regularized-regression

January 16, 2019

1 Exercise 06

1.1 1 Bias and variance of ridge regression

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \underbrace{||X\beta - y||_2^2 + \tau\, ||\beta||_2^2}_{=:B}$$

We will now take the derivative of B with respect to β in order to find $\hat{\beta}$.

$$0 \overset{!}{=} \frac{\partial B}{\partial \beta} = \frac{\partial}{\partial \beta}\left(||X\beta - y||^2\right) + \frac{\partial}{\partial \beta}\left(\tau||\beta||^2\right) = 2X^\intercal\ (X\beta - y) + 2\tau\beta$$

$$\Rightarrow X^\intercal y = \left(X^\intercal X + \tau \mathbb{1}\right)\beta$$

$$\Rightarrow \hat{\beta} = \left(X^\intercal X + \tau \mathbb{1}\right)^{-1} X^\intercal y$$

$$\Rightarrow \hat{\beta} = \left(X^\intercal X + \tau \mathbb{1}\right)^{-1} \underbrace{\left(X^\intercal X\right)\left(X^\intercal X\right)^{-1}}_{\mathbb{1}} X^\intercal y = \left(X^\intercal X + \tau \mathbb{1}\right)^{-1} X^\intercal X \beta = S_\tau^{-1} S \beta$$

Now we calculate the expectation value:

$$\mathbb{E}[\hat{\beta}] = \mathbb{E}[S_\tau^{-1} S\beta] = S_\tau^{-1} S\ \mathbb{E}[\beta] S_\tau^{-1} S \beta^*$$

Covariance matrix:

$$\operatorname{Cov}\left[\hat{\beta}\right] = \operatorname{Cov}\left[S_\tau^{-1} S\beta\right] = \left(S_\tau^{-1} S\right) \underbrace{\operatorname{Cov}\left[\beta\right]}_{S^{-1}\sigma^2} \left(S_\tau^{-1} S\right)^\intercal = S_\tau^{-1} S^\intercal \left(S_\tau^{-1}\right)^\intercal \sigma^2 = S_\tau^{-2} S \sigma^2$$

1.2 2 Denoising of a CT image

```
In [1]: import numpy as np
        import scipy.sparse as sp
        import scipy.sparse.linalg as sLA
        import matplotlib.pyplot as plt
        %matplotlib inline

In [2]: def construct_X(M, alphas, Np = None, tau = 0):
            '''
            M: resulting tomogram size D in one dimension (D=MxM)
            alphas: list of No angles in degrees
            Np: sensor resolution (optional)
```

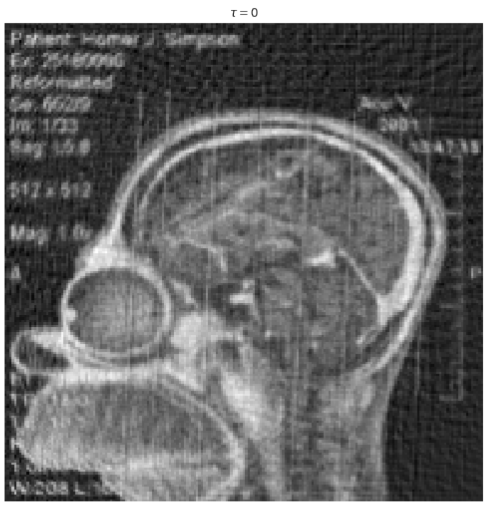
```
tau: regularization parameter
returns X of shape (Np*No)xD or (Np*No+D)xD if tau is greater than 0
'''
#check if tau is not negative:
if tau<0:
    raise ValueError('tau is negative')
#convert to numpy array and into radian
alphas=np.array(alphas)*np.pi/180
#in case Np is not given we will select one large enough to fit the dia
if Np==None:
    Np=np.ceil(np.sqrt(2)*M)
    if Np%2==0: #Np is even
        Np+=1

N=len(alphas)*Np #response vector size
D=M*M
#create coordinate matrix C of the ceter of each pixel measured from th

#"x" distance
C_0 = np.mod(np.arange(D),M) - (M - 1)/2
#"y" distance
C_1 = np.floor_divide(np.arange(D),M) - (M - 1)/2
#merge for C
C = np.array([C_0,C_1]).T
#project the C vector onto the sensor array direction
translation = C@np.array([np.cos(alphas),np.sin(-alphas)])
#this is now projected on the senor using the position of the center as
sensorpos = (Np - 1)/ 2 + translation.T
#first the indices of the smaller pixels then the ones of the larger on
i_p = np.array([np.floor(sensorpos), np.ceil(sensorpos)]).flatten()
#[0]*D,[1]*D,...,[No-1]*D twice repeated
i_o = np.tile(np.outer(np.arange(len(alphas)),np.ones(D)).flatten(),2)
#combine them to our i_indices
i_indices =  i_p + Np * i_o

#simply [0,1,...,No-1]*2D
j_indices = np.tile(np.tile(np.arange(D),len(alphas)),2)

#compute the weights
weight_ceil = np.mod(sensorpos, 1).flatten()
weight_floor = 1 - weight_ceil
#and combine them in the correct order
weights = np.append(weight_floor,weight_ceil)

#now we build the matrix X:
X = sp.coo_matrix((weights, (i_indices, j_indices)),shape = (N,D), dtyp
#if tau=0 we return the X matrix...
if tau==0:
```

