**Problem 1**

Question 1

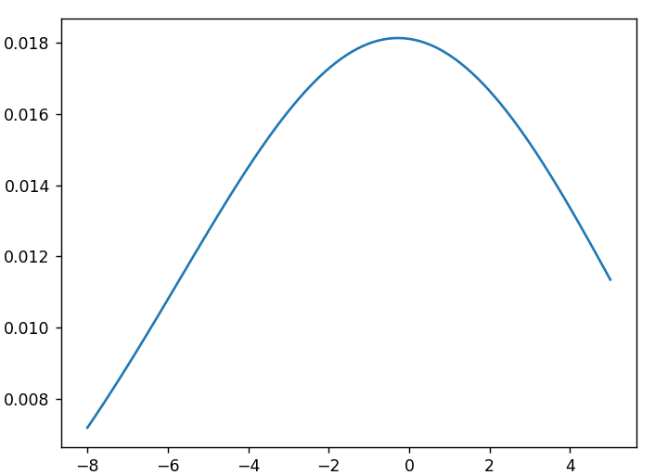
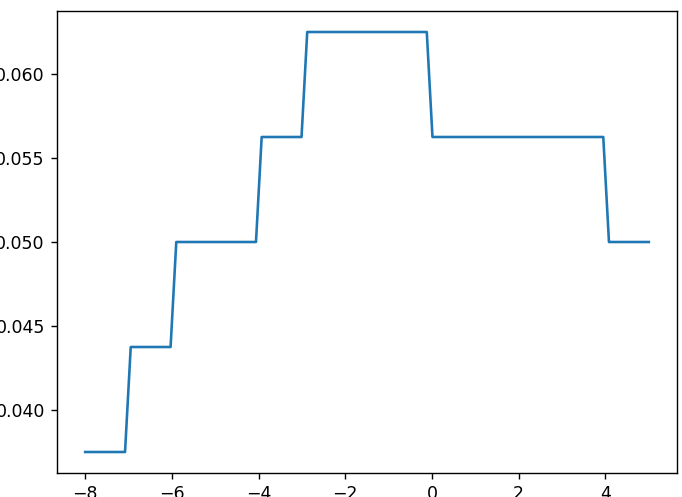
Notice that the risk is:

Substituting 0-1 loss function we get:

We want to minimize the risk, which means to maximize the probability . Notice that:

Notice that when , , and therefore we're simply counting the number of votes of each class, and choosing the one with the maximal votes.

Question 2

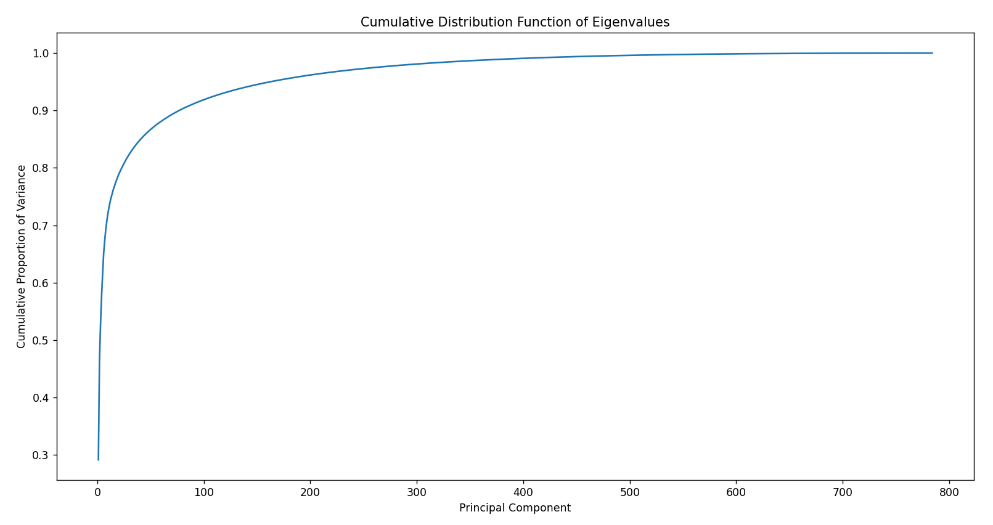
Here's a drawing of the probability estimation:

