sheet6

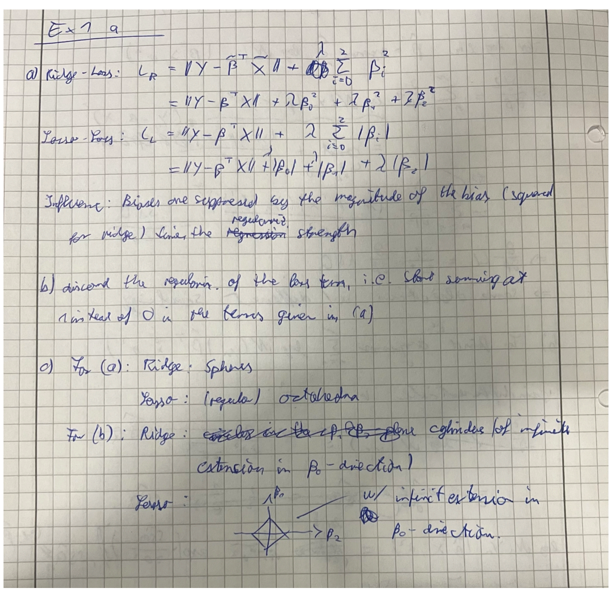
December 5, 2022

1 Sheet 6

Leonard Benkendorff, Dorothea Schwärzel, Sebastian Preuß

[1]:
```
import numpy as np
from matplotlib import pyplot as plt
import scipy.sparse
from sklearn.linear_model import Ridge, LinearRegression
```

1.1 1 Regularization and Bias



Ex 1 a

a) Ridge-Loss: $L_R = \|Y - \tilde{\beta}^T \tilde{X}\| + \lambda \sum_{i=0}^{2} \beta_i^2$

$= \|Y - \beta^T X\| + \lambda \beta_0^2 + \lambda \beta_1^2 + \lambda \beta_2^2$

Lasso-Loss: $L_L = \|Y - \beta^T X\| + \lambda \sum_{i=0}^{2} |\beta_i|$

$= \|Y - \beta^T X\| + \lambda|\beta_0| + \lambda|\beta_1| + \lambda|\beta_2|$

Influence: Biases are suppressed by the magnitude of the bias (squared for ridge) times the ~~regression~~ regularization strength

b) discard the regularization of the loss term, i.e. start summing at 1 instead of 0 in the terms given in (a)

c) For (a): Ridge: Spheres

Lasso: (regular) octahedra

For (b): Ridge: ~~circles in the β_1 β_2-plane~~ cylinders (of infinite extension in β_0-direction)

Lasso: β_0 β_2 w/ infinite extension in β_0-direction.

1.2 2 Estimating Parameter Relevance

[2]:
```
# load the data
with open('data/vostok.txt', 'r') as f:
    lines = f.readlines()

# remove header and split lines
lines = [l.split() for l in lines[2:]]
```

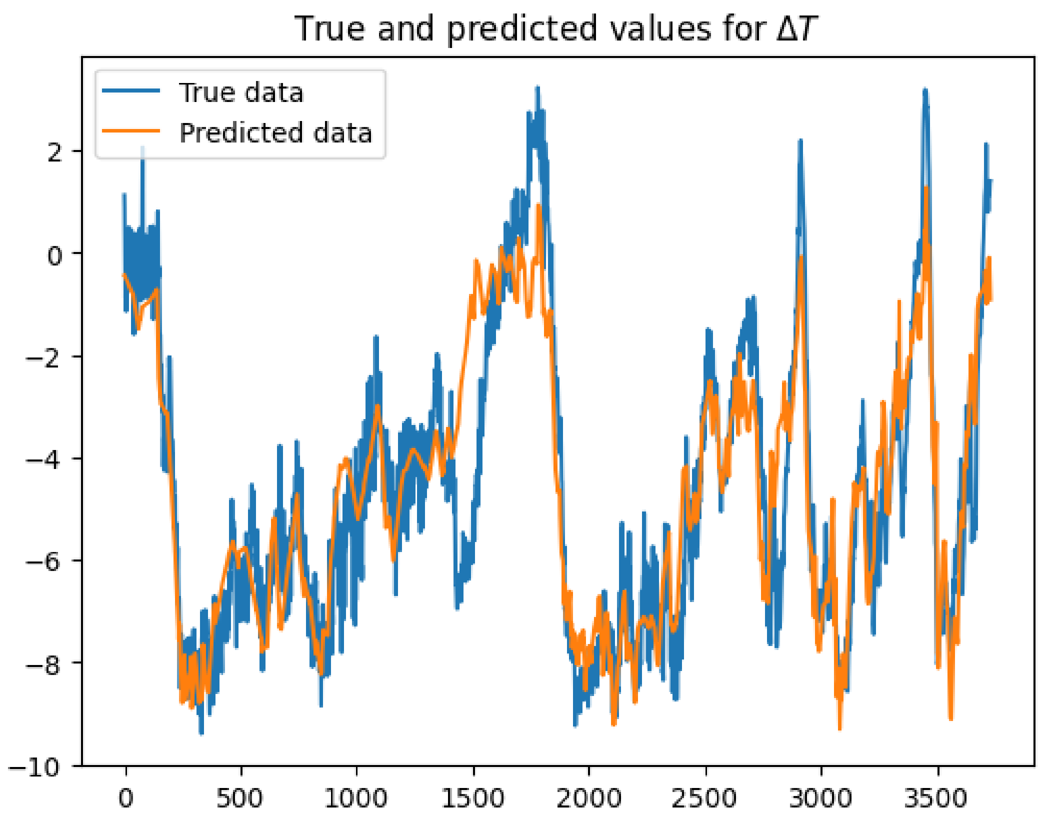
```python
# filter out lines with missing data
lines = [l for l in lines if len(l) == 4]

# convert to float
lines = np.array(lines).astype(np.float32)
print(f'{lines.shape=}')

features = np.concatenate([lines[:, :1], lines[:, 2:]], axis=1).T
feature_names = 'age', 'CO ', 'dust'
labels = lines[:, 1]
label_name = ' T'

print(f'{features.shape=}, {labels.shape=}')
```

```
lines.shape=(3729, 4)
features.shape=(3, 3729), labels.shape=(3729,)
```

[3]:

```python
from sklearn.linear_model import LinearRegression

# TODO: fit the linear regressor and compute the sum of square deviations
lr = LinearRegression()
lr.fit(features.T, labels)

print('Sum of squared deviations:', np.sum((lr.predict(features.T)-labels)**2))

plt.plot(labels, label='True data')
plt.plot(lr.predict(features.T), label='Predicted data')
plt.legend(loc='best')
plt.title(r'True and predicted values for $\Delta T$')
plt.show()
```

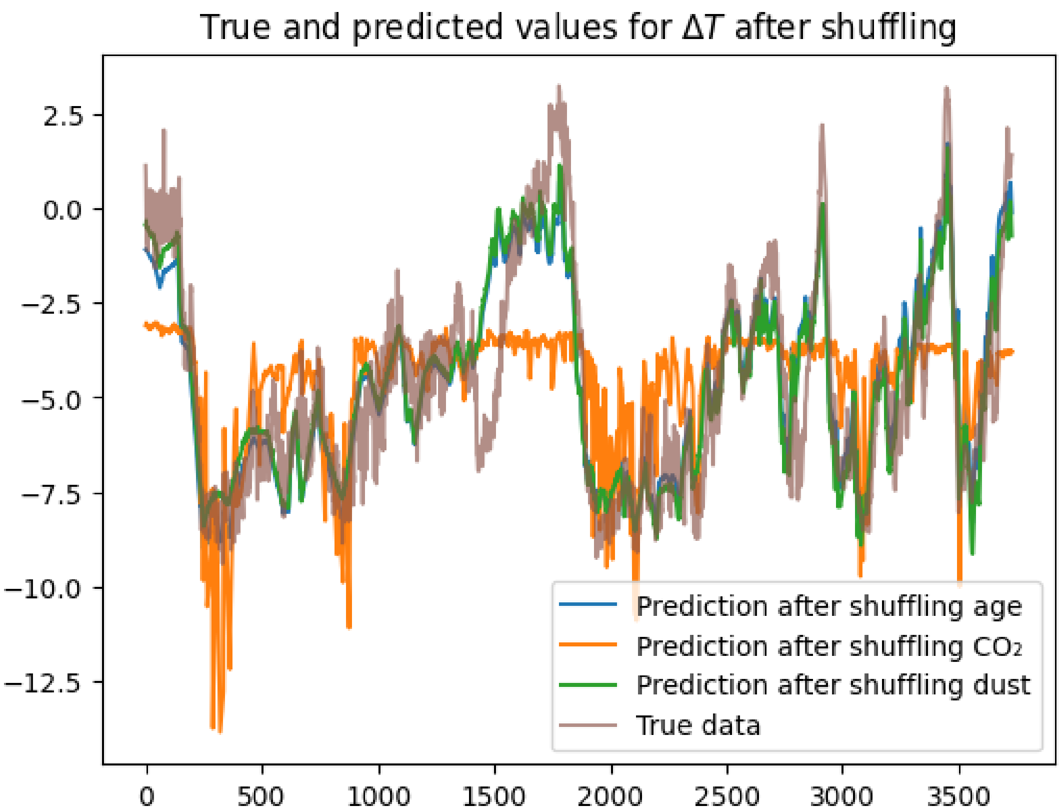
```
Sum of squared deviations: 6362.9375
```