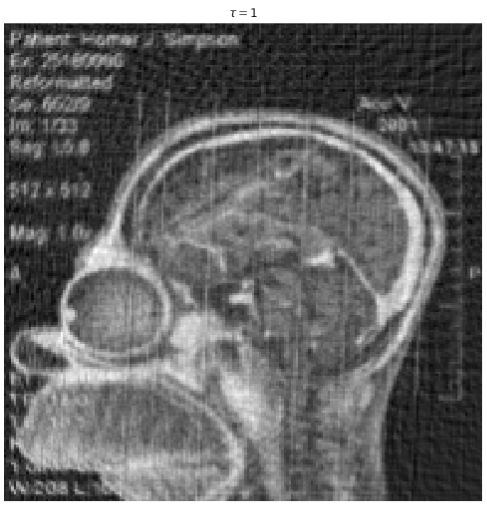
```
            return X
        elif tau>0:
            #... if not we return the augmented version X'
            return sp.vstack([X,np.sqrt(tau)*sp.identity(D,format='coo')])
```

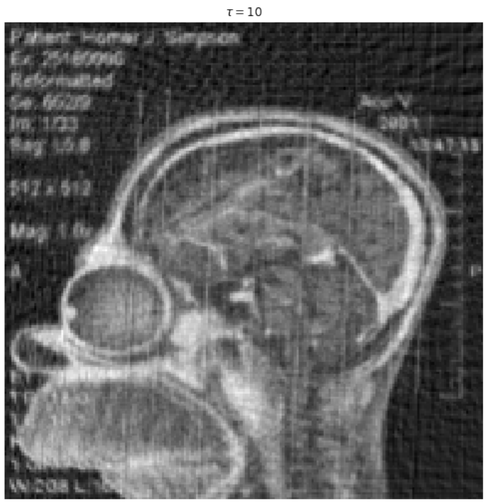
```
In [3]: def compute_mu(X,y):
            '''
            X: weight matrix of shape (N,D)
            y: sensor data
            returns the image
            '''
            #use scipy.sparse.linalg.lsqr to calculate beta, also convert X to Comp
            beta=sLA.lsqr(sp.csc_matrix(X),y,atol=1e-5,btol=1e-5)[0]
            #get M from D
            M=int(np.sqrt(len(beta)))
            #return the image mu by reshaping beta
            return beta.reshape(M,M)
```

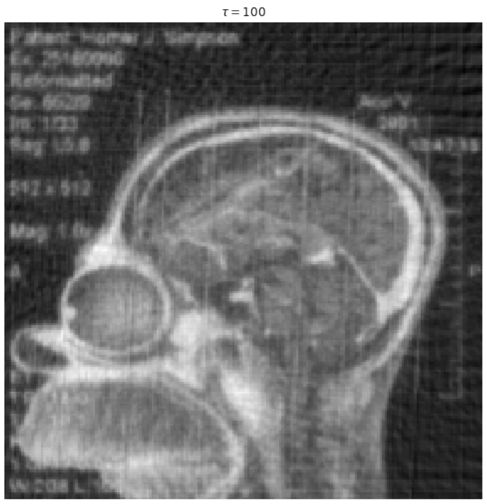
```
In [4]: alphas = np.load('hs_tomography/alphas_195.npy')
        y = np.load('hs_tomography/y_195.npy')
```

