MAD Exam 2019

MAD 2018

Department of Computer Science
University of Copenhagen

Exam number: 46

Version 1
Due: January 20th, 23:59

Contents

1. Exercise

(a)

To solve this, we need to realise that a pdf always has the following property: It's integral over the entire function is equal to 1. Since we know that

$$f(x) = 0, \text{ if } 0 > x, 5 < x$$

We can compute the definite integral between 0 and5, $\int_0^5 f(x)dx$ to find the integral. This means that we have to solve the following equation:

$$\int_0^5 c(5-x)dx = 1$$

First we will extend it to make it easier to work with

$$\int_0^5 5c - xcdx = 1$$

We know that definite integrals are solved as follows.

$$\int_a^b f(x)dx = F(b) - F(a)$$

First we find $F(x)$:

$$\int 5c - xcdx = 5xc - \frac{x^2c}{2}$$

Now we solve $F(5) - F(0)$:

$$(5 * 5c - \frac{5^2c}{2}) - (5 * 0c - \frac{0^2c}{2}) = 1$$

This can be simplified to:

$$12.5c = 1$$

We can then realise that for this to be true, $c = \frac{2}{25}$

(b)

First we will find the pdf of V. We know that to find the pdf, we have to compute the derivative of the cdf. To do this on a piecewise function, we compute the derivative of each piece:

$$f(v) = \frac{d}{dv}F(v) = \begin{cases} \frac{d}{dv}0 & if \quad v < 0 \\ \frac{d}{dv}v^5 & if \quad 0 \leq v \leq 1 \\ \frac{d}{dv}1 & if \quad v > 1 \end{cases} = \begin{cases} 0 & if \quad v < 0 \\ 5v^4 & if \quad 0 \leq v \leq 1 \\ 0 & if \quad v > 1 \end{cases}$$

To find the expectation, $\mathbb{E}V$ we have to use the following formula, since V is a continuous random variable. We know that $\int f(v)dv = F(v)$:

$$\mathbb{E}V = \int vf(v) = \begin{cases} \int 0vdv & if \quad v < 0 \\ \int 5v^5 dv & if \quad 0 \leq v \leq 1 \\ \int 0vdv & if \quad v > 1 \end{cases} = \begin{cases} 0 & if \quad v < 0 \\ \frac{5v^6}{6} & if \quad 0 \leq v \leq 1 \\ 0 & if \quad v > 1 \end{cases}$$

(c)

To prove this, we first have to set an $\epsilon\varepsilon > 0$. We know the following:

$$\mathbb{P}(|Y_n - Y| \geq \varepsilon) \longrightarrow 0$$

$$\mathbb{P}(|Z_n - Z| \geq \varepsilon) \longrightarrow 0$$

And we want to prove that:

$$\mathbb{P}(|(Y_n - Y) + (Z_n - Z)| \geq \varepsilon) \longrightarrow 0$$

We start off by writing the following, which we know is true, since the value of ε is arbitrarily small:

$$\mathbb{P}\Big(|Y_n - Y| \geq \frac{\varepsilon}{2}\Big) \longrightarrow 0$$

$$\mathbb{P}\Big(|Z_n - Z| \geq \frac{\varepsilon}{2}\Big) \longrightarrow 0$$

We can now write the following:

$$\mathbb{P}\Big(|Y_n - Y| \geq \frac{\varepsilon}{2}\Big) + \mathbb{P}\Big(|Z_n - Z| \geq \frac{\varepsilon}{2}\Big) \longrightarrow 0$$

We can then write the following, and also see that it still approaches 0:

$$\mathbb{P}\Big(|Y_n - Y| \geq \frac{\varepsilon}{2}\Big) + \mathbb{P}\Big(|Z_n - Z| \geq \frac{\varepsilon}{2}\Big) \geq \mathbb{P}\Big(|Z_n - Z| \geq \frac{\varepsilon}{2} \vee |Y_n - Y| \geq \frac{\varepsilon}{2}\Big)$$

$$\mathbb{P}\Big(|Z_n - Z| \geq \frac{\varepsilon}{2} \vee |Y_n - Y| \geq \frac{\varepsilon}{2}\Big) \longrightarrow 0$$

We can see that this probability *must* be greater than $\mathbb{P}(|(Y_n - Y) + (Z_n - Z)| \geq \varepsilon)$, since for this to be true at least one of the events in the previous probability have to happen. Because of this, we can write the following:

$$\mathbb{P}(|(Y_n - Y) + (Z_n - Z)| \geq \varepsilon) \leq \mathbb{P}\Big(|Z_n - Z| \geq \frac{\varepsilon}{2} \vee |Y_n - Y| \geq \frac{\varepsilon}{2}\Big)$$

We can now say that since

$$\mathbb{P}\Big(|Z_n - Z| \geq \frac{\varepsilon}{2} \vee |Y_n - Y| \geq \frac{\varepsilon}{2}\Big) \longrightarrow 0$$

and

$$\mathbb{P}(|(Y_n - Y) + (Z_n - Z)| \geq \varepsilon) \leq \mathbb{P}\Big(|Z_n - Z| \geq \frac{\varepsilon}{2} \vee |Y_n - Y| \geq \frac{\varepsilon}{2}\Big)$$