[4]:
```
# TODO: for each feature, randomly permute it amongst the samples,
#       refit the regressor and compte sum of squared deviations
for i in range(3):
    new_order = np.arange(features.shape[-1])
    np.random.shuffle(new_order)
    shuffled_features = np.zeros(features.shape)
    for j in range(3):
        if i == j:
            shuffled_features[i,:] = features[i,new_order]
        else:
            shuffled_features[j,:] = features[j,:]

    lr = LinearRegression()
    lr.fit(shuffled_features.T, labels)
    print(f'Sum of squared deviations after shuffling {feature_names[i]}:', np.
 ↪sum((lr.predict(features.T)-labels)**2))
    plt.plot(lr.predict(shuffled_features.T), label=f'Prediction after␣
 ↪shuffling {feature_names[i]}')
plt.plot(labels, label='True data', color='#8c564baa') # add some transparency
plt.legend(loc='best')
```

```
plt.title(r'True and predicted values for $\Delta T$ after shuffling')
plt.show()
```

```
Sum of squared deviations after shuffling age: 6796.764275972411
Sum of squared deviations after shuffling CO : 18962.56096840998
Sum of squared deviations after shuffling dust: 6565.202563606381
```



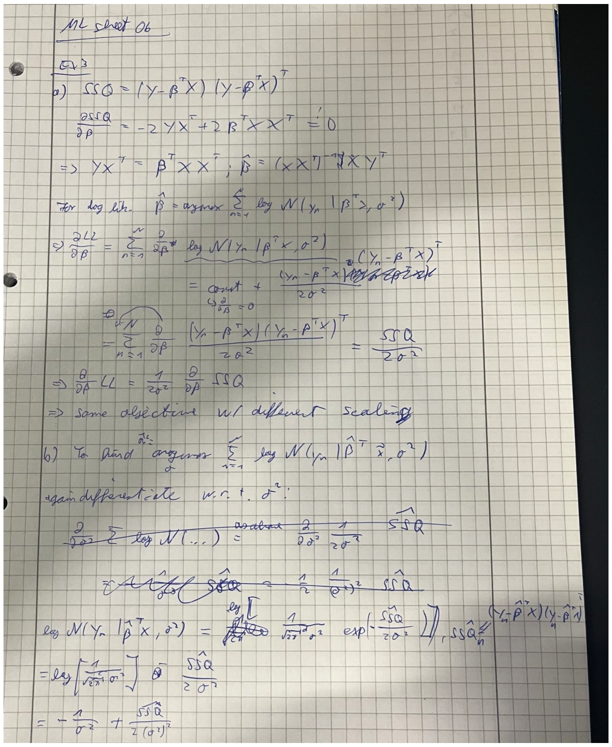
Which feature is most important and which is least relevant? The difference to the original sum of squared deviations is as follows:

$$
\begin{aligned}
\Delta_{\text{age}} &= |6362.9375 - 6824.1124| &= 461.2 \quad (1)\\
\Delta_{\text{CO}_2} &= |6362.9375 - 18944.9081| &= 12582.0 \quad (2)\\
\Delta_{\text{dust}} &= |6362.9375 - 6537.4003| &= 174.5 \quad (3)
\end{aligned}
$$

(This may differ from the output above if the code is re-run, because `np.random.shuffle` returns a new, random, result every time).

We conclude: The feature CO_2 is the most important one, the feature dust is the least important one. This conclusion can (at least for CO_2) also be drawn from the visualisation we included.

1.3 3 σ^2-estimation and Heteroscedastic Noise


ML sheet 06

Ex 3

a) $SSQ = (Y - \beta^T X)(Y - \beta^T X)^T$

$$\frac{\partial SSQ}{\partial \beta} = -2YX^T + 2\beta^T X X^T \overset{!}{=} 0$$

$$\Rightarrow YX^T = \beta^T X X^T ;\ \hat{\beta} = (XX^T)^{-1} X Y^T$$

For log lik. $\hat{\beta} = \operatorname{argmax} \sum_{n=1}^{N} \log \mathcal{N}(y_n \mid \beta^T x, \sigma^2)$

$$\Rightarrow \frac{\partial LL}{\partial \beta} = \sum_{n=1}^{N} \frac{\partial}{\partial \beta} \log \mathcal{N}(y_n \mid \beta^T x, \sigma^2)$$

$$= \text{const} + \frac{(y_n - \beta^T x)(y_n - \beta^T x)^T}{2\sigma^2}$$

$\frac{\partial}{\partial \beta} = 0$

$$= \sum_{n=1}^{N} \frac{\partial}{\partial \beta} \frac{(y_n - \beta^T x)(y_n - \beta^T x)^T}{2\sigma^2} = \frac{SSQ}{2\sigma^2}$$

$$\Rightarrow \frac{\partial}{\partial \beta} LL = \frac{1}{2\sigma^2} \frac{\partial}{\partial \beta} SSQ$$

$\Rightarrow$ same objective w/ different scaling

b) To find $\hat{\sigma}^2 = \operatorname{argmax}_{\sigma} \sum_{n=1}^{N} \log \mathcal{N}(y_n \mid \hat{\beta}^T \vec{x}, \sigma^2)$

again differentiate w.r.t. σ^2:

$$\cancel{\frac{\partial}{\partial \sigma^2} \sum \log \mathcal{N}(\dots) = \frac{\partial}{\partial \sigma^2} \frac{1}{2\sigma^2} \widehat{SSQ}}$$

$$\cancel{= -\frac{1}{2} \frac{1}{(\sigma^2)^2} \widehat{SSQ}}$$

$$\log \mathcal{N}(y_n \mid \hat{\beta}^T x, \sigma^2) = \log\left[\frac{1}{\sqrt{2\pi}\,\sigma^2} \exp\left(-\frac{\widehat{SSQ}}{2\sigma^2}\right)\right],\ \widehat{SSQ}_n = (y_n - \hat{\beta}^T x)(y_n - \hat{\beta}^T x)^T$$

$$= \log\left[\frac{1}{\sqrt{2\pi}\,\sigma^2}\right] - \frac{\widehat{SSQ}}{2\sigma^2}$$

$$= -\frac{1}{\sigma^2} + \frac{\widehat{SSQ}}{2(\sigma^2)^2}$$

Comparison to SSQ: In the lecture we derived the covariance of β from the distribution of the variance of the original data. We did not find an expression for the likeliest variance σ^2.

1.4 4 Visualize Regularization Contours

[5]:
```python
# load the data
data = np.load('data/linreg.npz')
x = data['X']
y = data['Y']
print(f'{x.shape} {y.shape}')
```