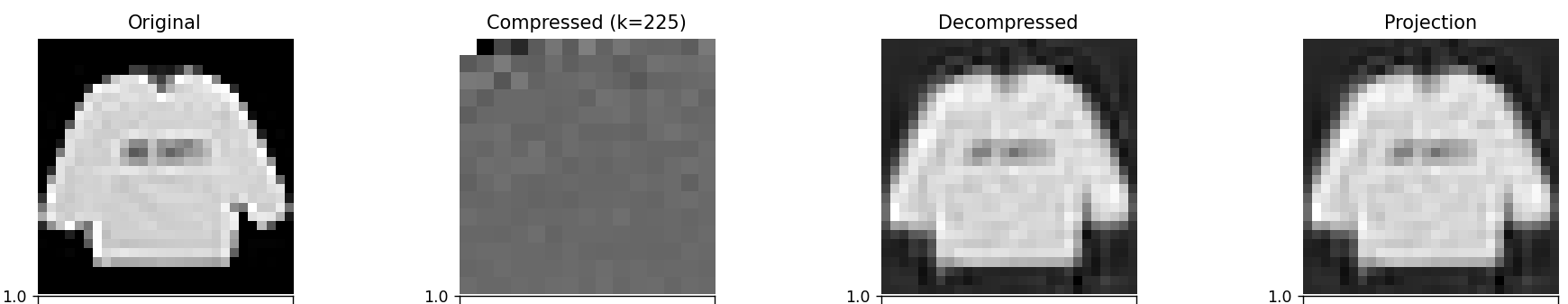
We assume the distribution is gaussian. Here are the MLE estimators:

**Problem 2**

We built a function which loads data given a path, and then returns samples and labels separately as numpy arrays. We plotted a picture using the function plot\_picture to ensure everything is going well.

Next, we defined PCA class which is initialized with the original size of the picture () and the new size (), and the dataset. We found the eigenvalues and eigenvectors of the scatter matrix, and saved the k vectors with the k greatest eigenvalues in a matrix E. We defined a vectorized function that compresses each sample by multiplying it by the matrix E. In addition, we wrote two functions: one that calculates the projection (in dimension d), and one that decompresses images (back to dimension d).

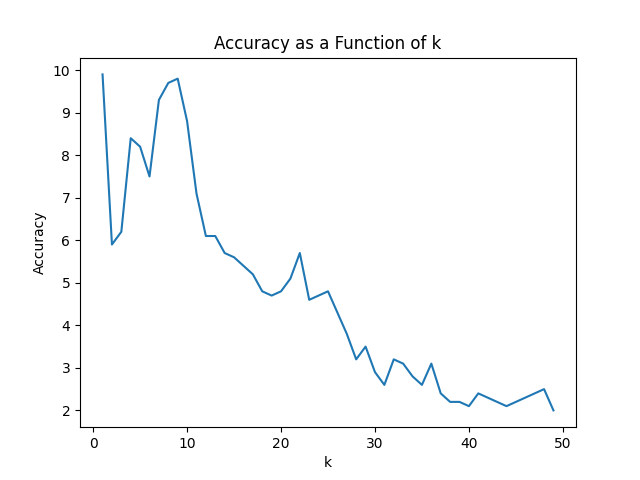
Now, we plot the CDF of eigenvalues:

 As observed in the picture, we can reduce to dimension 225 almost without losing data (0.975 CPV) -great! Now we plot a few pictures

Now we are ready to run kNN. We defined a class, and its constructor, which receives training set, and calculates the square norms of them all (for a futuristic trick). Then, we defined a function "classify". This function gets a sample, and calculates its distance from all points, then does the vote within k closest samples. Notice that calculating the distance for each point would take 550 operations (subtraction and dot product), so instead we decided to find

And so we can use the square norms we preprocessed and do 226 operations per sample instead (dot product 255, subtraction of scalars 1), and this way we have saved half of our time.

We then defined a function 'test' which takes the trained model and a compressed test-set, and calculates its accuracy. To find the optimal k, we ran over a small sample of test set and training set, and got the following graph:

Note: accuracy is very low in this graph because we don't have a lot of samples.