The following must be true:

$$\mathbb{P}(|(Y_n - Y) + (Z_n - Z)| \geq \varepsilon) \longrightarrow 0$$

This is exactly what we wanted to prove, since:

$$Y_n + Z_n \xrightarrow{\mathbb{P}} Y + Z = \mathbb{P}(|(Y_n - Y) + (Z_n - Z)| \geq \varepsilon) \longrightarrow 0$$

2. Exercise

(a)

First, we will find the likelihood, which is the same as the *joint pdf*. It is defined as thus:

$$f_\theta^{joint}(x_1, ..., x_n) = \prod_{i=1}^{n} f_\theta(x_n)$$

We can then figure out that the likelihood is:

$$f_\beta^{joint}(x_1, ..., x_n) = \begin{cases} \prod_{i=1}^{n} f_\beta(x_n) & \text{if } 0 \leq x \leq \beta \\ 0 & otherwise. \end{cases}$$

$$f_\beta^{joint}(x_1, ..., x_n) = \begin{cases} \prod_{i=1}^{n} \frac{2}{\beta^2} \cdot (\beta - x_n) & \text{if } 0 \leq x \leq \beta \\ 0 & otherwise. \end{cases}$$

We then have to figure out the maximum likelihood estimator. This is done using the following formula:

$$\hat{\theta}_n^{ML}(x_1, ..., x_n) = \underset{\theta}{argmax} \prod_{i=1}^{n} f_\theta(x_n)$$

We insert the previous formula with the given values for X:

$$\hat{\beta}_n^{ML}(x_1, ..., x_n) = \underset{\beta}{argmax} \prod_{i=1}^{n} f_\beta(x_n) = \underset{\beta}{argmax} \left(\frac{2}{\beta^2} \cdot (\beta - 3) \cdot \frac{2}{\beta^2} \cdot (\beta - 4) \right)$$

First we rewrite it to make our lives easier:

$$\underset{\beta}{argmax} \left(\frac{4(\beta - 4)(\beta - 3)}{\beta^4} \right) = \underset{\beta}{argmax} \left(\frac{4}{\beta^2} - \frac{28}{\beta^3} + \frac{48}{\beta^4} \right)$$

Now we have to differentiate it to find the local extrema:

$$\frac{d}{d\beta}\frac{4}{\beta^2} - \frac{28}{\beta^3} + \frac{48}{\beta^4} = \frac{-8\beta^2 + 84\beta - 192}{\beta^5}$$

Then we have to set this to 0 and solve:

$$\frac{-8\beta^2 + 84\beta - 192}{\beta^5} = 0$$

$$-8\beta^2 + 84\beta - 192 = 0$$

And find that there are two solutions:

$$\beta_1 = \frac{21}{4} - \frac{\sqrt{57}}{4}$$

$$\beta_2 = \frac{21}{4} + \frac{\sqrt{57}}{4}$$

Then, to find whether they are maxima or minima, we compute the derivative of this:

$$f''(\beta) = -16\beta + 84$$

And solve this with $\beta = \beta_1$ and $\beta = \beta_2$ respectively:

$$-16\left(\frac{21}{4} - \frac{\sqrt{57}}{4}\right) + 84 = 4\sqrt{57}$$

$$-16\left(\frac{21}{4} + \frac{\sqrt{57}}{4}\right) + 84 = -4\sqrt{57}$$

Since $f''(\beta_2) < 0$, we find that the $\underset{\beta}{argmax}$ is at:

$$\beta = \frac{21}{4} + \frac{\sqrt{57}}{4}$$

We find that the approximate value of this is ≈ 7.137, and verify that it satisfies $x \leq \beta$

$$x_1 = 3 < x_2 = 4 \leq \beta \approx 7.137$$

(b)

To perform this statistical test, we have to go through the six steps:

- A model:
  The model is given as $X \sim Bin(n, \theta)$, with $n = 20$
- A hypothesis:
  A hypothesis is given as well: $H_0 : \theta = 0.5, H_1 : \theta \neq 0.5$

- A test statistic and distribution:
  We use a binomial test to find the probability that a $\theta = 0.5$ is correct when $n = 20$ and $X = 13$. What this means is that we can simply say that our test statistic is:
  $$T = X$$
  This is because the rejection region we will calculate later only needs to know our X value.

- A significance level:
  The significance level is also given as 0.05

- A rejection region:
  A rejection region is given as:
  $$\mathcal{R} = \{0, ..., \alpha\} \cup \{20 - \alpha, ..., 20\}$$
  So we have to find an α. We do this by solving the following function, and find that it gives us $\alpha = 5$:
  $$\underset{a}{argmax}\left(\sum_{k=0}^{a}\left(\binom{n}{k}\theta^k(1-\theta)^{n-k}\right) \leq 0.025\right) = 5$$
  We thus find that $\mathcal{R} = \{0, ..., 5\} \cup \{15, ..., 20\}$

- Compute test statistic:
  We insert the values into the test statistic to see whether the given data is within the rejection region:
  $$T = X = 13$$
  We can see that this is not within the rejection region:
  $$T = 13 \notin \mathcal{R} = \{0, ..., 5\} \cup \{15, ..., 20\}$$
  Thus, we can attribute the higher than average number of correct guesses to chance, not the predictions.