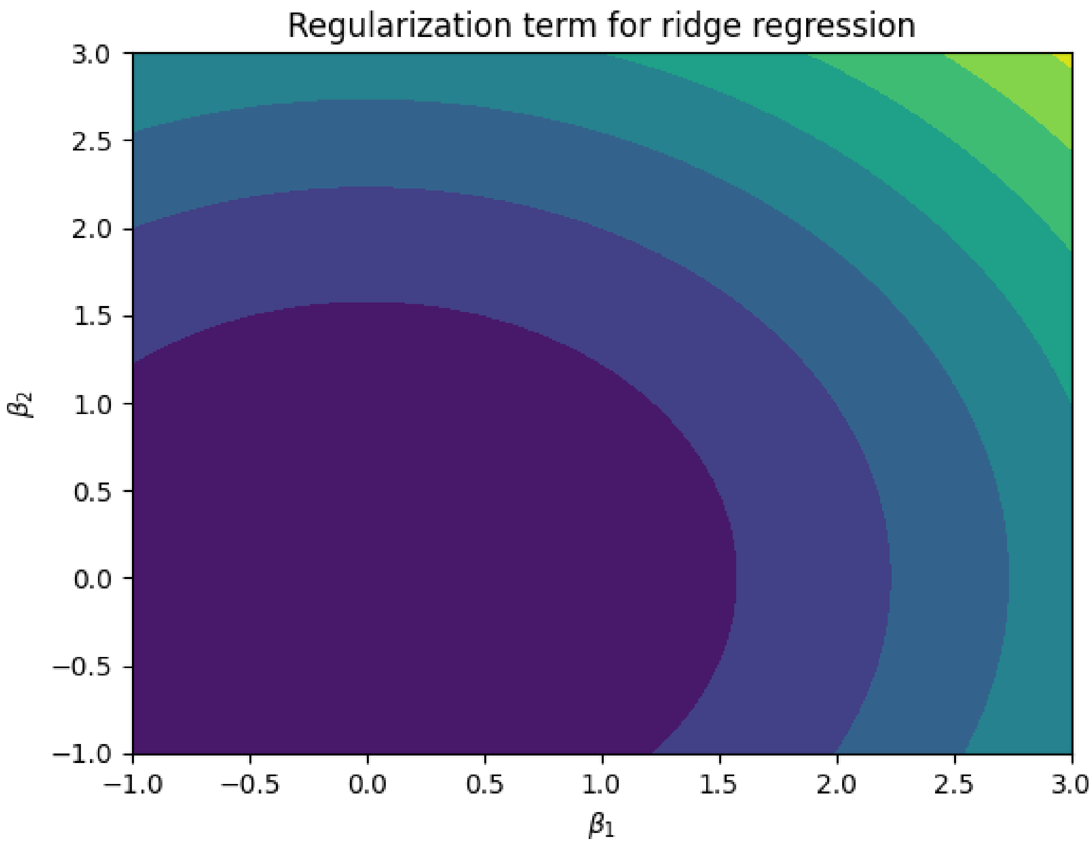
```
(2, 100) (1, 100)
```

[6]:
```python
# TODO: create a grid of points in the parameter space
beta1 = np.linspace(-1,3,150)
beta2 = np.linspace(-1,3,150)
beta1,beta2 = np.meshgrid(beta1, beta2)
```

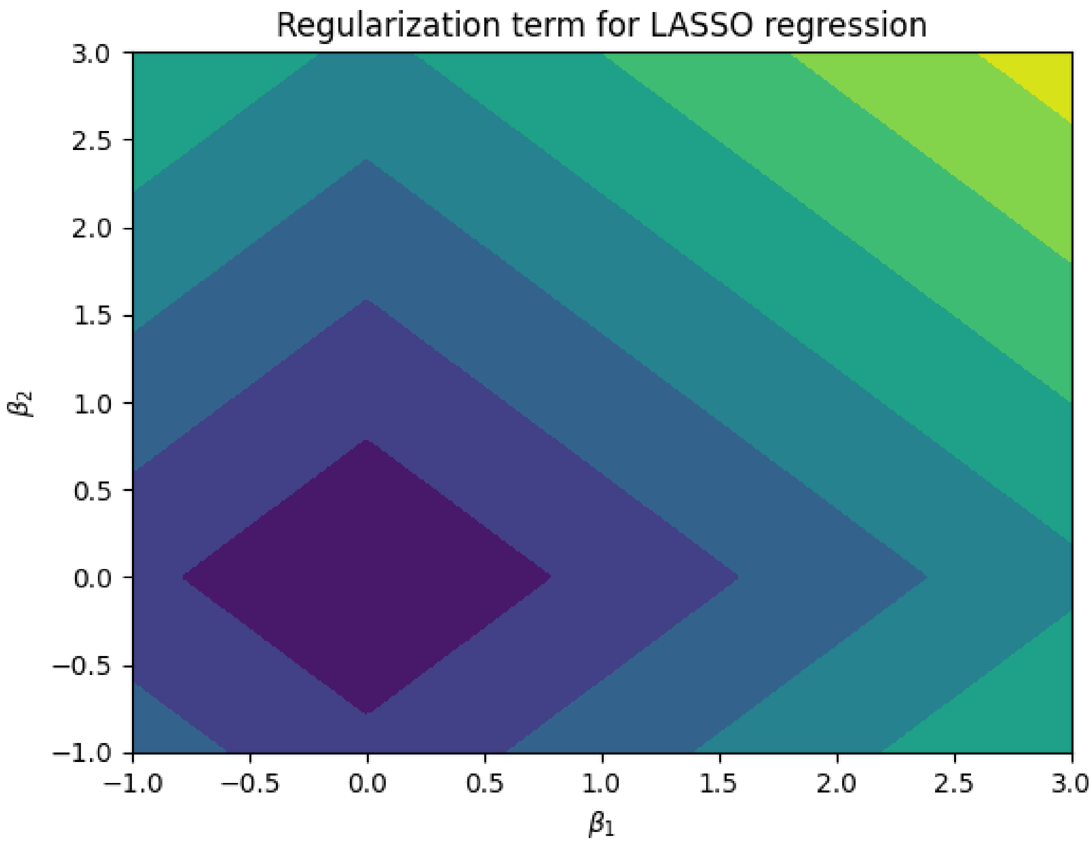
(a)

[7]:
```python
# TODO: make coutour plots for ridge and lasso regularization terms
def regulariz_ridge(beta1, beta2): # essentially the 2-norm
    return beta1*beta1 + beta2*beta2

def regulariz_lasso(beta1, beta2): # essentially the 1-norm
```

```
    return np.abs(beta1)+np.abs(beta2)


plt.title('Regularization term for ridge regression')
plt.xlabel(r'$\beta_1$')
plt.ylabel(r'$\beta_2$')
plt.contourf(beta1, beta2, regulariz_ridge(beta1,beta2))
plt.show()

plt.title('Regularization term for LASSO regression')
plt.xlabel(r'$\beta_1$')
plt.ylabel(r'$\beta_2$')
plt.contourf(beta1, beta2, regulariz_lasso(beta1,beta2))
```

