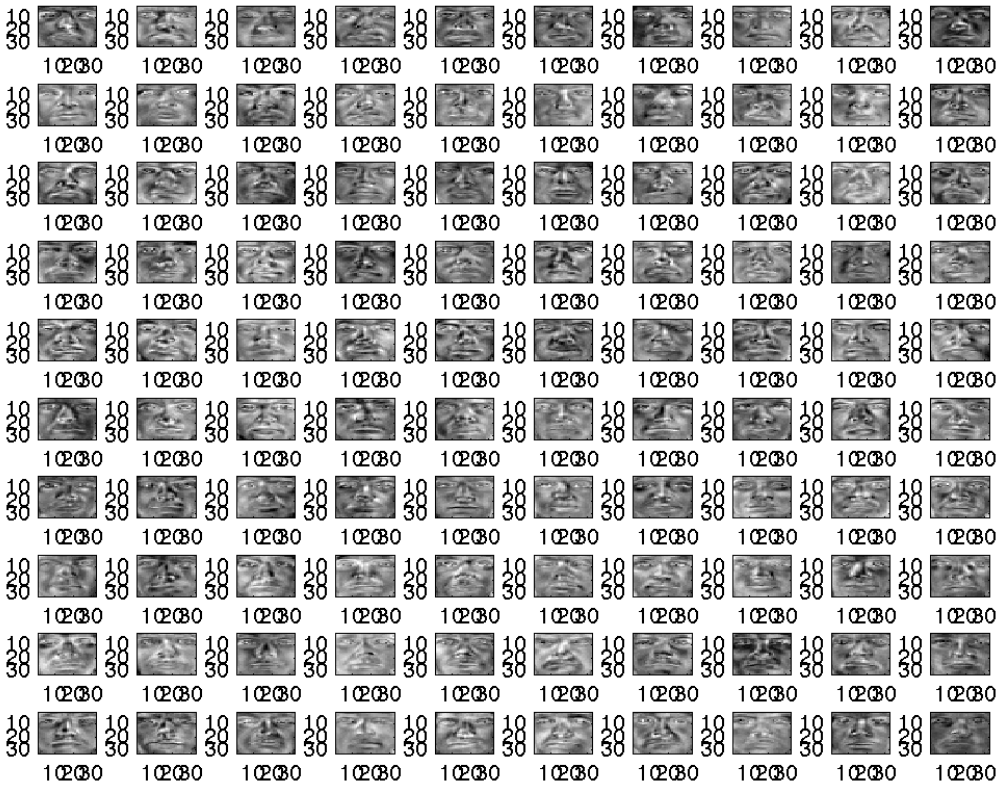
1.1 Uniqueness of Principal Components

If you re-run the script, you may get different principal components, even though all that changes between runs is the order of the training examples. What is the specific difference between the principal components that are obtained between different runs of the algorithm?

Answer:
Even though we've enforced that the rows of W are normalized and are mutually orthogonal and are fit sequentially, a further source of non-identifiability is that we can flip signs of W and Z and still get the same model. The difference obtained between different runs is from flipping signs in rows of W.

1.2 Non-Negative Matrix Factorization

If you replace the function *dimRedPCA* with *dimRedPCA_alternate*, then instead of using the SVD to compute the principal components it will compute them numerically by using a gradient method that alternates between updating W and Z. Note that this returns different principal components because it doesn't enforce any constraints on W, but (up to numerical accuracies) it will give an equivalent model. You would never actually use this method to fit a PCA model, but this optimization strategy generalizes to other models and it shows the learned principal components without the constraints. Below are the results obtained by *dimRedPCA_alternate* (right) with $k = 100$, and note that these look different from the usual PCA results because we aren't enforcing constraints on W (to make this plot, I initialized with samples from a standard normal divided by 10^{-5}, but in the a5.zip it uses samples directly. This leads to poor results for PCA, that look like random noise, but makes a better initialization for other methods):



A disadvantage of PCA is that the principal components tend to be dense (they are all non-zero). One of the first models to address this is non-negative matrix factorization, where we use the PCA objective but add non-negative constraints on W. Using *dimRedPCA_alternate* as a template, write a function *dimRedNMF*

that implements the non-negative matrix factorization (NMF) model. Hand in your code and hand in a plot of the latent factors (Figure 3) obtained when $k = 100$.