3. Exercise

(a)

This exercise has been completed in the attached *K_Means.ipynb* Jupyter Notebook. The most relevant bits of source code can be seen here as well:

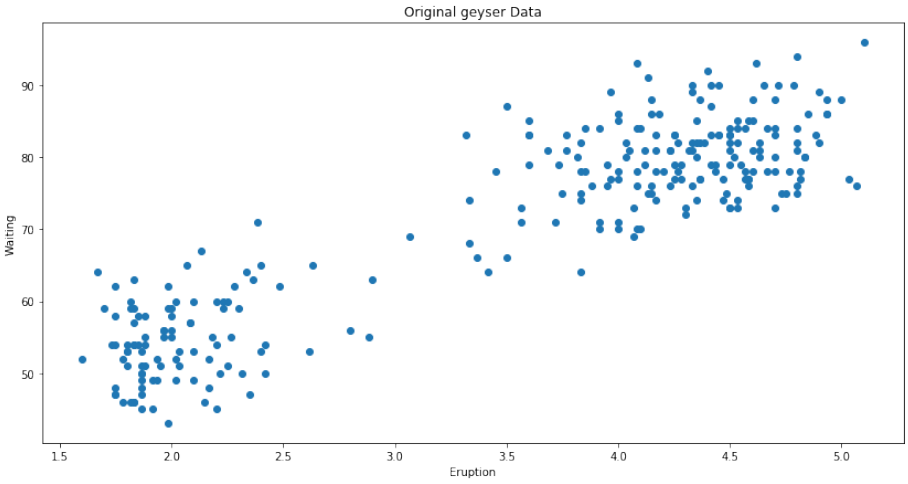
```
data = np.loadtxt("old_faithful.csv", delimiter=',', skiprows=1)

meanData = KMeans(n_clusters=2, max_iter=30, seed=0)
meanData.fit(data)

for i in range(data.shape[0]):
```

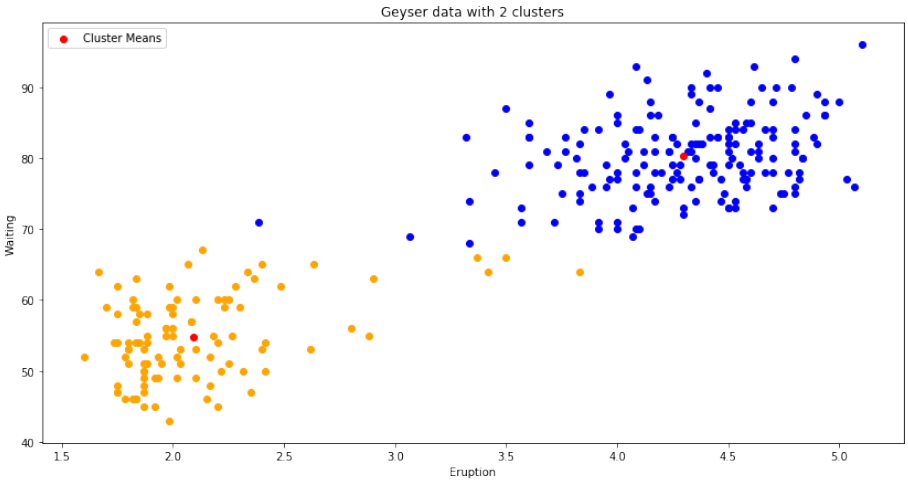
```
if (meanData.cluster_assignments[i] == 0):
    plt.scatter(data[i,0],data[i,1], color='blue')
else:
    plt.scatter(data[i,0],data[i,1], color='orange')
```

The 2 cluster means, calculated using the Jupyter Notebook, look as follows:

$$[4.29793023, 80.28488372]$$
$$[2.09433, 54.75]$$

And the resulting images, with, with each cluster in their own colour, and the cluster means plotted as singular red dots:





(b)

This exercise has been completed in the attached *K_Means.ipynb* Jupyter Notebook. The most relevant bits of source code can be seen here as well, though all

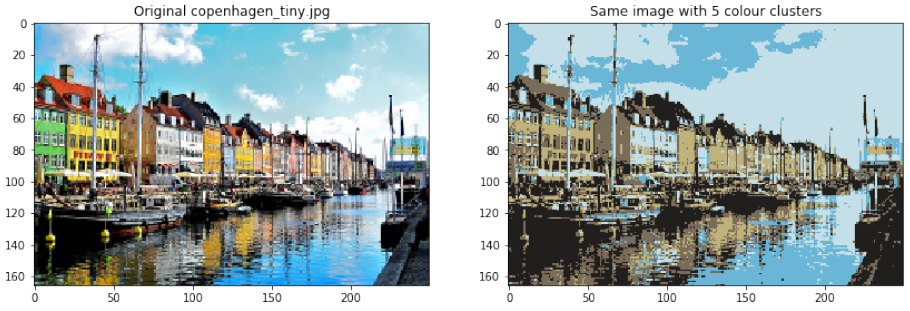
comments are left out. The final version is also different, as the function has to accommodate larger images. The function remains unchanged for this image though:

```
def makeNewImg(image, n_clusters, max_iter, seed):
    imgx   = image.shape[0]
    imgy   = image.shape[1]
    imgRGB = image.shape[2]
    model  = KMeans(n_clusters, max_iter, seed)
    image  = image.reshape((imgx*imgy),imgRGB)
    model.fit(image)

    retval = np.zeros(((imgx*imgy),imgRGB))

    for i in range(n_clusters):
        for j in range(imgRGB):
            model.cluster_means[i,j] =
                model.cluster_means[i,j]/255

    for i in range(retval.shape[0]):
        retval[i] =
            model.cluster_means[model.cluster_assignments[i]]

    retval = retval.reshape((imgx,imgy,imgRGB))
    return retval, model.cluster_means
```