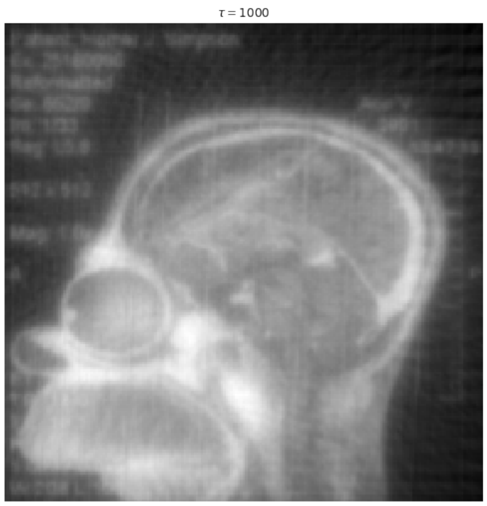
```
In [5]: fig=plt.figure(figsize=(20,30))
        M,Np,No=195,275,64
        #regularization parameter at each step
        taus=[0, 1, 10, 100, 1000, 10000]
        #64 angels:
        indices=[int(np.ceil(len(alphas) * p/No)) for p in range(No)]
        angels=alphas[indices]
        ys=y.reshape(len(alphas),Np)[indices]
        #if tau is not 0 we need to add D zeros
        ytau=np.append(ys,np.zeros(M**2))

        for i in range(len(taus)):
            fig.add_subplot(int(len(taus)/2),2,1+i)
            plt.gray()
            #select the correct y
            yi=ys if taus[i]==0 else ytau
            plt.imshow(compute_mu(construct_X(M,angels,Np,taus[i]),yi.flatten()))
            plt.title(r'$\tau= %i$'%(taus[i]))
            plt.axis('off')
```