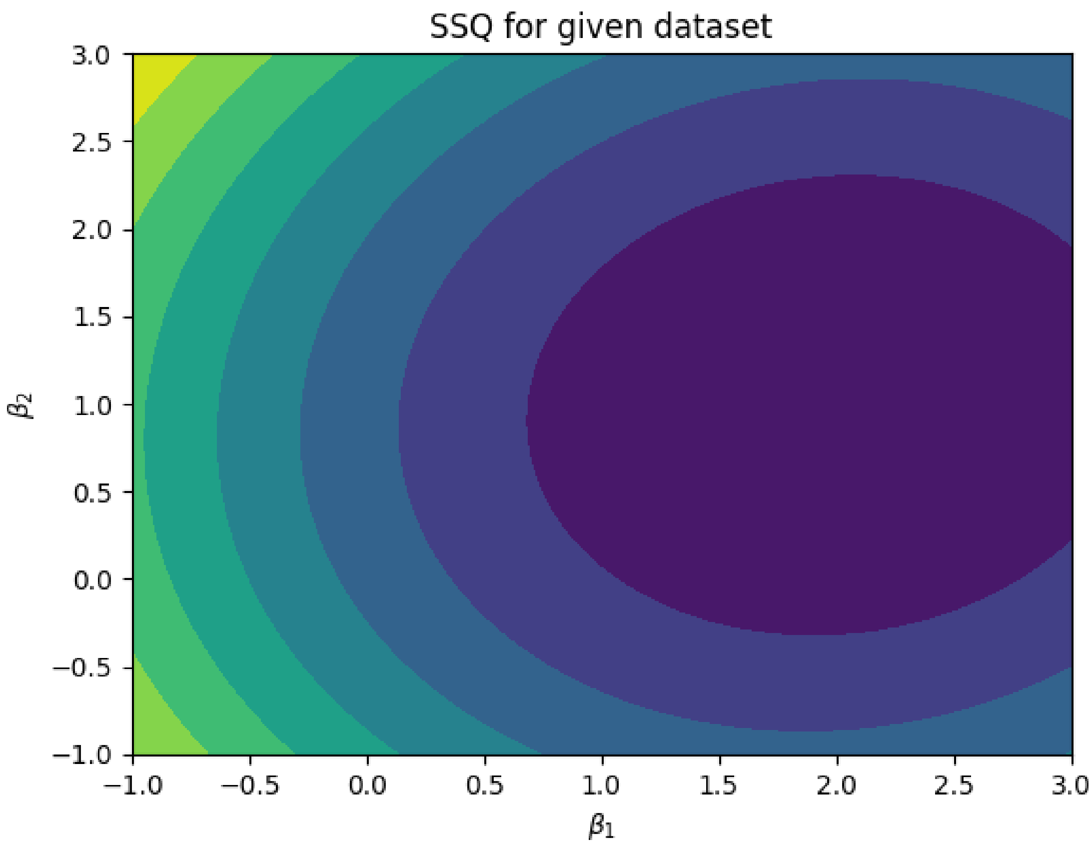


[7]: <matplotlib.contour.QuadContourSet at 0x7f2116bd3a00>



[8]:
```
# TODO: for each combination of parameters, compute the sum of squared␣
 ↪deviations.
#       do not use loops, but numpy broadcasting!
# TODO: make a coutour plot for sum of squared deviations

# the stuff below uses loops, can't figure out how to do it with broadcasting :/
def SSQ(beta1,beta2,features, labels):
    z = np.zeros((beta1.shape[0], beta2.shape[0]))
    for i,b1 in enumerate(beta1):
        for j,b2 in enumerate(beta2):
            beta = [b1, b2]
            z[i,j] = np.sum((beta @ features - labels)**2)
    return z

#print(np.outer(beta1,x[0,:]).shape, beta1.shape, x[0,:].shape)
#print(beta1)
#print(beta1.shape, beta2.shape, z.shape)
beta1 = np.linspace(-1,3,150)
beta2 = np.linspace(-1,3,150)
z = SSQ(beta1, beta2, x, y)
```

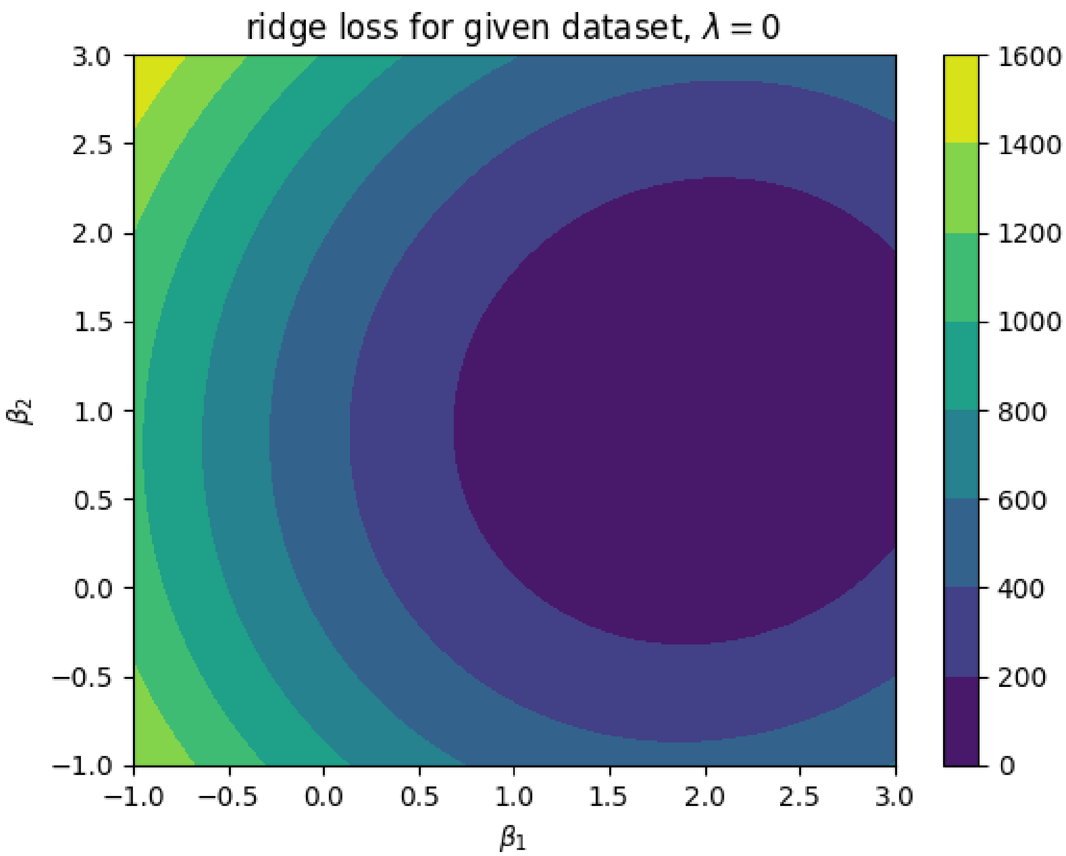
```
plt.contourf(beta1,beta2,z)
plt.title('SSQ for given dataset')
plt.xlabel(r'$\beta_1$')
plt.ylabel(r'$\beta_2$')
plt.show()
```