The 5 cluster means, calculated using the Jupyter Notebook, look as follows. It should be noted that they are all RGB values, but Python wants RGB values in one of two forms; floats between 0 and 1, or integers between 0 and 255. Here they are floats between 0 and 1:

$$[0.41773344, 0.71691352, 0.84028094]$$
$$[0.13106363, 0.12181295, 0.11628755]$$
$$[0.47558986, 0.42848816, 0.35511877]$$
$$[0.76865056, 0.87663444, 0.90978927]$$
$$[0.75330927, 0.69978601, 0.48412478]$$

The assigned cluster for each pixel has been calculated, and the new image has been plotted:



(c)

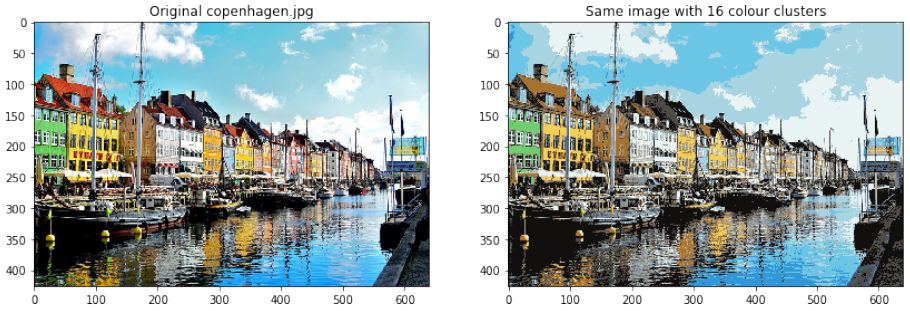
This exercise has been completed in the attached *K_Means.ipynb* Jupyter Notebook. The most relevant bits of source code can be seen here as well, though all comments are left out. It should be noted that the code for this is the exact same as in the previous subexercise, except the "model.fit(image)" line has been exchanged for the following. The full source code is in the notebook:

```
if (imgx*imgy > 50000):
        modLen = 5000

        temp = np.arange(0,(imgx*imgy),1).reshape((imgx*imgy))
        smallImgInd = np.random.choice(temp, modLen, False)

        smallImg = image[smallImgInd]

        model.fit(smallImg)
        model.cluster_assignments =
            model.assign_to_clusters(image, model.cluster_means)
    else:
        model.fit(image)
```

The 16 cluster means, calculated using the Jupyter Notebook, look as follows. It should be noted that the cluster means have been calculated by choosing 5000 random pixels in the image, and fitting the model to those instead of the entire image. Doing it this way should only have a minimal effect on the result, if 5000 pixels are enough to find a representative of the image. It should also be noted that they are all RGB values, but Python wants RGB values in one of two forms; floats between 0 and 1, or integers between 0 and 255. Here they are floats between 0 and 1:

[0.49441064, 0.46930358, 0.42724814]
[0.31073644, 0.29808085, 0.27156238]
[0.66100118, 0.82807700, 0.88582735]
[0.44086687, 0.80361197, 0.43168215]
[0.76419118, 0.74735294, 0.69117647]
[0.07393128, 0.06195302, 0.06012036]
[0.65701033, 0.44427577, 0.18943707]
[0.58076367, 0.58930857, 0.57473684]
[0.94262126, 0.81946042, 0.34577621]
[0.25930767, 0.37516340, 0.48351489]
[0.19096334, 0.59283887, 0.85473146]
[0.59541250, 0.66392897, 0.70810211]
[0.42336683, 0.77392124, 0.89866355]
[0.45891803, 0.51344200, 0.56130404]
[0.71649763, 0.62609872, 0.47396890]
[0.92123446, 0.95420905, 0.95794388]

The assigned cluster for each pixel has been calculated, and the new image has been plotted:



4. Exercise

(a)

The function h_σ is what determines how heavily the regression function should weight each point of data. This means that when it is very large, each point in the given data means less to the end outcome. Likewise, when it is small, each point changes the resulting model significantly more. What this means is that the same effects happen for small sigma and large sigma values, respectively. The inverse is then also true for $||\bar{x} - x_n||^2$.