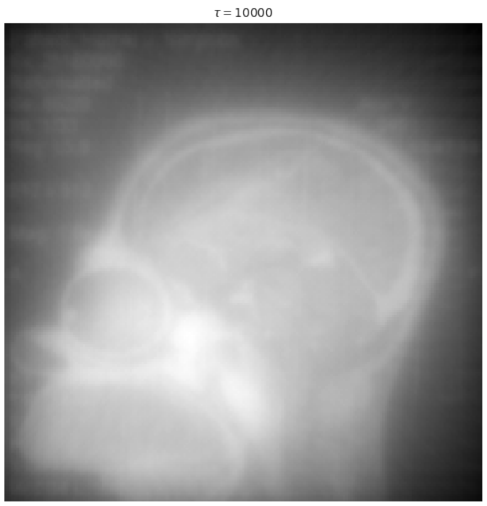

$\tau = 0$


$\tau = 1$


$\tau = 10$


$\tau = 100$

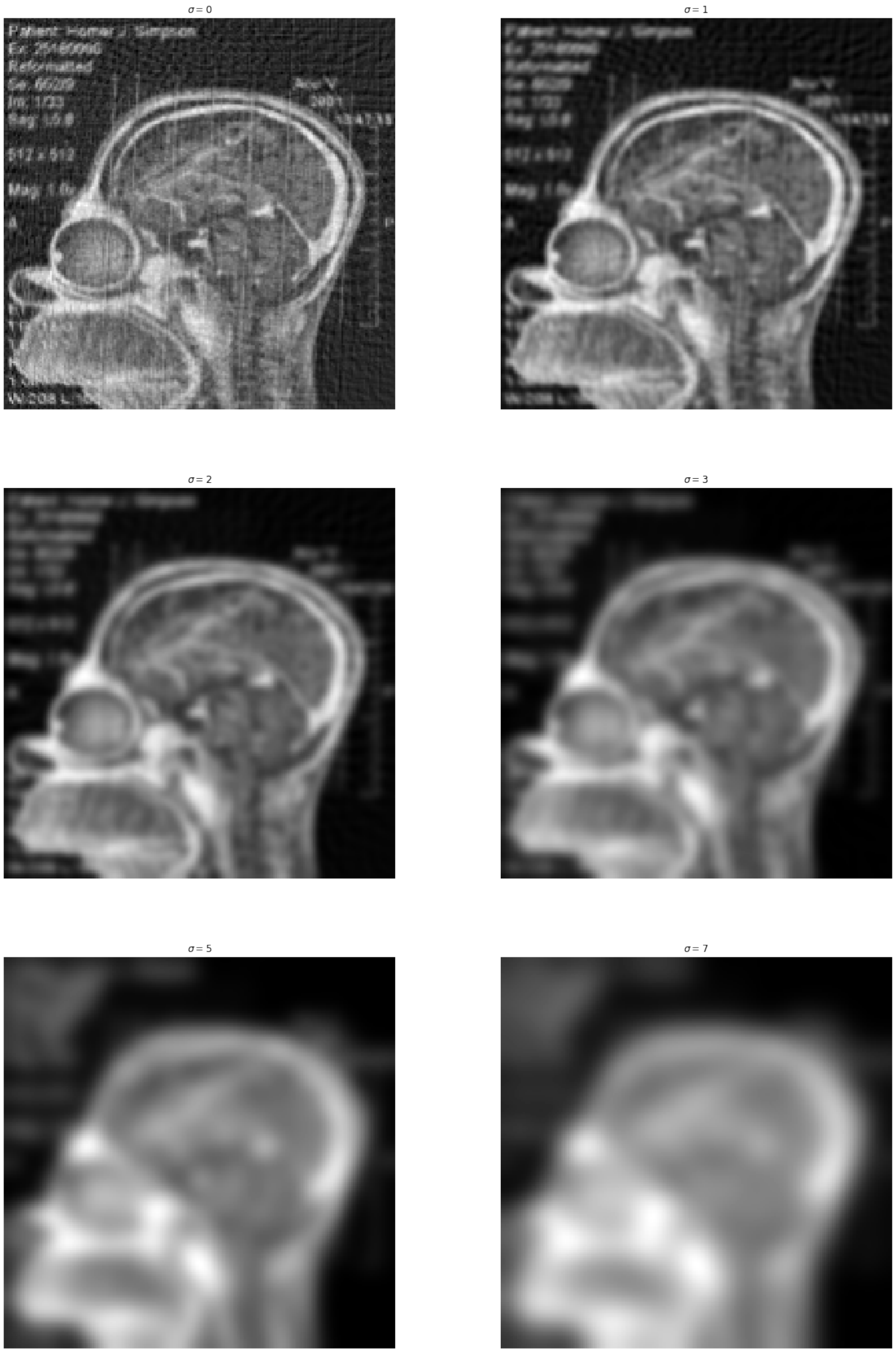

$\tau = 1000$


$\tau = 10000$

out of the given τ's $\tau = 10$ and $\tau = 100$ have the best compromise between the noise and the sharpness of the picture without the glowing effect that's visible for larger τ's. Maybe a better τ could be found in between the two values 10 and 100.

```
In [6]: #Now use the gaussian filter
        from scipy.ndimage.filters import gaussian_filter
```

```
In [7]: fig=plt.figure(figsize=(20,30))
        M,Np,No=195,275,64
        #regularization parameter at each step
        sig=[0,1,2,3,5,7]
        #64 angels:
        indices=[int(np.ceil(len(alphas) * p/No)) for p in range(No)]
        angels=alphas[indices]
        ys=y.reshape(len(alphas),Np)[indices]
        #mu for tau=0:
        Mu=compute_mu(construct_X(M,angels,Np,0),ys.flatten())
        for i in range(len(sig)):
            fig.add_subplot(int(np.ceil(len(sig)/2)),2,1+i)
            plt.gray()
            #select the correct y
            yi=ys if taus[i]==0 else ytau
            plt.imshow(gaussian_filter(Mu,sig[i]))
            plt.title(r'$\sigma= %i$'%(sig[i]))
            plt.axis('off')
```



$\sigma = 0$
$\sigma = 1$
$\sigma = 2$
$\sigma = 3$
$\sigma = 5$
$\sigma = 7$

For a high sigma the picture becomes blurry. It seems like information is lost. The ridge regression on the other hand develops a glow effect but, so the overall intensity is getting higher but we still can recognize features of the patients face without problems. Both algorithms are denoising the image. The gaussian filter is better at removing the background noise but in the process the image looses sharpness.

1.3 3 Automatic feature selection for regression

```
In [8]: from sklearn.datasets import load_digits
        from sklearn import model_selection
        #for least squares
        from sklearn import linear_model
```

1.3.1 3.1 Implement Orthogonal Matching Pursuit

```
In [9]: def omp_regression(X,y,T):
            '''
            X: ndarray with shape (N,D)
            y: ndarray with shape (N)
            T: int, desired number of non-zero elements
            returns a matrix of betas of the shape (D,T)
            '''
            #check if T > 0
            assert T>0
            #Initialization
            A = []
            B = [*range(X.shape[1])] #0...D-1
            r = y
            solutions = []
            #Iteration
            for t in range (1, T + 1):
                corr = np.abs(np.matmul(X.T, r))
                corr[A] = -1 # we are only interested in indices from B
                j = np.argmax(corr)
                A.append(B.pop(B.index(j))) #move the element j from B to A
                X_t = np.delete(X, B, axis = 1)
                reg = linear_model.LinearRegression()
                reg.fit(X_t, y)
                beta = reg.coef_
                r = y - np.matmul(X_t,beta)
                beta=np.insert(beta,(np.sort(B) - np.arange(len(B))).astype(int),0)
                beta[B] = 0
                solutions.append(beta)
                #solutions.append(beta)
            return sp.csc_matrix(np.transpose(solutions))
```