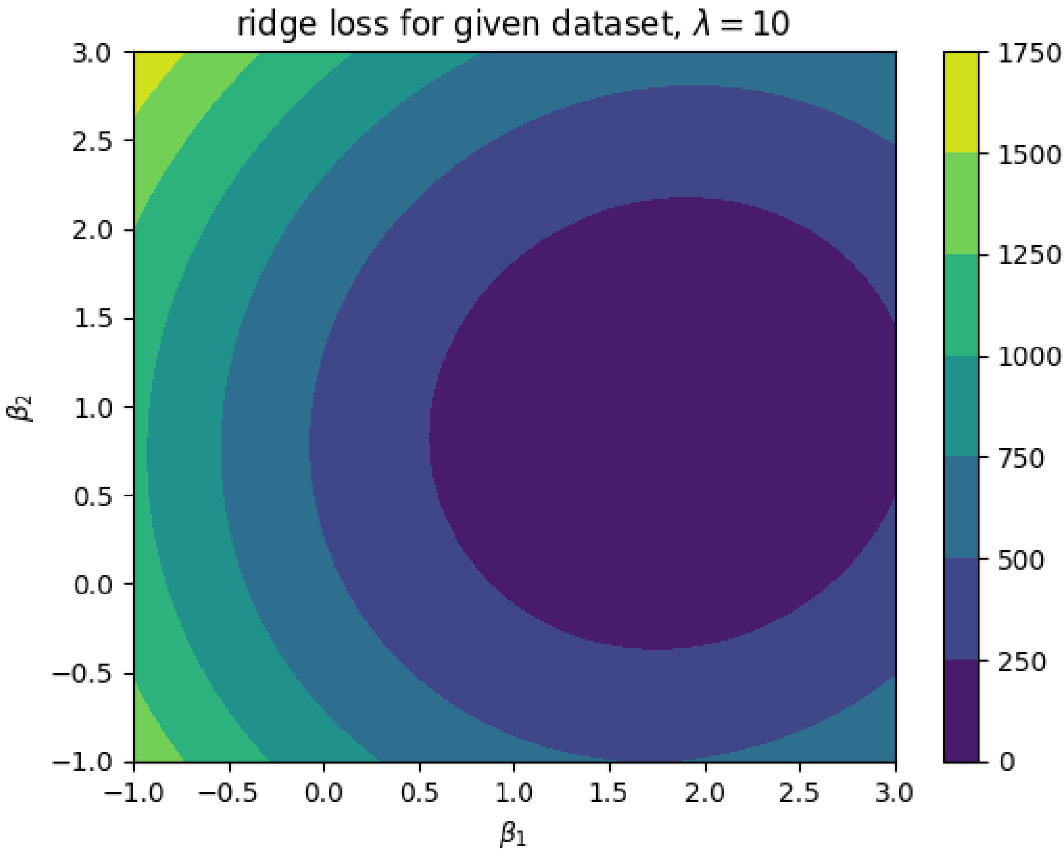


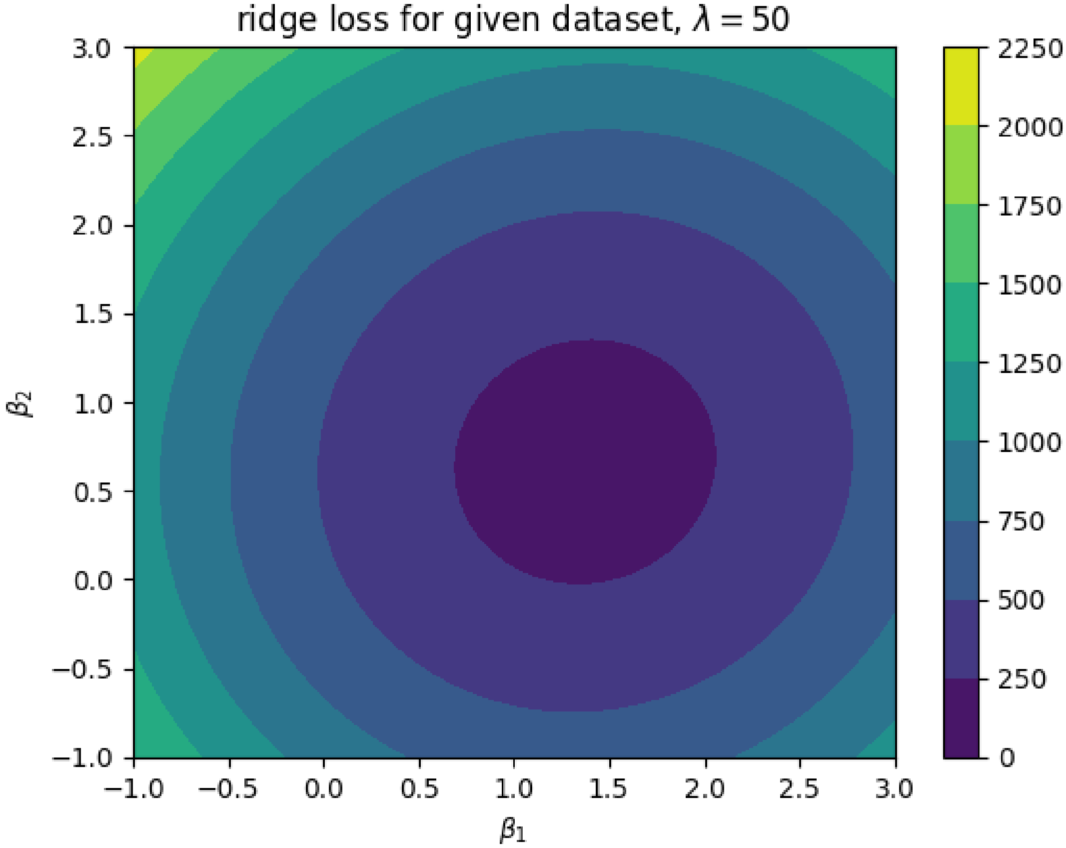
(c)

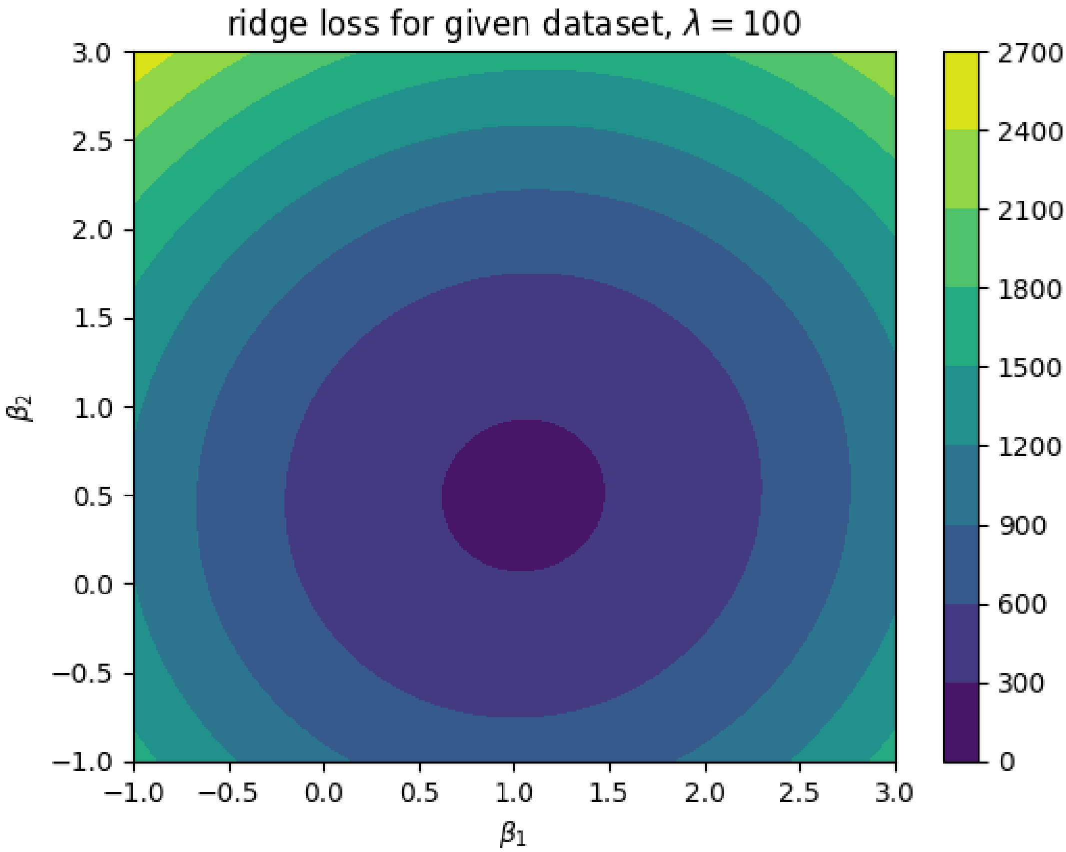
[9]:
```
# TODO: for each lambda, plot both ridge regression and lasso loss functions
def loss(beta1, beta2, lam, features, labels, method='ridge'):
    mbeta1, mbeta2 = np.meshgrid(beta1, beta2) # use meshgrid-functions for␣
 ↪regulariz.terms
    regulariz = np.zeros((beta1.shape[0], beta2.shape[0]))
    if method=='ridge':
        regulariz = regulariz_ridge(mbeta1, mbeta2)
    elif method=='lasso':
        regulariz = regulariz_lasso(mbeta1, mbeta2)
    else:
```

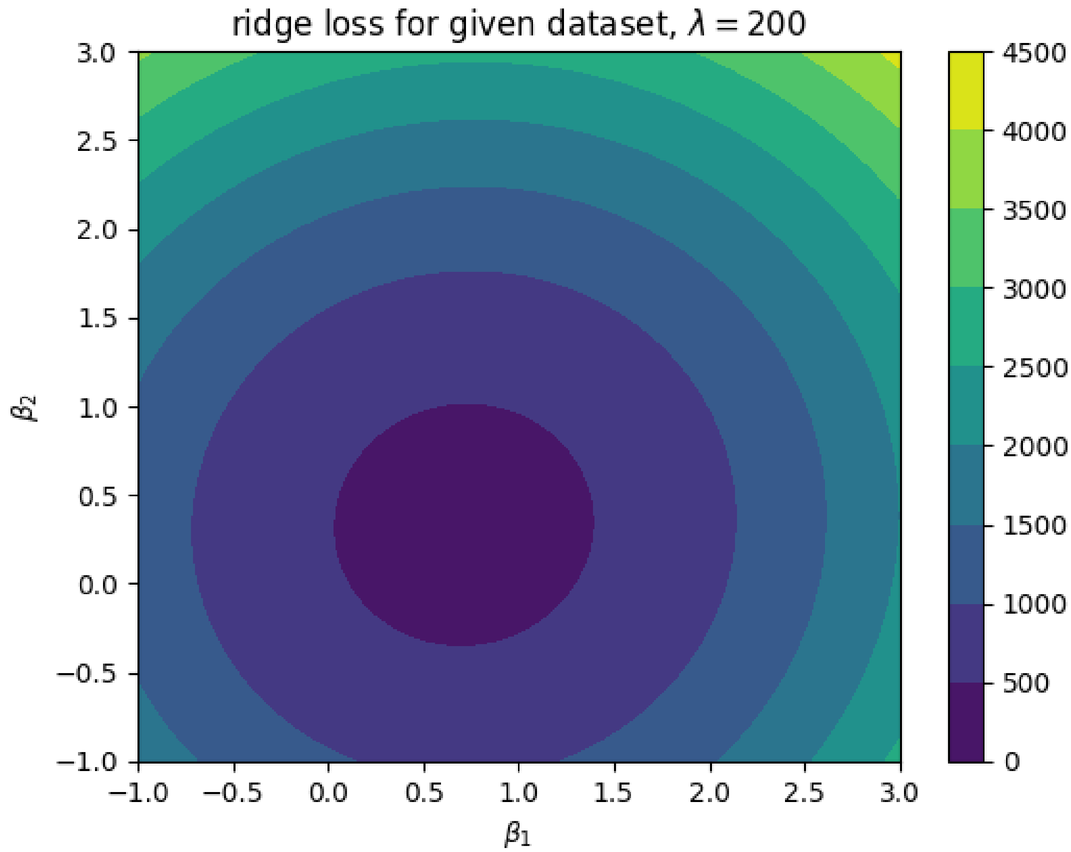
```
        raise NameError(f'Name {method} is not a specified regularization␣
↪method :(')
    return SSQ(beta1, beta2, features, labels) + lam*regulariz

for meth in ['ridge', 'lasso']: # lets _speed_ this up with some _meth_s
    for lam in [0, 10, 50, 100, 200, 300]:
        lss = loss(beta1, beta2, lam, x, y, method=meth)
        plt.contourf(beta1, beta2, lss)
        plt.colorbar()
        plt.title(f'{meth} loss for given dataset, $\lambda={lam}$')
        plt.xlabel(r'$\beta_1$')
        plt.ylabel(r'$\beta_2$')
        plt.show()
```