(b)

This exercise has been completed in the attached *Regression.ipynb* Jupyter Notebook, and *linreg.py* python file, which is imported by the Notebook. The most relevant bits of source code can be seen here as well, though all comments are left out. The *linreg.py* file is a modified version of the given file of the same name. Code not included here is simply unmodified. The full source code is in the notebook, along with comments:

```
def hsig(xbar,xn,sigma):
    return np.exp(-(np.linalg.norm(xbar-xn)**2)/(2*sigma**2))

def nonlinfit(self, X, t, xbar, sigma):
        X = numpy.array(X).reshape((len(X), -1))
        t = numpy.array(t).reshape((len(t), 1))
        ones = numpy.ones((X.shape[0], 1))
        X = numpy.concatenate((ones, X), axis=1)
        diag = self.lam * len(X) * numpy.identity(X.shape[1])
        h = np.zeros((X.shape[0],X.shape[0]))
        for j in range(X.shape[0]):
            h[j,j] = hsig(X[j], xbar, sigma)
        a = X.T @ h @ X + diag
        b = X.T @ h @ t
        self.w = numpy.linalg.solve(a,b)
```

To find $\hat{w}(\bar{x})$ we first have to derive the function given. First off, we will write it as a function of matrices and vectors:

$$\underset{w}{argmin}\frac{1}{N}\sum_{n=1}^{N} h_\sigma(\bar{x}, x_n)(w^T x_n - t_n)^2 = \nabla_w \frac{1}{N} h_\sigma\Big((Xw-t)^T(Xw-t)\Big)$$

$$\nabla_w \frac{1}{N} h_\sigma\Big((Xw^T - t^T)(Xw-t)\Big)$$

$$\nabla_w \frac{1}{N} h_\sigma\Big((Xw)^T Xw - t^T Xw - (Xw)^T t + t^T t\Big)$$

$$\nabla_w \frac{1}{N} h_\sigma\Big(w^T X^T Xw - 2w^T X^T t + t^T t\Big)$$

$$\nabla_w \Big(\frac{1}{N} w^T X^T h_\sigma Xw - \frac{2}{N} w^T X^T h_\sigma t + \frac{1}{N} t^T h_\sigma t\Big)$$

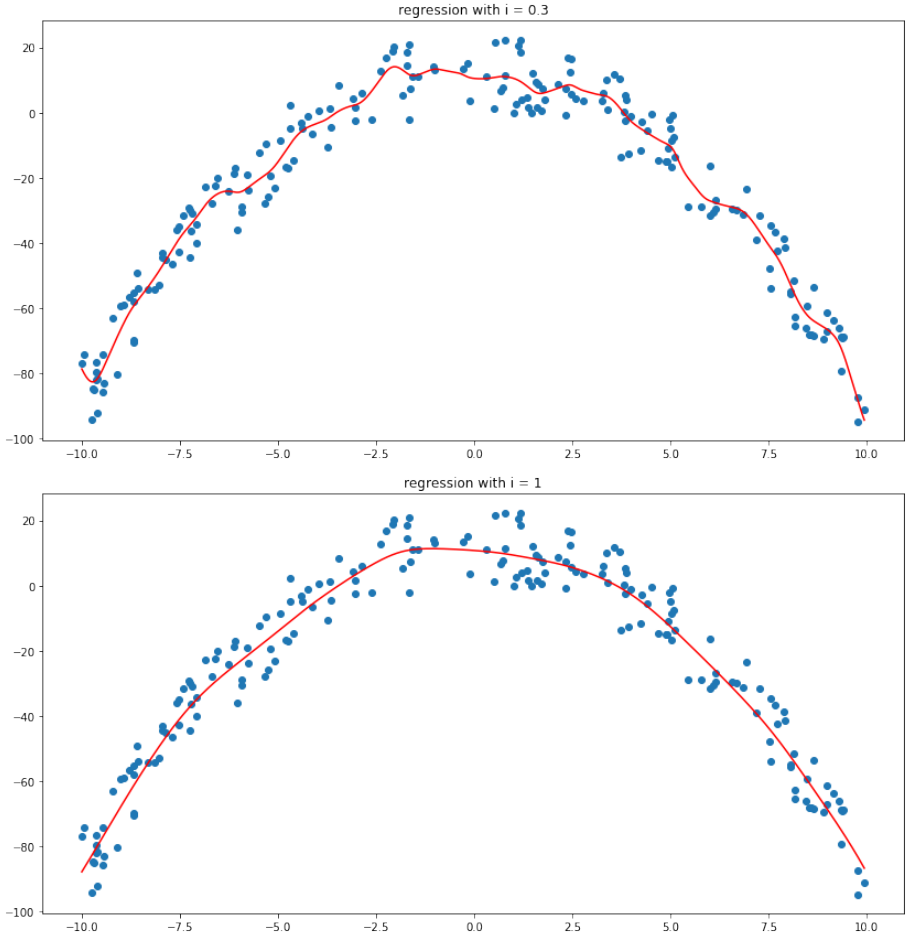
$$\frac{2}{N} X^T h_\sigma Xw - \frac{2}{N} X^T h_\sigma t$$