1.3.2 3.2 Classification with sparse LDA

```
In [10]: #load digits
         digits = load_digits ()
         print(digits.keys ()) #Python 3
         data = digits ["data"]
         images = digits ["images"]
         target = digits ["target"]
         target_names = digits ["target_names"]
```

```
dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])
```

```
In [11]: # filter out all 1's and 7's
         X_all = data[(target==1)|(target==7)]
         y_all = target[(target==1)|(target==7)]
         y_all[y_all == 7] = -1 # y_i=-1 if i is a '7'

         # split into train and test data
         X_train, X_test, y_train , y_test = model_selection.train_test_split(X_all
         #choose randomly instances so that the trainig set is balanced (same amoun
         N=sum(y_train)

         instances=np.random.permutation(len(y_train[y_train==N/abs(N)]))[:abs(N)]
         X_test=np.append(X_test,X_train[y_train==N/abs(N)][instances],axis=0)
         X_train=np.delete(X_train,np.arange(len(y_train))[y_train==N/abs(N)][insta

         y_test=np.append(y_test,y_train[y_train==N/abs(N)][instances],axis=0)
         y_train=np.delete(y_train,np.arange(len(y_train))[y_train==N/abs(N)][insta
         #surly not the best way to do it but it works
```

```
In [12]: T=40
         LDAsol=omp_regression(X_train,y_train,T)
         prediction=X_test@LDAsol
         #split the predictions at 0
         prediction[prediction<0]=-1
         prediction[prediction>=0]=1
         #substract y_test from each column
         difference=abs(prediction-y_test[:,None])/2 #need to divide by two because
         errorRate=np.mean(difference,axis=0)
         var=np.var(difference,axis=0)
```

```
In [13]: for i in range(T):
             print('t=%i, Error rate: (%.2f +/- %.2f)%s'%(i+1,errorRate[i]*100,100*
```

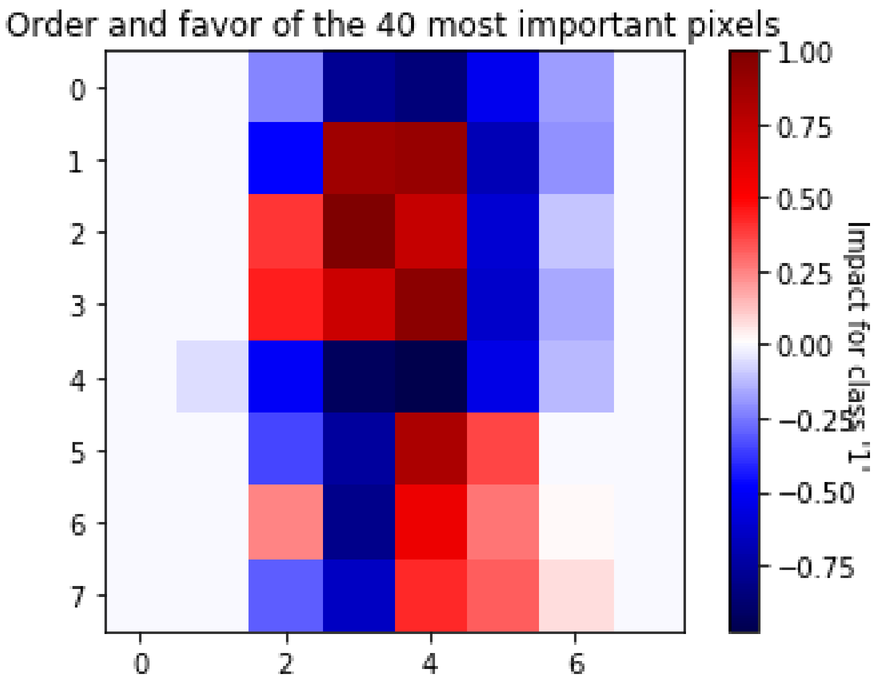
```
t=1, Error rate: (49.14 +/- 49.99)%
t=2, Error rate: (49.14 +/- 49.99)%
t=3, Error rate: (49.14 +/- 49.99)%
t=4, Error rate: (48.57 +/- 49.98)%
```

```
t=5, Error rate: (48.00 +/- 49.96)%
t=6, Error rate: (40.57 +/- 49.10)%
t=7, Error rate: (38.86 +/- 48.74)%
t=8, Error rate: (36.00 +/- 48.00)%
t=9, Error rate: (34.86 +/- 47.65)%
t=10, Error rate: (18.86 +/- 39.12)%
t=11, Error rate: (25.71 +/- 43.71)%
t=12, Error rate: (22.86 +/- 41.99)%
t=13, Error rate: (22.86 +/- 41.99)%
t=14, Error rate: (28.57 +/- 45.18)%
t=15, Error rate: (32.00 +/- 46.65)%
t=16, Error rate: (8.57 +/- 27.99)%
t=17, Error rate: (8.57 +/- 27.99)%
t=18, Error rate: (8.57 +/- 27.99)%
t=19, Error rate: (8.57 +/- 27.99)%
t=20, Error rate: (8.57 +/- 27.99)%
t=21, Error rate: (8.57 +/- 27.99)%
t=22, Error rate: (4.00 +/- 19.60)%
t=23, Error rate: (3.43 +/- 18.20)%
t=24, Error rate: (2.86 +/- 16.66)%
t=25, Error rate: (2.29 +/- 14.94)%
t=26, Error rate: (2.29 +/- 14.94)%
t=27, Error rate: (2.29 +/- 14.94)%
t=28, Error rate: (2.86 +/- 16.66)%
t=29, Error rate: (2.29 +/- 14.94)%
t=30, Error rate: (1.71 +/- 12.98)%
t=31, Error rate: (1.14 +/- 10.63)%
t=32, Error rate: (1.14 +/- 10.63)%
t=33, Error rate: (0.57 +/- 7.54)%
t=34, Error rate: (0.57 +/- 7.54)%
t=35, Error rate: (0.57 +/- 7.54)%
t=36, Error rate: (0.57 +/- 7.54)%
t=37, Error rate: (0.00 +/- 0.00)%
t=38, Error rate: (0.00 +/- 0.00)%
t=39, Error rate: (0.00 +/- 0.00)%
t=40, Error rate: (0.00 +/- 0.00)%
```

```
In [14]: featureList=np.where(LDAsol.todense().T!=0)[1] #all features contributing
         featureList=featureList[np.sort(np.unique(featureList,return_index=True)[1

In [15]: #to check if a certain pixel is in favour of class '1' we'll take the aver
         #'1' and '7' class instances and check which value is higher.
         averageDifference=np.mean(X_train[y_train==1][:,featureList],axis=0)-np.me
         #divide by absolute value for only 1 and -1 as entries
         averageDifference=averageDifference/abs(averageDifference)

In [16]: order=np.zeros(64)
```

```
    for i in range(T):
        order[featureList[::-1][i]]=(i+1)/T*averageDifference[::-1][i]
```