The optimal k is 8, and so we chose it.

The final accuracy is 81.025 %, and running time is 112 seconds (about 2 minutes).

**Problem 3**

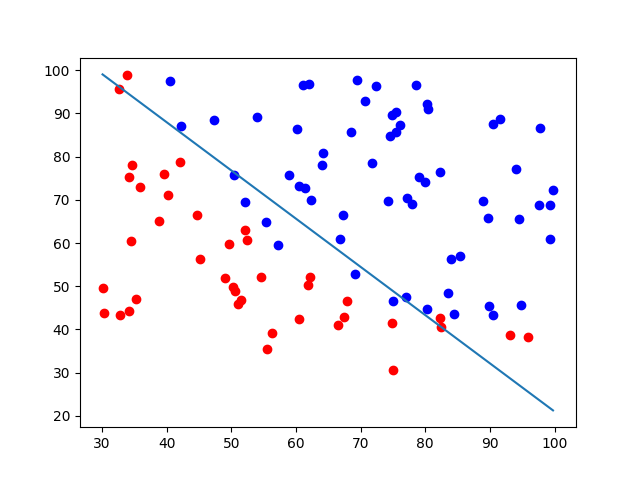
We will be doing logistic regression. For that purpose, we define a class which has 3 attributes: learning rate (etta), convergence constant, and LDF vector (w).

Then, we load the data from the CSV file, and split it into 90-10 train-test sets in the function train. Then, we iterate 28,000 iterations (or until convergance) with gradient descent. This gives us a value for w, which we use to test our model with the function test, which uses a vectorized function of the function classify, then calculates the mean. This whole thing is done within the scope of the function 'evaluate', which tests the model epoch\_num times, and returns the average accuracy rate.

I split the train set into validation and training set (0.9-0.1), and ran over different values. I found accuracy rate as a function of learning rate running until convergance/until finishing 28,000 iterations, and plotted it (it took too much time and I forgot to take a picture so I won't be showing a picture)

The maximal accuracy is 90%, which can be accomplished for different values. The program returned 6.35, and so it is the learning rate in the code.

Average running time is 40 seconds.

Average accuracy rate is 90%. Data is not linearly separable. Best classifier for this dataset is some special kernel with SVM, although it is reasonably almost linearly separable. Here’s the best separating line:

**Problem 4**

We run for each dataset.