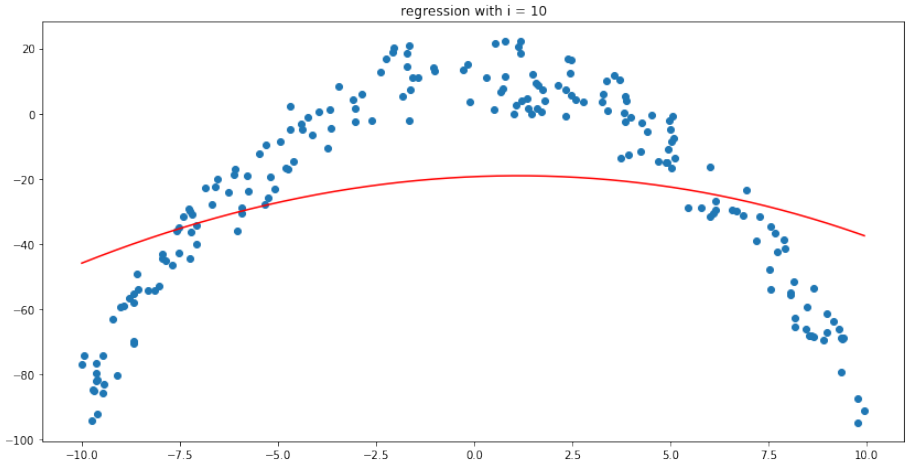
$$X^T h_\sigma Xw - X^T h_\sigma t = 0$$

$$X^T h_\sigma Xw = X^T h_\sigma t$$

This expression is what we solve for in the function, as a and b respectively.
It should be noted that the numerical problems described in the assignment already happen when $\sigma = 0.1$ on my pc. We instead run it with $\sigma = 0.3$ to give

a similar effect.
We see that the outcome we predicted in 4*a* was true. Higher values of sigma leads to each point having a smaller effect on the overall look of the model, and lower values do the opposite. We see that when we run the function with low values of sigma, we get significant overfitting that is not useful in a model. When we go up to higher values we see that the resulting graph flattens out. Eventually, when we try a $\sigma = 100$ we see that it resembles the linear regression shown earlier in the given notebook. Plots showing graphs for high sigma values are in the attached Notebook.





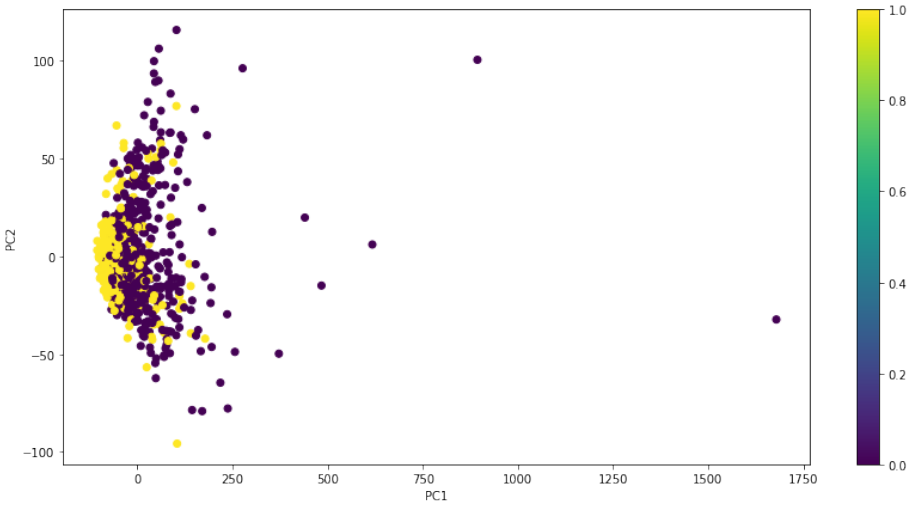
5. Exercise

(a)

This exercise has been completed in the attached *PCA and classification.ipynb* Jupyter Notebook. The most relevant bits of source code can be seen here as well, though all comments are left out. It should be noted that the pca function is based on the one from lecture slides, but it has been modified. The full source code is in the notebook, along with comments:

```
def pca(data):
    d, N = data.shape

    center = np.mean(data, 1)
    centers = np.matlib.repmat(center, N, 1)
    data_cent = data - np.transpose(centers)

    Sigma = np.cov(data_cent)
    evals, evecs = np.linalg.eigh(Sigma)

    return np.flip(evals,0), np.flip(evecs, 1), data_cent

eVals, eVecs, data_cent = pca(trainset.T)

PC1 = eVecs[:,0]
PC2 = eVecs[:,1]

PC1projs = np.matmul(data_cent.T,PC1)
PC2projs = np.matmul(data_cent.T,PC2)
```

The resulting plot can be seen here:



We can see here that the two different labels can be quite difficult to differentiate, as they are very close together based on these two pricipal components. We do however, see that the label '0' has slightly higher variance in its clustering, especially on the PC1 axis.