Hint: you need to make three changes. First, you need to change the initialization so that negative values of W and Z are set to 0. Second, you need to change the updates of W and Z to use an optimization method that includes the non-negativity constraint. You can use the function *findMinNN* minimizes a smooth function subject to non-negative constraints. Finally, you need to update the *compress* function so that it finds the non-negative Z minimizing the objective with W fixed.

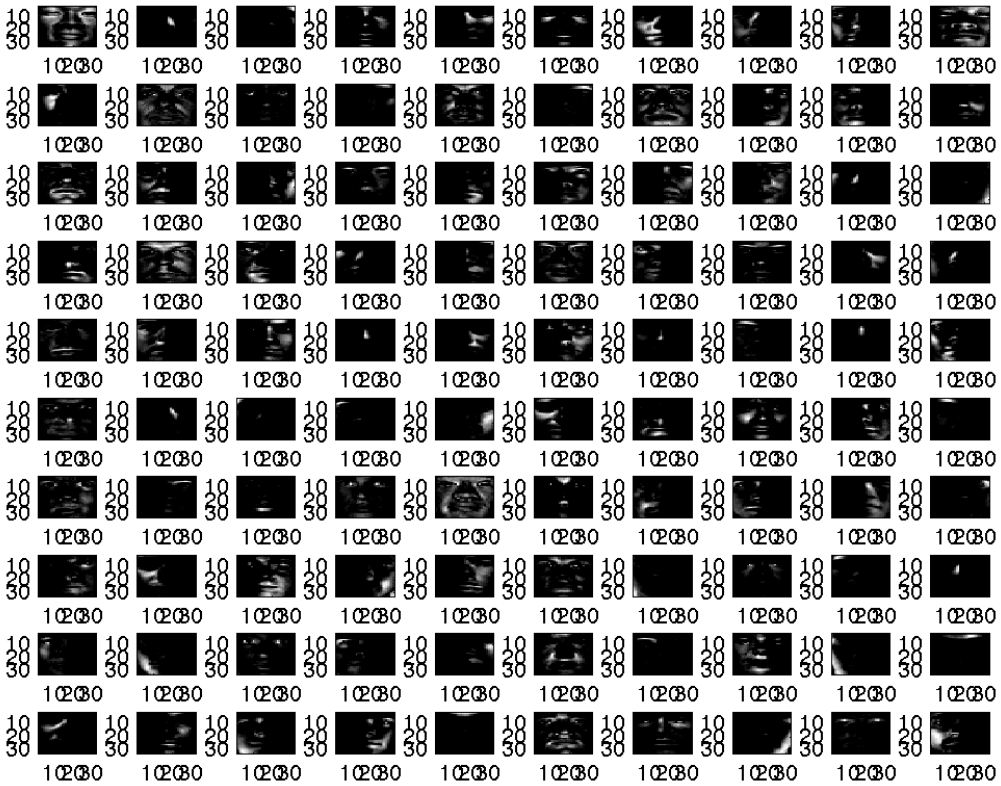
Answer:
The main *dimRedNMF* should look roughly like this:

```
function [model] = dimRedNMF_alternate(X,k)

[n,d] = size(X);

% Subtract mean
mu = mean(X);
X = X - repmat(mu,[n 1]);

% Initialize W and Z
W = randn(k,d);
Z = randn(n,k);

W(W < 0) = 0;
Z(Z < 0) = 0;

f = (1/2)*sum(sum((X-Z*W).^2));
for iter = 1:50
    fOld = f;

    % Update Z
    Z(:) = findMinNN(@funObjZ,Z(:),10,0,X,W);

    % Update W
    W(:) = findMinNN(@funObjW,W(:),10,0,X,Z);

    f = (1/2)*sum(sum((X-Z*W).^2));
    fprintf('Iteration %d, loss = %.5e\n',iter,f);

    if abs(fOld - f) < 1
        break;
    end
end
model.mu = mu;
model.W = W;
model.compress = @compress;
model.expand = @expand;
end
```

The compress function should also be updated:

```
function [Z] = compress(model,X)
[t,d] = size(X);
k = size(model.W,1);
mu = model.mu;
W = model.W;

X = X - repmat(mu,[t 1]);
Z = zeros(t,k);
Z(:) = findMinNN(@funObjZ,Z(:),500,0,X,W);
end
```

The exact basis functions will change because of the random initialization, but they roughly look like this:



Notice that NMF is learning sparse latent factors, which leads to it selecting parts of faces rather than the global factors learned by PCA.