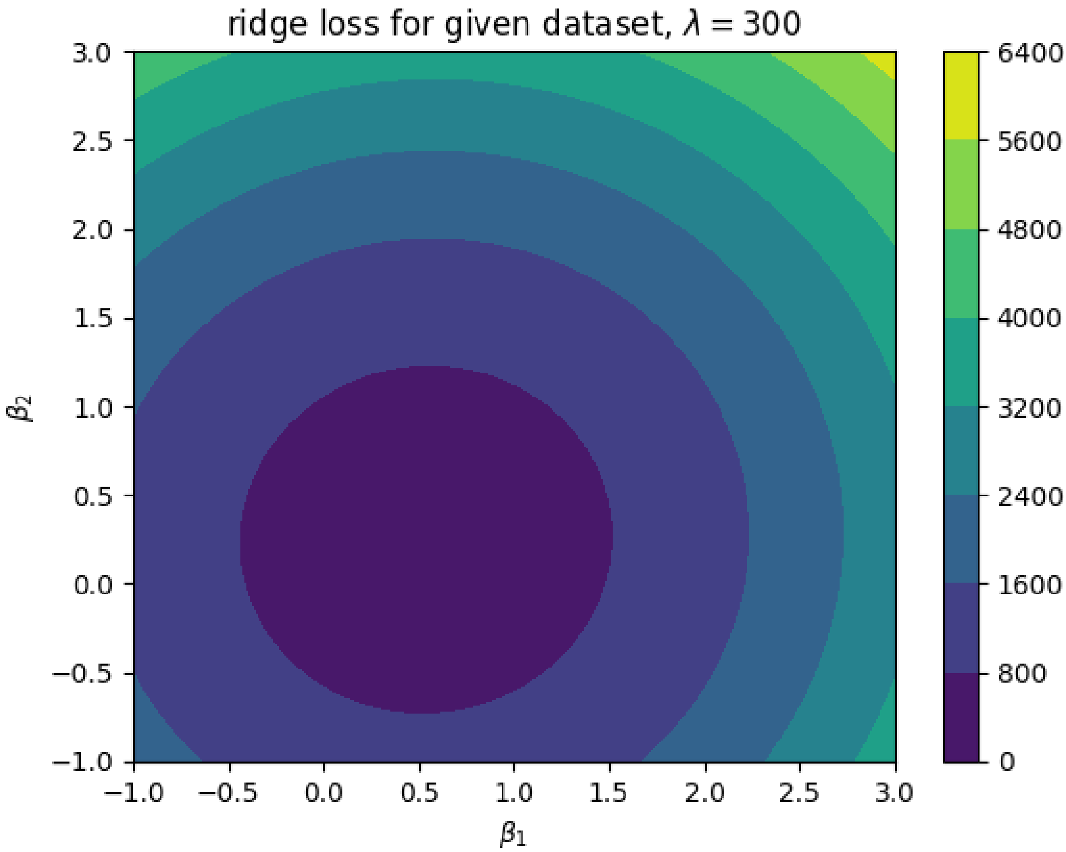



ridge loss for given dataset, $\lambda = 10$
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
β_2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
β_1
1750
1500
1250
1000
750
500
250
0

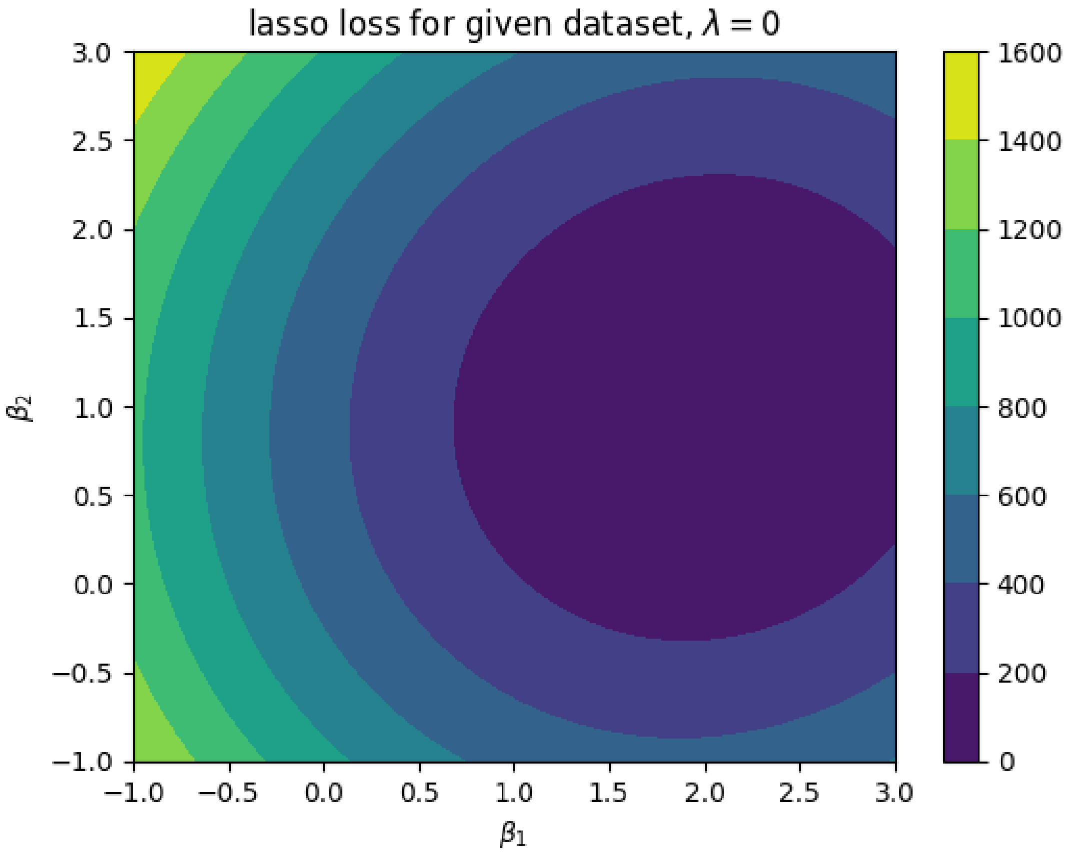

ridge loss for given dataset, $\lambda = 50$
β_1
β_2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
250
500
750
1000
1250
1500
1750
2000
2250

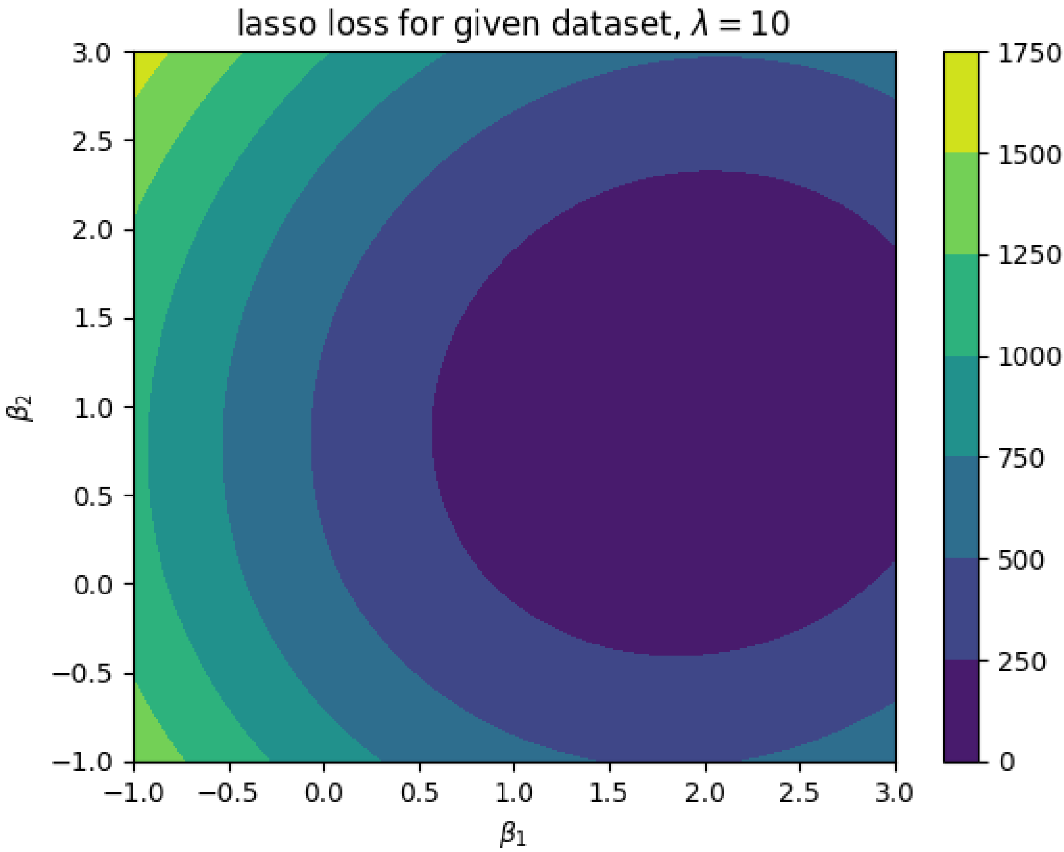

ridge loss for given dataset, $\lambda = 100$
β_1
β_2
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
2700
2400
2100
1800
1500
1200
900
600
300
0