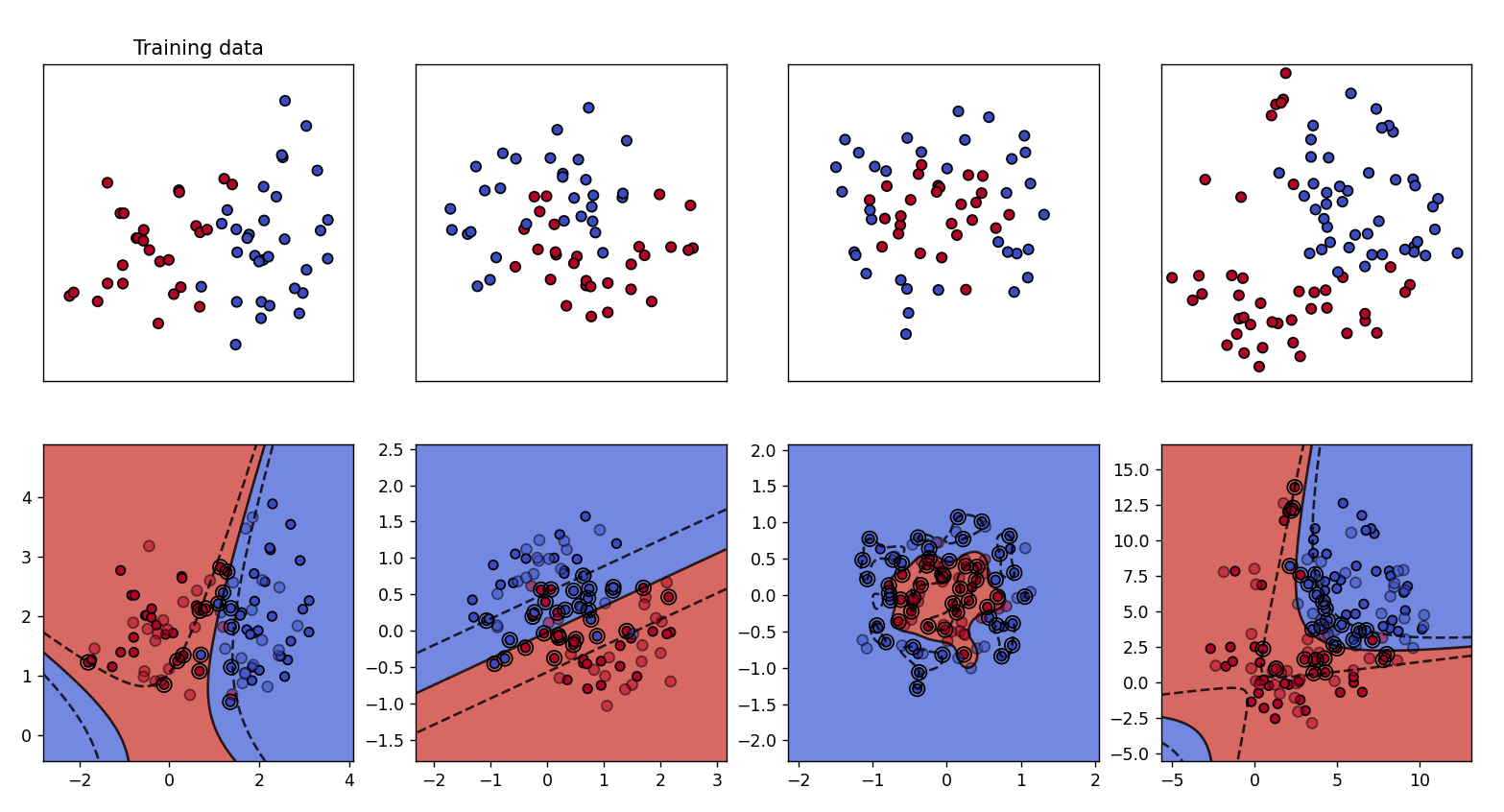
The first one is obviously not RBF(not radical), but could be separated by linear or polynomial kernels with 95% accuracy. The arguments are: .

The second one is almost linearly separable, and so the best kernel is linear with 90% accuracy. The arguments are: .

The third one looks radical, so using RBF is reasonable, and gives 92.5% accuracy, which is good. The arguments are:

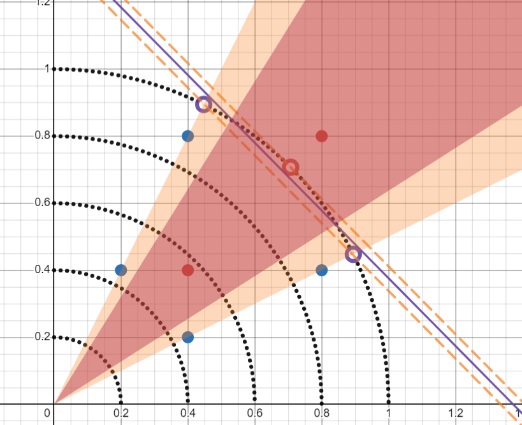
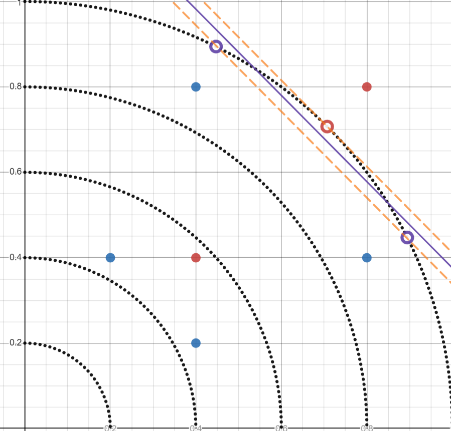
Last but not least, the forth one, obviously not linear, or radical, so by trying polynomial kernel we can get to 96.55% with the parameters: .