As some students had trouble with Octave's plotting function, it's ok if the solution isn't a perfet 100 by 100 grid.

1.3 Sparse Matrix Factorization

While NMF leads to sparse values of W and Z, disallowing negative values leads to a much worse reconstruction error. In other words, ZW is a much worse approximation of X when using NMF than when using PCA. To explicitly trade off between the reconstruction error and the sparsity, we could instead use L1-regularization of W and Z. Write a function *dimRedSPCA* (for 'sparse' PCA) that uses *dimRedPCA_alternate* as a template but applies L1-regularization when estimate W and when estimating Z . Hand in your code and hand in a plot of the latent factors (Figure 3) obtained with $k = 100$ and $\lambda = \sqrt{nd}$.

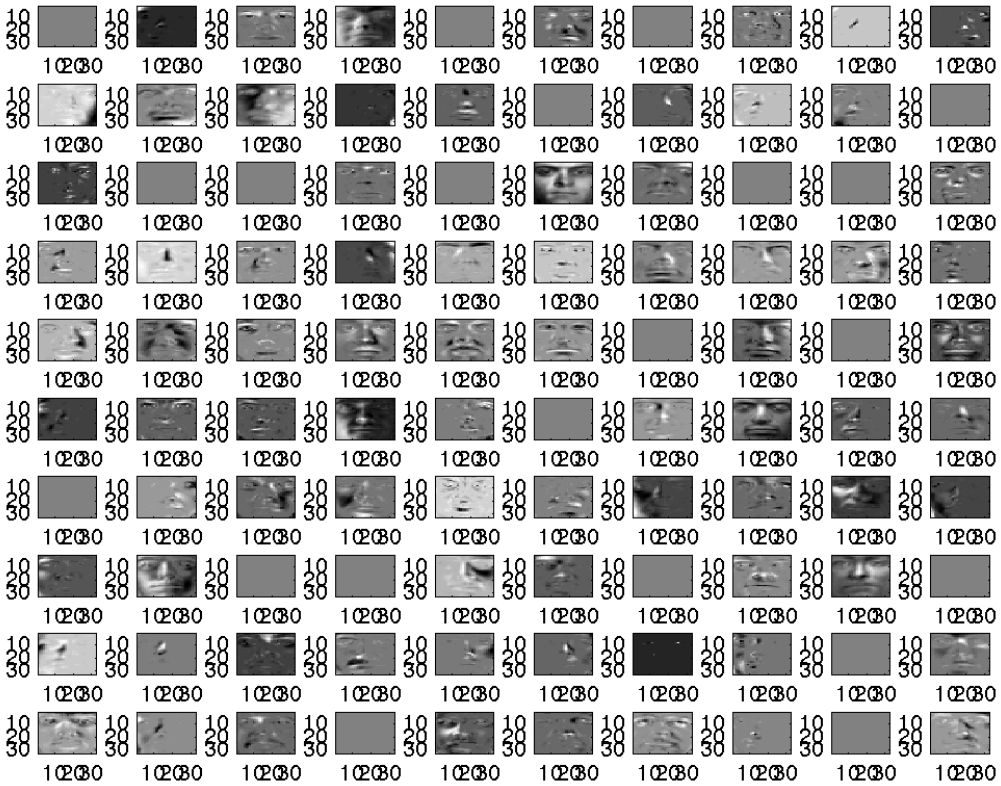
Answer:
The main *dimRedSPCA* should look roughly like this:

```
function [model] = dimRedSC(X,k,lambda)

[n,d] = size(X);

% Subtract mean
mu = mean(X);
X = X - repmat(mu,[n 1]);

% Initialize W and Z
W = randn(k,d);
Z = randn(n,k);

f = (1/2)*sum(sum((X-Z*W).^2));
for iter = 1:50
    fOld = f;

    % Update Z
    Z(:) = findMinL1(@funObjZ,Z(:),lambda,10,0,X,W);

    % Update W
    W(:) = findMinL1(@funObjW,W(:),lambda,10,0,X,Z);

    f = (1/2)*sum(sum((X-Z*W).^2));
    fprintf('Iteration %d, loss = %.5e\n',iter,f);

    if abs(fOld - f) < 1
        break;
    end
end
model.mu = mu;
model.W = W;
model.compress = @compress;
model.expand = @expand;
model.lambda = lambda;
end
```