ridge loss for given dataset, $\lambda = 200$
β_1
β_2
-1.0
-0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
500
1000
1500
2000
2500
3000
3500
4000
4500

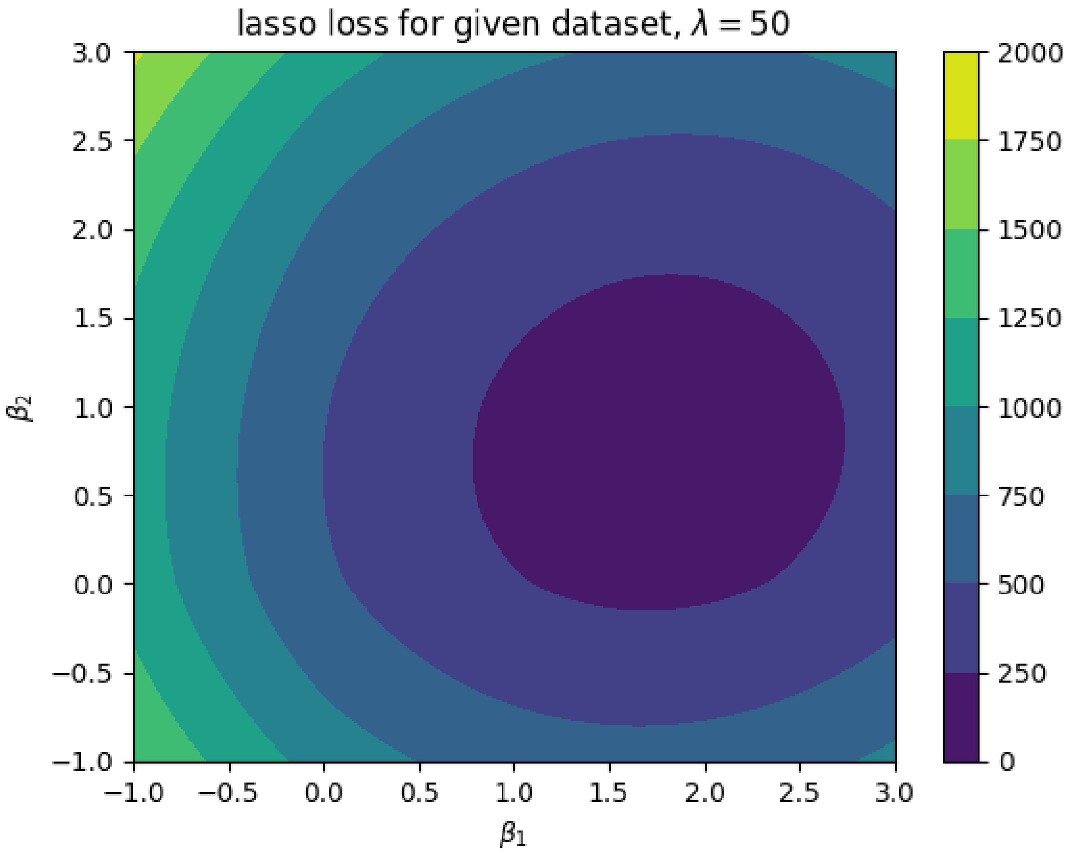


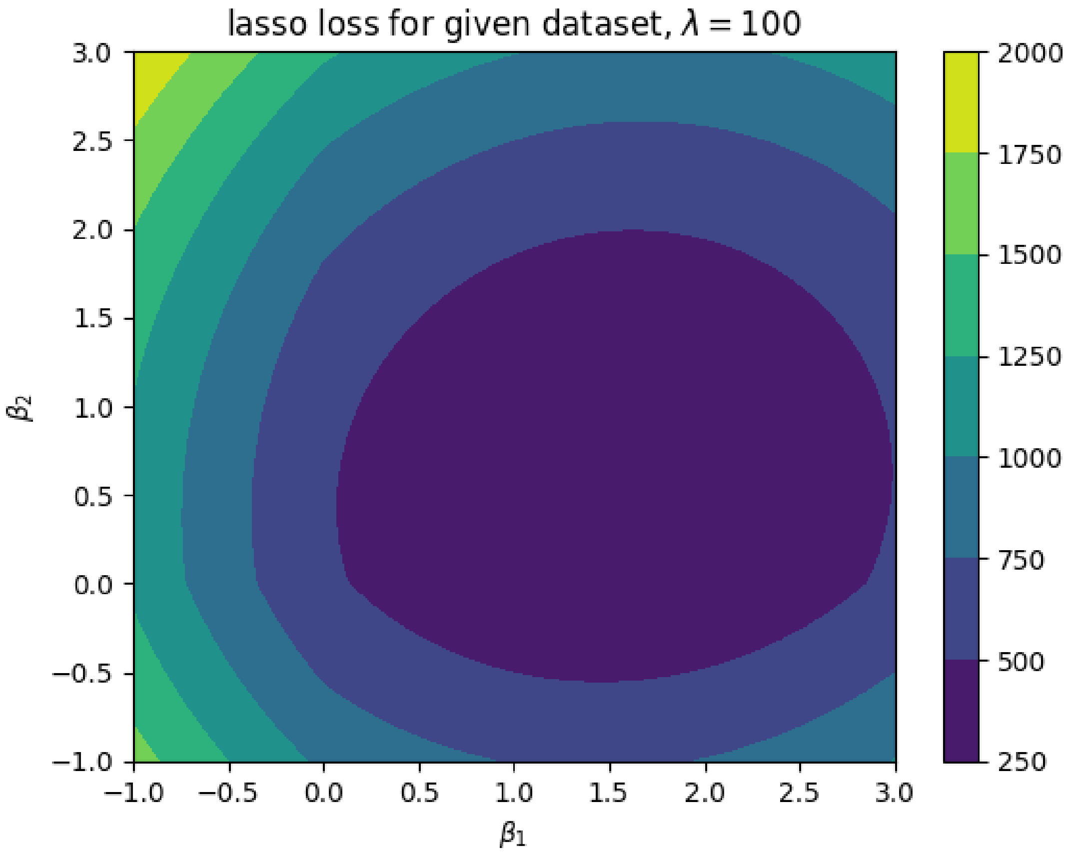
ridge loss for given dataset, $\lambda = 300$
β_1
β_2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
800
1600
2400
3200
4000
4800
5600
6400


lasso loss for given dataset, $\lambda = 0$
β_1
β_2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
200
400
600
800
1000
1200
1400
1600