Here are all 4 separations aligned in one picture:



**Problem 5**

1. This new kernel maps points to points , and therefore we get the following mapping function:
2. In the following figure you could notice the mapping (empty points)
3. the separating line found by trial (blue line/read area borders), and the margin) shaded area/dashed lines). Notice that our separating line decides red for points between the intersection points, and blue for all the rest, which gives us the angular decision boundary:



1. Now, our data space is:

With a simple substitution we get that blue is 1, red is -1. Now we substitute in:

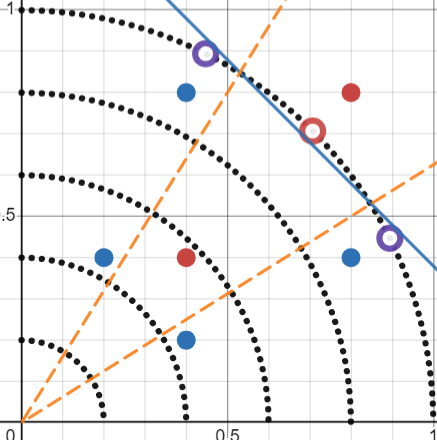
We obtain:

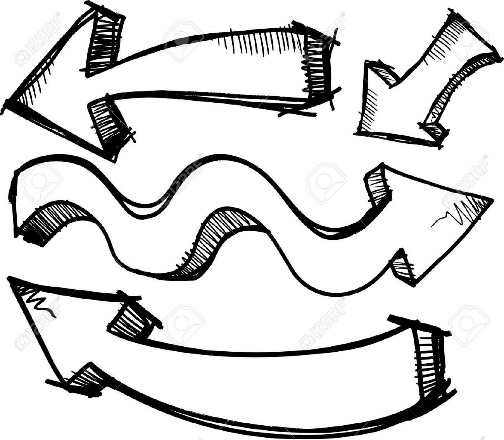
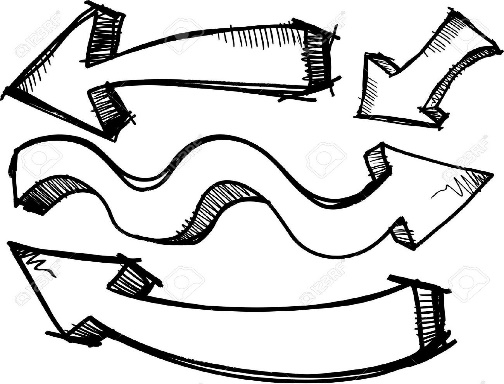
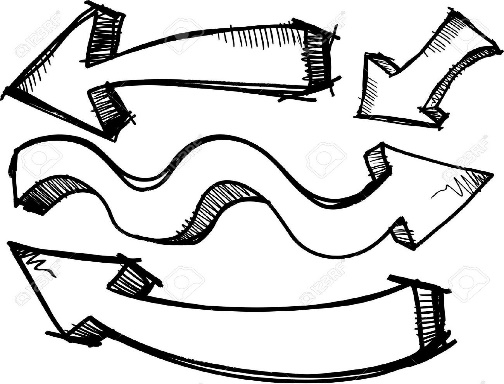
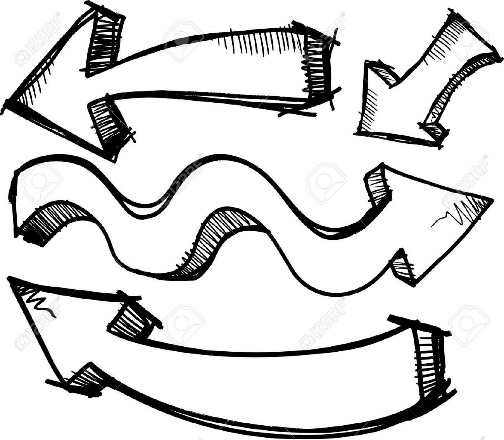
Which is solved by:

1. The non-linear hyperplane is given by:

Which unsurprisingly simplifies to:

Which is the dashed line in the following drawing:

 For all equations used, click on the fish.



THE FISH!!