The compress function should also be updated:

```
function [Z] = compress(model,X)
[t,d] = size(X);
k = size(model.W,1);
mu = model.mu;
W = model.W;

X = X - repmat(mu,[t 1]);
Z = zeros(t,k);
Z(:) = findMinL1(@funObjZ,Z(:),model.lambda,500,0,X,W);
end
```

The exact basis functions will change because of the random initialization, but they roughly look like this:



Notice that SPCA is learning sparse latent factors but obtains a much lower loss than NMF. Also notice that some of the factors are completely sparse, indicating that it is using less than 100 factors.

As some students had trouble with Octave's plotting function, it's ok if the solution isn't a perfet 100 by 100 grid.

2 Recommender Systems

If you run the function *example_movies*, it will load a dataset consisting of movie ratings for different users. The vector y contains the ratings, the first column of X contains the user numbers, and the second column of X contains the movie numbers. The script runs several simple baseline methods, and reports their performance on the validation set.

2.1 Latent-Factor Model

We have no features for the user/movies, we must predict the labels based on other labels (collaborative filtering). One way to improve on these methods is with a latent-factor model. Consider a model of the form

$$y_{um} = b_u + b_m + w_m^T z_u,$$

where the model has four parameters:

- b_u: a bias variable specific to user u.
- b_m: a bias variable specific to movie m.
- W: a matrix whose columns w_j represent latent features for movie m.
- Z: a matrix whose rows z_u represent latent features for user u.

Consider training this based on the squared loss function, which means that our error for a particular user u and movie m is given by

$$f(b_u, b_m, w_m, z_u) = \frac{1}{2}(y_{um} - (b_u + b_m + w_m^T z_u))^2.$$

Using the notation $r_{um} = (y_{um} - (b_u + b_m + w_m^T z_u))$, derive the partial derivative of this expression with respect to (i) b_u, (ii) b_m, (iii) $(w_m)_i$ for a particular element i of w_m, and (iv) $(z_u)_i$ for a particular element i of z_u.

Answer:
We have:

$$\frac{\partial f}{\partial b_u} = -r_{um}.$$

$$\frac{\partial f}{\partial b_m} = -r_{um}.$$

$$\frac{\partial f}{\partial (w_m)_i} = -r_{um} z_{ui}.$$

$$\frac{\partial f}{\partial (z_u)_i} = -r_{um} w_{mi}.$$

So the biases are only affected by the value of the residual, where the latent factor for movie m is influence when by the latent factors of users u that rate the movie (and vice versa).

2.2 Stochastic Gradient

The function *recommendSVD* implements the model from the previous question, and trains it using gradient descent with a constant step size of .0001. Since there are nearly a million ratings, this is quite slow and the script only runs the method for 10 passes through the data (technically, it's our Matlab implementation that is slow as this dataset really isn't that large). In cases like this where we have lots of data but are limited by time, we can often obtain better performance using stochastic gradient methods. Modify *recommendSVD* so it trains using a stochastic gradient method with a constant step-size, doing 10 'passes' of *nRatings* stochastic gradient iterations (which takes the same time as 10 gradient descent iterations). Hand in your modified code, and report the validation error of the method using a larger step-size of .01 and $k = 10$.

Hint: since we are only changing the optimization method, you don't need to change anything outside the 'for iter = 1:maxIter' loop. However, you will now need to compute the gradient with respect to a single *randomly-chosen* training example within the inner 'for' loop, and update the parameters based on the gradient with respect to this single example.