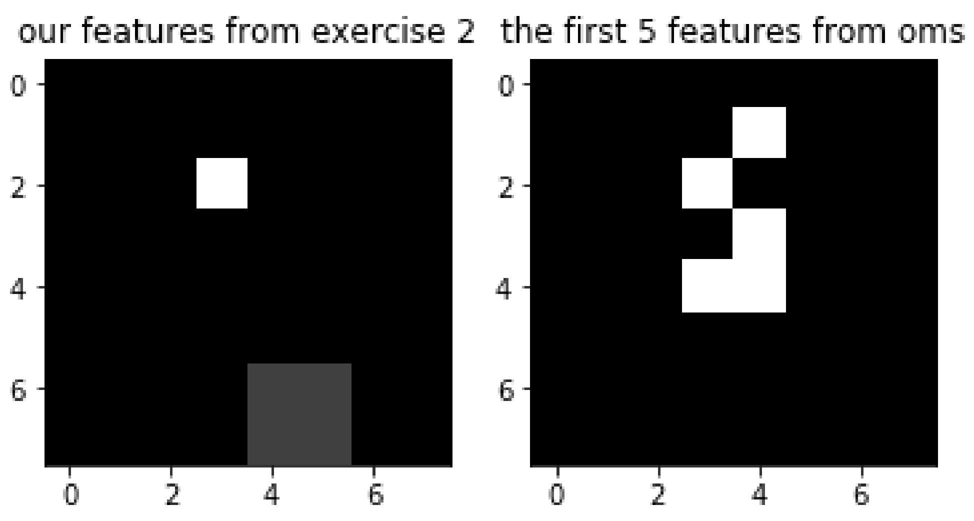
```
In [17]: plt.title('Order and favor of the %i most important pixels'%T)
         plt.imshow(order.reshape((8,8)),cmap='seismic')
         plt.colorbar().set_label("Impact for class '1'",rotation=270)
```



The darker the pixels the more important are they for the corresponing class. We can roughly a blue 7 in front of a red one. That shows that the pixels were classified correcly by the higher average.

```
In [18]: plt.subplot('121')
         plt.title('our features from exercise 2')
         f=np.zeros((8,8))
         f[2,3]=1
         f[7,4]=f[7,5]=f[6,4]=f[6,5]=.25
         plt.imshow(f)
         plt.subplot('122')
         plt.title('the first 5 features from oms')
         g=np.zeros(64)
         g[featureList[:5]]=1
         g=g.reshape((8,8))
         plt.imshow(g)
```

```
Out[18]: <matplotlib.image.AxesImage at 0x220bede4198>
```



We chose one feature like the algorithm. Our other feature however used other pixels.