(b)

This exercise has been completed in the attached *PCA and classification.ipynb* Jupyter Notebook. The most relevant bits of source code can be seen here as well, though all comments are left out. It should be noted that the distanceMat function is based on the one from lecture slides, but it has been modified. The full source code is in the notebook, along with comments:

```
def distanceMat(train, test):
    distMat = np.zeros((train.shape[0], test.shape[0]))

    for i in range(train.shape[0]):
        for j in range(test.shape[0]):
            distMat[i,j] = np.linalg.norm(train[i,:]-test[j,:])

    return distMat

def knn(train, test, trainlab, testlab, k):
    distMat = distanceMat(test,train)
    sortedInd = np.argsort(distMat, axis=1)
    sortedPreds = trainlab[sortedInd]

    predictions =
        np.floor(np.sign(np.sum((sortedPreds[:,range(k)]-0.5)*2,1))/2 + 0.5)
    accuracy =
        1 - (np.sum(np.abs(predictions-testlab))/len(testlab))
```

```
    return predictions, accuracy
```

(c)

The results of first testing the training set against itself, then testing against the test set gave the following results:

- Accuracy when testing against the training set with k = 1: 100.0%
- Accuracy when testing against the training set with k = 3: 83.5714%
- Accuracy when testing against the training set with k = 5: 80.7142%
- Accuracy when testing against the test set with k = 1: 69.8744%
- Accuracy when testing against the test set with k = 3: 73.6401%
- Accuracy when testing against the test set with k = 5: 75.1046%

We realise that when testing on the same set we trained on, we will always get 100% when $k = 1$, because the nearest neighbour will always be itself. We see that when looking at higher values of k (displayed in the attached Jupyter Notebook), the accuracy decreases slightly as we look at more neighbours. When testing against the test set, we see that the accuracy starts low, then increases quickly with larger values of k to hover around 75%.

(d)

This exercise has been completed in the attached *PCA and classification.ipynb* Jupyter Notebook. The most relevant bits of source code can be seen here as well, though all comments are left out. The full source code is in the notebook, along with comments:

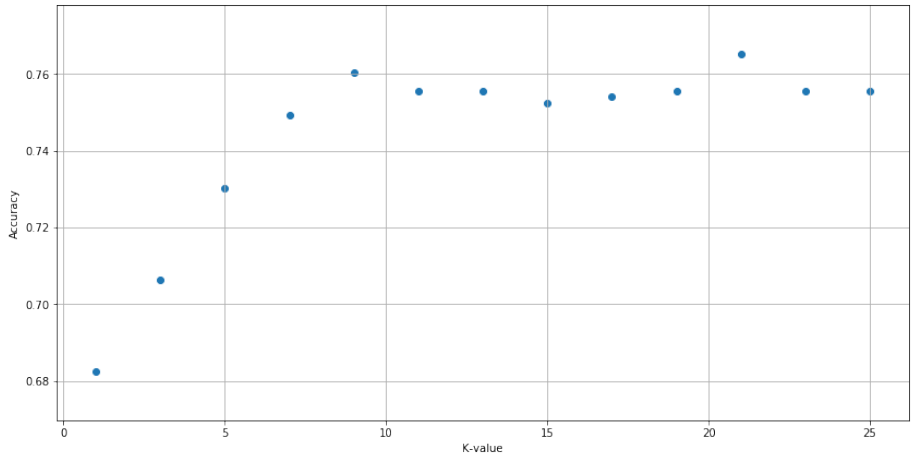
```
def kfold(train, labels, x):
    retTrain = numpy.zeros(train.shape)
    retLabs = numpy.zeros(labels.shape)
    perm = numpy.random.permutation(train.shape[0])

    for oldInd, newInd in enumerate(perm):
        retTrain[newInd] = train[oldInd]
        retLabs[newInd] = labels[oldInd]

    retTrain =
        retTrain.reshape(x,
                         int(np.floor(train.shape[0]/x)),
                         train.shape[1])
    retLabs =
        retLabs.reshape(x,int(np.floor(labels.shape[0]/x)))
    return retTrain, retLabs

def crossVal(train, trainlab, x, k):
```

```
    foldedArrs, foldedLabs = kfold(train, trainlab, x)

    accs = np.zeros(int(np.ceil(k/2)))
    nums = np.arange(1,k+1,2)

    for i in range(1,k+1,2):
        tempAccs = np.zeros(x-1)

        for j in range(x-1):
            tempTrain =
                np.zeros((foldedArrs.shape[0]-1,foldedArrs.shape[1],foldedArrs.shape[2]))
            tempTrainLab =
                np.zeros((foldedLabs.shape[0]-1,foldedLabs.shape[1]))

            tempTest  = np.zeros((1,foldedArrs.shape[1],foldedArrs.shape[2]))
            tempTestLab  = np.zeros((1,foldedLabs.shape[1]))

            for l in range(x):
                if (l<j):
                    tempTrain[l] = foldedArrs[l]
                    tempTrainLab[l] = foldedLabs[l]
                elif (l>j):
                    tempTrain[l-1] = foldedArrs[l]
                    tempTrainLab[l-1] = foldedLabs[l]
                else:
                    tempTest =
                        foldedArrs[l].reshape(1,foldedArrs.shape[1],foldedArrs.shape[2])
                    tempTestLab = foldedLabs[l].reshape(1,foldedLabs.shape[1])

            tempTrain =
                tempTrain.reshape((tempTrain.shape[0]*tempTrain.shape[1]),
                                  tempTrain.shape[2])
            tempTest =
                tempTest.reshape((tempTest.shape[0]*tempTest.shape[1]),
                                 tempTest.shape[2])
            tempTrainLab =
                tempTrainLab.reshape((tempTrainLab.shape[0]*tempTrainLab.shape[1]))
            tempTestLab =
                tempTestLab.reshape((tempTestLab.shape[0]*tempTestLab.shape[1]))
            tempAccs[j] =
                knn(tempTrain,tempTest,tempTrainLab,tempTestLab, i)[1]

        index = np.argmax(accs)
        accs[int(np.ceil(i/2))-1] = np.mean(tempAccs)

    return nums[np.argmax(accs)], accs, index
```