Answer:
The code in the inner loop implementing the stochastic gradient update should look roughly like this:

```
for inner = 1:nRatings
    i = ceil(rand*nRatings);

    % Make prediction for this rating based on current model
    u = X(i,1);
    m = X(i,2);
    yhat = bu(u) + bm(m) + W(:,m)'*Z(u,:)';

    % Compute residual
    r = y(i)-yhat;

    % Update based on gradient
    bu(u) = bu(u) + alpha*r;
    bm(m) = bm(m) + alpha*r;
    W(:,m) = W(:,m) + alpha*r*Z(u,:)';
    Z(u,:) = Z(u,:) + alpha*r*W(:,m)';
end
```

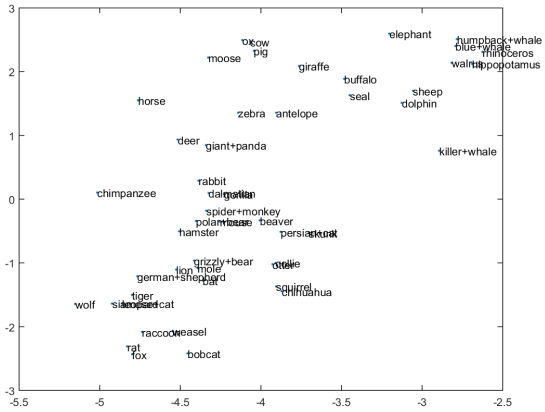
The output will change on each run due to the random initialization and random selection of training examples, but I got a validation error of approximately 0.74.
It's ok if they followed the earlier assignment description and only did n updates on the inner loop, and in this case their error will be fairly similar to the error obtained with the full gradient method.

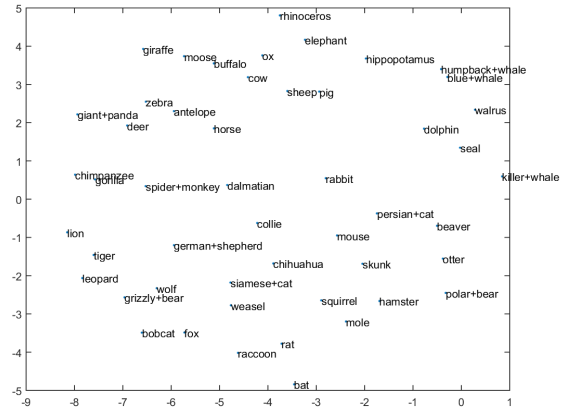
3 Multi-Dimensional Scaling

The function *example_MDS* loads the animals dataset and then shows (i) the raw data, (ii) the data projected onto the first two principal components, and (iii) the result of applying gradient descent to minimize the following multi-dimensional scaling (MDS) objective (starting from the PCA solution):

$$\underset{Z \in \mathbb{R}^{n \times k}}{\operatorname{argmin}} \frac{1}{2} \sum_{i=1}^{n} \sum_{j=i+1}^{n} (\|x_i - x_j\| - \|z_i - z_j\|)^2. \tag{1}$$

As with PCA, it's possible to minimize this objective with a singular value decomposition, but this code uses a gradient method since this generalizes to variations on the objective. The results of applying PCA and MDS are shown below on the left and right.





We see that the crowding effect of PCA makes some non-sensical clusters (e.g., the cluster containing 'spider monkey', 'polar bear', and 'hamster'). MDS is less crowded, but many of the results don't make sense ('persian cat' and 'siamise cat' have 'mouse' and 'chihuahua' in between them, while 'wolf' and 'grizzly bear' are close to each other while 'polar bear' is on the opposite side of the scatterplot).