1.4 3.3 One-against-the-rest classification

```
In [19]: import pandas as pd
         pd.options.display.float_format = '{:,.2f}'.format
```

```
In [22]: class classifier:
             def __init__(self,k,T):
                 self.k=k
                 self.T=T
             def train(self,X,y):
                 y[y!=self.k]=-1
                 y[y==self.k]=1
                 self.beta=omp_regression(X,y,self.T).toarray().T[self.T-1]
             def predict(self,X):
                 return X@self.beta
```

```
In [23]: class OneAgainstTheRestClassifier:
             def __init__(self,classes,T):
                 self.classes=classes
                 Cs=[]
                 for i in range(len(classes)):
                     Cs.append(classifier(classes[i],T))
                 self.C=Cs
             def train(self,X,y):
                 for i in range(len(self.classes)):
                     self.C[i].train(X,y.copy())
             def predict(self,X):
```

```
            def predictOneInstance(X):
                predictions=[]
                for i in range(len(self.classes)):
                    predictions.append(self.C[i].predict(X))
                response=np.argmax(predictions)
                if predictions[response]<0: #biggest response is negative
                    return None #class "unknown"
                else:
                    return self.classes[response]

            if type(X)==np.ndarray:
                if len(X.shape)==2: #whole set
                    result=[]
                    #could have been vectorized but it still is reasonably fas
                    for i in range(len(X)):
                        result.append(predictOneInstance(X[i]))
                    return np.array(result)
                elif len(X.shape)>2:
                    raise ValueError('Wrong dimensions for X')
            #only one instance to predict
            return predictOneInstance(X)
```

```
In [24]: # split into train and test data, this time with all digits
         xTrain, xTest, yTrain , yTest = model_selection.train_test_split(data,targ
```

```
In [50]: oatr=OneAgainstTheRestClassifier(np.arange(10),40)
         oatr.train(xTrain,yTrain)
```

```
In [51]: #build confusion matrix
         conf=np.zeros((10,11))
         data_subsets = [xTest[yTest==i] for i in  range(10)]
         for i in range(10):
             # predict with OneAgainstTheRestClassifier classifier
             predictions = np.array([oatr.predict(j) for j in data_subsets[i]])
             predictions[pd.isnull(predictions)]=10
             predictions=predictions.astype(int)
             conf[i,:] = np.bincount(predictions,minlength=11)/len(data_subsets[i])
```

```
In [52]: matrix=pd.DataFrame(data = conf, index =range(10), columns =[*range(10),'u
         matrix.rename_axis('Actual/Predicted', axis = 'columns')
         matrix
```

```
Out[52]:       0      1      2      3      4      5      6      7      8      9  unknown
         0 96.67   0.00   0.00   1.67   0.00   1.67   0.00   0.00   0.00   0.00     0.00
         1  0.00  93.15   1.37   0.00   0.00   0.00   0.00   0.00   5.48   0.00     0.00
         2  0.00   8.45  80.28   4.23   0.00   0.00   0.00   1.41   5.63   0.00     0.00
         3  0.00   0.00   0.00  92.86   0.00   0.00   0.00   0.00   5.71   1.43     0.00
```

```
4   0.00   3.17   0.00   0.00 90.48   0.00   0.00   1.59   4.76   0.00      0.00
5   0.00   1.12   0.00   0.00   0.00 95.51   1.12   0.00   0.00   2.25      0.00
6   0.00   1.32   0.00   0.00   0.00   2.63 90.79   0.00   5.26   0.00      0.00
7   0.00   0.00   0.00   3.08   0.00   0.00   0.00 96.92   0.00   0.00      0.00
8   0.00   6.41   0.00   0.00   0.00   0.00   0.00   0.00 92.31   1.28      0.00
9   0.00   0.00   0.00   6.76   0.00   2.70   0.00   0.00   9.46 81.08      0.00
```