When running these functions, we get the following results:

- k value for highest accuracy on training set: 21
- k_{best} accuracy on training set : ~ 0.7650
- Accuracy of k-NN against test set with $k = k_{best}$: ~ 0.7531

We see that the accuracy of k-NN with k_{best} is slightly more accurate than the originals we calculated in 5c, but no much. We can also see that our k-NN with k_{best} performs slightly worse on the test set than on the training set. This is likely due to pure chance, as we would expect the model to perform the same, or slightly better since it has been trained on the entire training set, instead of only 9/10ths of it.



(e)

This exercise has been completed in the attached *PCA and classification.ipynb* Jupyter Notebook. The most relevant bits of source code can be seen here as well, though all comments are left out. It should be noted that most of the logistic calculation functions have been taken and slightly modified from the lecture slides.

```
def logistic_insample(X, y, w):
    N, num_feat = X.shape
    w = w.reshape(num_feat,1)
    E = 0
    for n in range(N):
        xn = X[n,:].reshape(num_feat,1)
        E += (1/N)*np.log(1/logistic(y[n]*np.matmul(w.T,xn)))
    return E[0,0]

def logistic_gradient(X, y, w):
    N, num_feat = X.shape
    w = w.reshape(num_feat,1)
    g = np.zeros(w.shape)
```

```
    for n in range(N):
        xn = X[n,:].reshape(num_feat,1)
        increment = ((-1/N)*y[n]*xn)*logistic(-y[n]*np.matmul(w.T,xn))
        g += increment
    return g

def log_reg(Xorig, y, max_iter, grad_thr):
    num_pts, num_feat = Xorig.shape
    onevec = np.ones((num_pts,1))
    X = np.concatenate((onevec, Xorig), axis = 1)
    dplus1 = num_feat + 1
    learningrate = 0.1
    w = 0.1*np.random.randn(num_feat + 1).reshape(num_feat+1,1)
    value = logistic_insample(X,y,w)
    num_iter = 0
    convergence = 0
    E_in = []
    while convergence == 0:
        num_iter = num_iter + 1
        g = logistic_gradient(X,y,w)
        v = -g
        w_new = w + learningrate*v
        cur_value = logistic_insample(X,y,w_new)
        if cur_value < value:
            w = w_new
            value = cur_value
            E_in.append(value)
            learningrate *=1.1
        else:
            learningrate *= 0.9

        g_norm = np.linalg.norm(g)
        if g_norm < grad_thr:
            convergence = 1
            print('converged')
        elif num_iter > max_iter:
            convergence = 1
            print('reached maximum nr of iterations')

    return w, E_in

def log_pred(Xorig, w):
    num_pts, num_feat = Xorig.shape
    w = w.reshape(num_feat+1,1)
    onevec = np.ones((num_pts,1))
    X = np.concatenate((onevec, Xorig), axis = 1)
    P = np.zeros(num_pts)
```

```
    for n in range(num_pts):
        xn = X[n,:].reshape(num_feat+1,1)
        P[n] = logistic(np.matmul(w.T,xn)) # Probability of having label +1

    Pthresh = np.round(P) #0/1 class labels
    pred_classes = Pthresh*2-1
    return P, pred_classes
```

We see that it returns the following accuracies:

- Accuracy on training set: 0.6285
- Accuracy on test set: 0.6234

It seems that *k-NN* works better for this particular dataset. This, as we will also discover in $5f$ is likely because the different features' "position" seem to be important to which label that particular object is the correct one. The likely reason that *k-NN* is better at this is because it is based on Euclidean distances, while logistic regression is based on probability.

(f)

This exercise has been completed in the attached *PCA and classification.ipynb* Jupyter Notebook. The most relevant bits of source code can be seen here as well, though all comments are left out.

```
def center(centSet):
    return centSet-np.mean(centSet, axis=0)

def normalise(normSet):
    retSet = np.zeros(normSet.shape)
    for i in range(normSet.shape[1]):
        retSet[:,i] = (normSet[:,i] - min(normSet[:,i]))/(max(normSet[:,i]) - min(normSet[
    return retSet

def preprocess(procSet):
    return center(normalise(procSet))
```

We see that for both methods we get the exact same accuracy. This makes sense for the same reason that we got a higher accuracy with the *k-NN* algorithm earlier. It seems that much of the difference between the two classes lie in the "positional" values. When we essentially remove this by preprocessing, they are on more equal footing. However, since the difference is "positional" both algorithms lose accuracy when we normalise. Losing accuracy like this is something that is very dependent on the data set. Sometimes the noise you remove when centering and normalising actually contains information that can be used to determine its label.

- Accuracy of knn against test set with k_best: 0.5815
- Accuracy of logistic regression against test set: 0.5815