3.1 Samman Mapping

Make new function *visualizeSammon* that implements gradient descent for MDS Sammon mapping objetive,

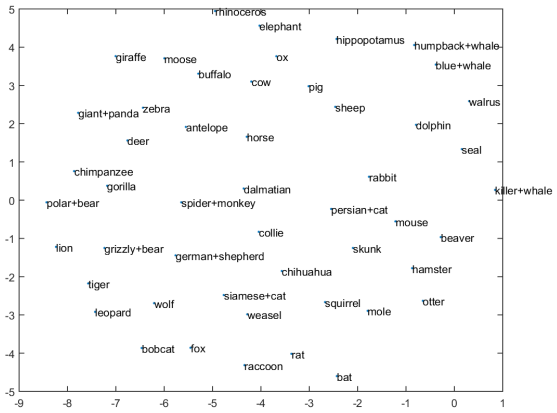
$$\underset{Z\in\mathbb{R}^{n\times k}}{\operatorname{argmin}} \frac{1}{2}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{(\|x_i - x_j\| - \|z_i - z_j\|)^2}{\|x_i - x_j\|}.$$

Hand in your code and the plot of the result.

Answer:
We can implement Sammon's mapping by changing the objective function and gradient:

```
% Objective Function
Dz = norm(Z(i,:)-Z(j,:));
s = D(i,j) - Dz;
f = f + (1/2)*s^2/D(i,j);

% Gradient
df = s/D(i,j);
dgi = (Z(i,:)-Z(j,:))/Dz;
dgj = (Z(j,:)-Z(i,:))/Dz;
g(i,:) = g(i,:) - df*dgi;
g(j,:) = g(j,:) - df*dgj;
```

The result of using this weighting is:



Ultimately, this reweighting doesn't seem to make a big difference, although it does put 'grizzly bear' and 'polar bear' closer together.

3.2 ISOMAP

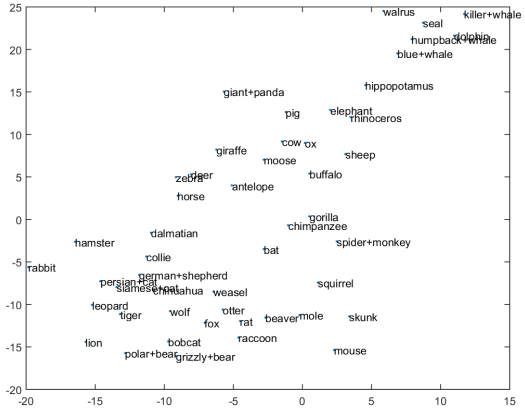
Euclidean distances between very different animals are unlikely to be particularly meaningful. However, since related animals tend to share similar traits we might expect the animals to live on a low-dimensional manifold. This suggests that ISOMAP may give a better visualization. Make a new function *visualizeISOMAP* that computes the approximate geodesic distance (shortest path through a graph where the edges are only between nodes that are k-nearest neighbour) between each pair of points, and then fits a standard MDS model (1) using gradient descent. Hand in your code and the plot of the result when using the 3-nearest neighbours.

Hint: the function *dijskstra* can be used to compute the shortest (weighted) distance between two points in a weighted graph. This function requires an n by n matrix giving the weights on each edge (use 0 as the weight for absent edges). Note that ISOMAP uses an undirected graph, while the k-nearest neighbour graph might be assymetric. One of the usual heuristics to turn this into a undirected graph is to include an edge i to j if i is a KNN of j or if j is a KNN of i. (Another possibility is to include an edge only if i and j are mutually KNNs.)

Answer:
We can implement ISOMAP by changing the distance function:

```
% Compute all distances
D = X.^2*ones(d,n) + ones(n,d)*(X').^2 - 2*X*X';
D = sqrt(abs(D));

% Find neighbours
G = zeros(n);
for i = 1:n
    [minDist,sorted] = sort(D(:,i));
    neighbours = setdiff(sorted(1:K+1),i);
    for j = neighbours
        G(i,j) = D(i,j);
        G(j,i) = D(j,i);
    end
end

D = zeros(n);
for i = 1:n
    for j = i+1:n
        D(i,j) = dijkstra(G,i,j);
    end
end
```

This assumes that we add an edge if i is a KNN of j or vice versa, but other distance functions in the spirit of constructing a KNN graph are also acceptable. The result of using this weighting is:



This distance funciton gets the cats/dogs close together as well as 'grizzly bear' and 'polar bear' (although it still isn't completely intuitive as pandas are not placed with the other bears). Note that the 'dijkstra' function was missing until the Sunday before the deadline, so it's ok if they an alternate means of computing the shortest paths.