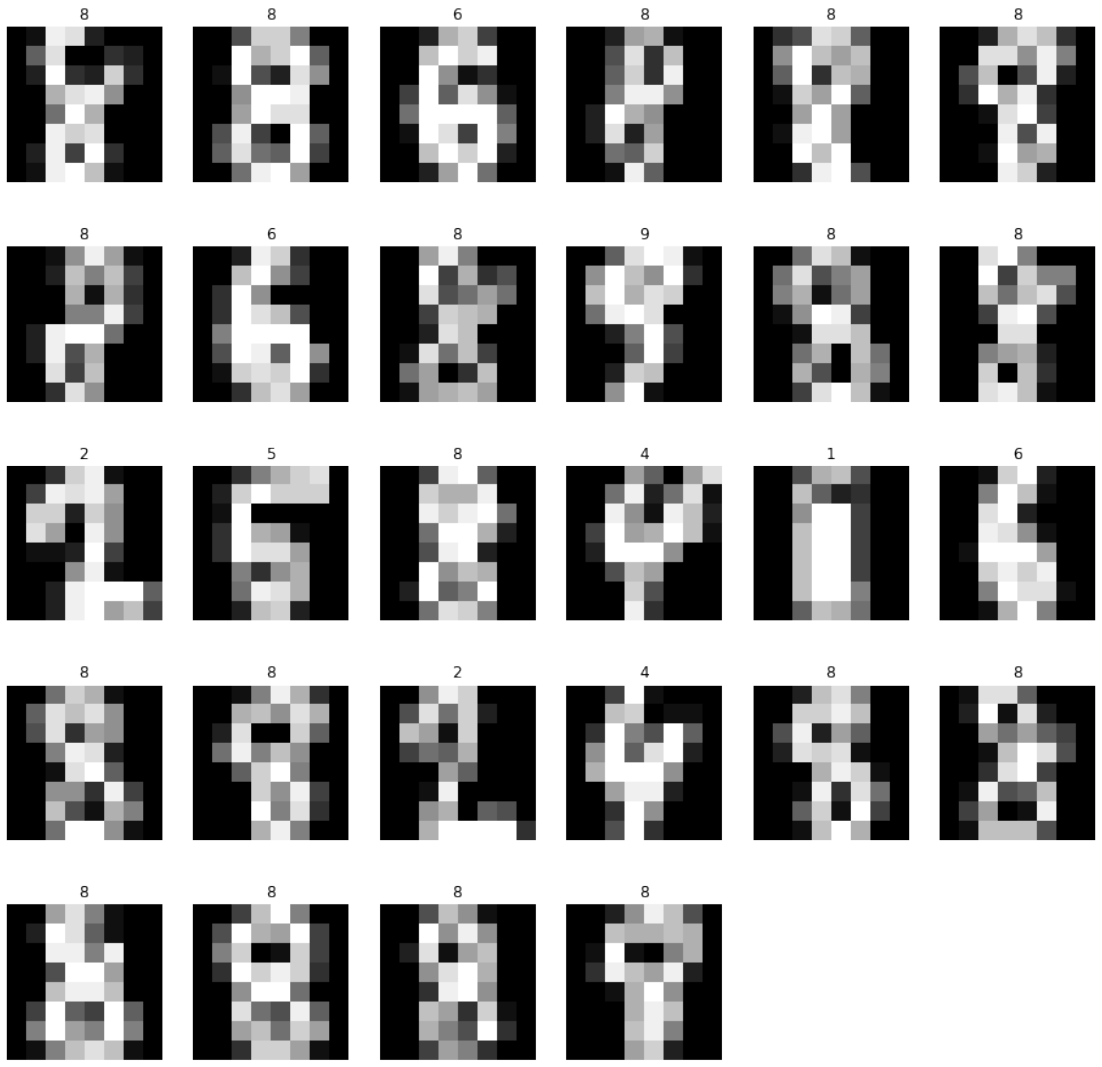
More than 80% of each digit were classified correctly. Except for the 3 and the 9 the error rate was even $< 10\%$. For the digit 7 96.92% of the test data was classified correctly.

An "unknown" class is useful because if we have a corrupted test instance it is better to not take the highest value if it is still negative because that classifier did not recognize the sample as it's class and decided against it.

```
In [35]: badOatr=OneAgainstTheRestClassifier(np.arange(10),12) #small T asures man
         badOatr.train(xTrain,yTrain)
```

```
In [36]: pred=badOatr.predict(xTest)
         imag=xTest[pd.isnull(pred)]
         label=yTest[pd.isnull(pred)]
```

```
In [37]: plt.figure(figsize=(15,15))
         for i in range(len(imag)):
             n=int(np.sqrt(len(imag)))
             plt.subplot(n,np.ceil(len(imag)/n),i+1)
             plt.axis('off')
             plt.imshow(imag[i].reshape((8,8)))
             plt.title(label[i])
```



As we can see most of the "unknown" images represent a 8. That is because the digit 8 the most pixels from the whole set to be displayed. For a small T we are limiting the amount of pixels hence the classification of the digit 8 becomes harder. The rest of the digits are hard to read even when you know what digit it is supposed to display.

```
In [ ]:
```