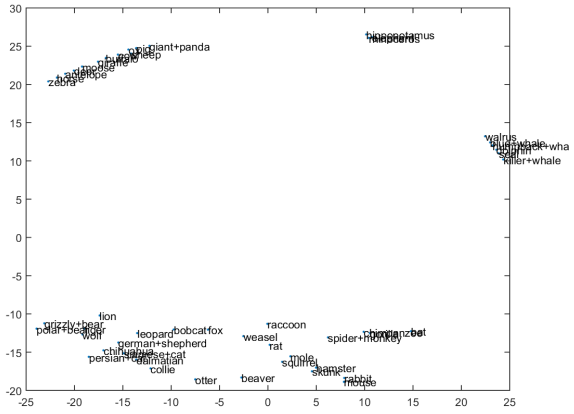
3.3 ISOMAP with Disconnected Graph

An issue with measuring distances on graphs is that the graph may not be connected. For example, if you run your ISOMAP code with 2-nearest neighbours then some of the distances are infinite. One heuristic to address this is to set these infinite distances to the maximum distance in the graph (i.e., the geodesic distance), which will encourage non-connected points to be far apart. Modify your ISOMAP function to implement this heuristic. Hand in your code and the plot of the result when using the 2-nearest neighbours.

Answer:
We can modify the last part of the ISOMAP distance calculation to add this heuristic:

```
D = zeros(n);
maxVal = 0;
for i = 1:n
    for j = i+1:n
        D(i,j) = dijkstra(G,i,j);
        if ~isinf(D(i,j)) && D(i,j) > maxVal
            maxVal = D(i,j);
        end
    end
end
D(isinf(D)) = maxVal;
```

The result when using 2-nearest neighbours and this heuristic is:

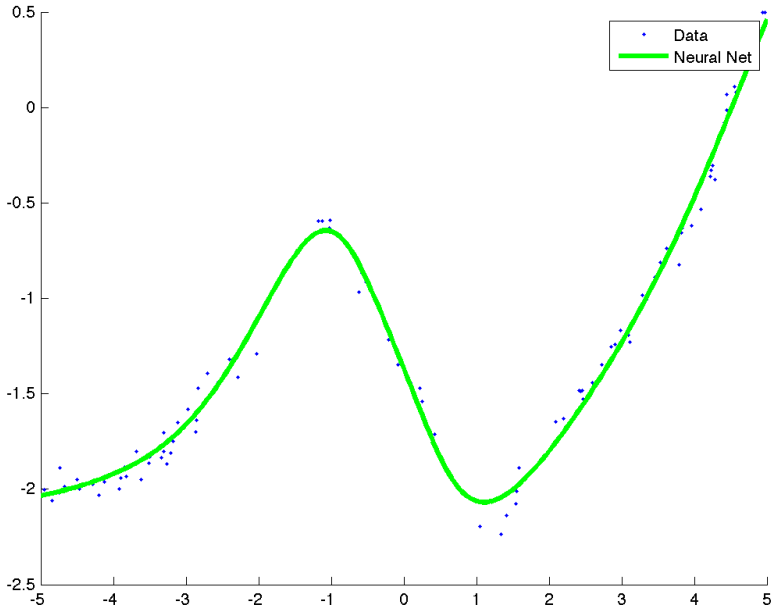


We see that the disconnected graph had the benefit and pushing the connected componenets far away from each other.

4 Visualizing a neural net for 1D regression

The file `example_nnet.m` contains a script to train a basic neural net. It is set up with a 1D example, i.e. where the neural net is used to learn a function mapping $\mathbb{R} \to \mathbb{R}$. When you run the script, you should be able to see training progress as the network begins to fit the data. However, in its current form it doesn't fit the data very well. Try to improve the performance of the method by changing the structure of the network (*nHidden* is a vector giving the number of hidden units in each layer) and the training procedure (e.g., change the sequence of step sizes, add momentum, or use *findMin* from the previous assignment). Hand in your plot after changing the code to have better performance, and list the changes you made.

Answer:
There are a variety of ways to improve the performance, but with a little tuning you should be able to get plot that looks quite nice like (roughly) this:



I found all that was needed was increasing the number of hidden units (to 10) and the step-size (to 0.01).