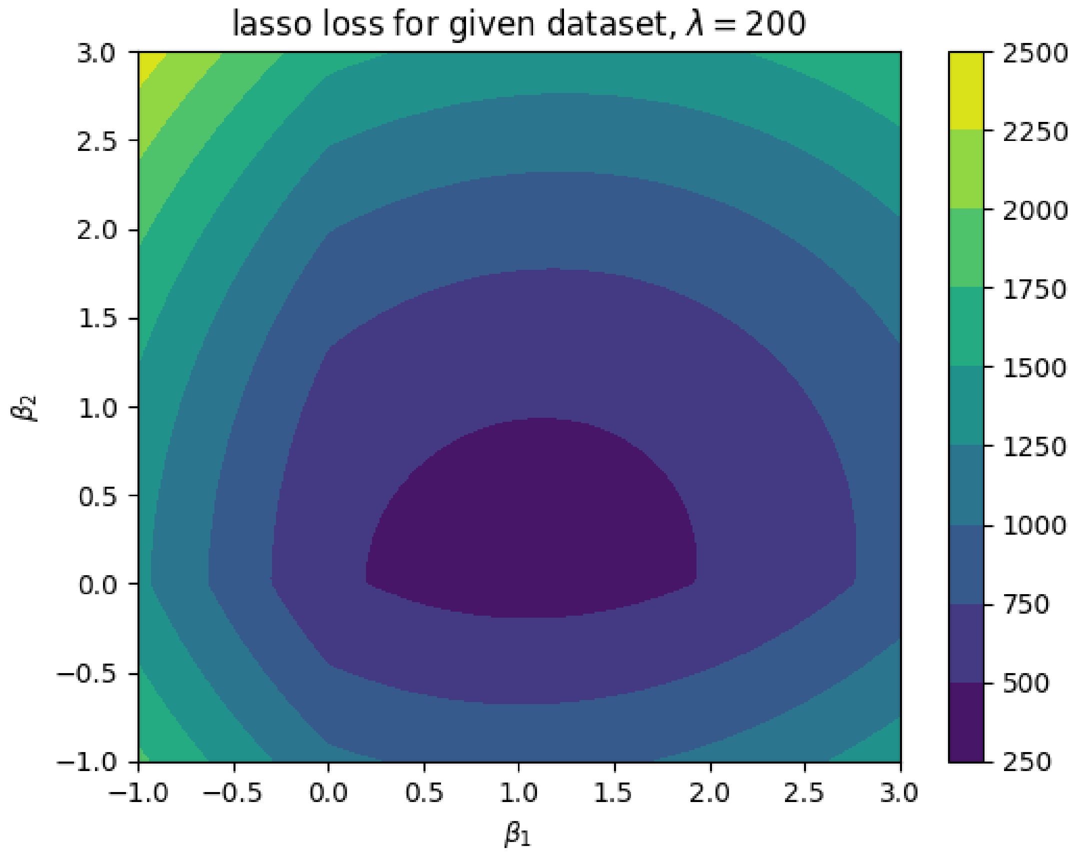


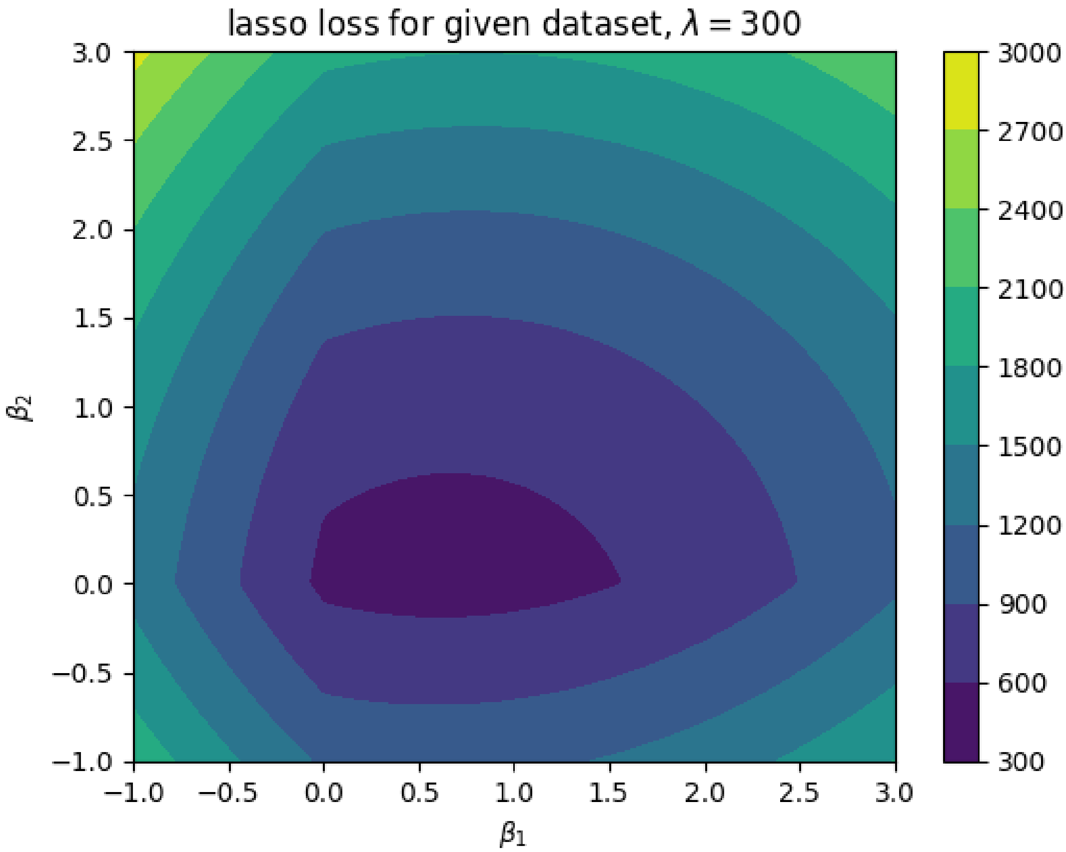
lasso loss for given dataset, $\lambda = 10$
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
β_2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
β_1
1750
1500
1250
1000
750
500
250
0


lasso loss for given dataset, $\lambda = 50$
β_1
β_2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
250
500
750
1000
1250
1500
1750
2000



lasso loss for given dataset, $\lambda = 100$
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
β_2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
β_1
2000
1750
1500
1250
1000
750
500
250


lasso loss for given dataset, $\lambda = 200$
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
β_2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
β_1
2500
2250
2000
1750
1500
1250
1000
750
500
250



Discussion: We observe the following:

- For higher values of λ, we see that the overall magnitude of losses increases (obviously), hence the gradient of the loss surfaces is increased as well.
- For higher values of λ, the shape of the loss surface stays roughly the same for ridge regression: It is transformed from a elliptical shape to a more circular shape, i.e. the eccentricity of the contour lines shrinks.
- For ridge regression, higher values of λ distort the contour lines towards a rhombus: The loss surfaces are just a superposition of the (elliptical) SSQ and the (rhomboidal) lasso loss.

1.5 5 CT reconstruction

set up design matrix (run this once to save to disk)

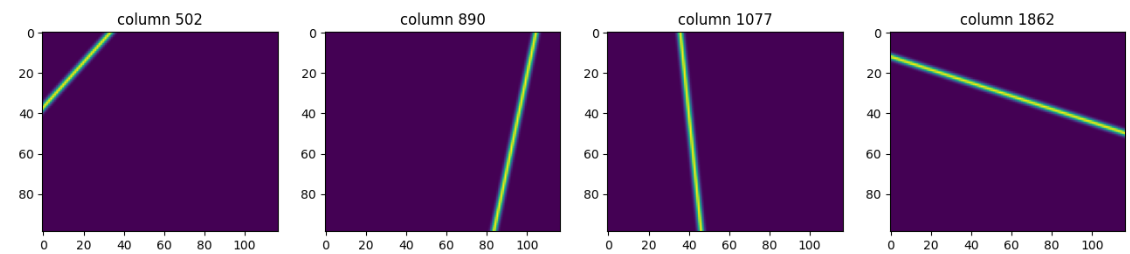
[10]:
```
# create design matrix
# don't change any of this, just run it once to create and save the design␣
↪matrix
import os

if not os.path.exists('data/design_matrix.npy'):
```

```
res = (99, 117)
xs = np.arange(0, res[1]+1) - res[1]/2 # np.linspace(-1, 1, res[1] + 1)
ys = np.arange(0, res[0]+1) - res[0]/2 #np.linspace(-1, 1, res[0] + 1)

# rays are defined by origin and direction
n_parallel_rays = 70
ray_offset_range = [-res[1]/1.5, res[1]/1.5]
n_ray_angles = 30
n_rays = n_parallel_rays * n_ray_angles

ray_angles = np.linspace(0, np.pi, n_ray_angles, endpoint=False) + np.pi/n_ray_angles

# offsets for ray_angle = 0, i.e. parallel to x-axis
ray_0_offsets = np.stack([np.zeros(n_parallel_rays), np.linspace(*ray_offset_range, n_parallel_rays)], axis=-1)
ray_0_directions = np.stack([np.ones(n_parallel_rays), np.zeros(n_parallel_rays)], axis=-1)

def rot_mat(angle):
    c, s = np.cos(angle), np.sin(angle)
    return np.stack([np.stack([c, s], axis=-1), np.stack([-s, c], axis=-1)], axis=-1)

ray_rot_mats = rot_mat(ray_angles)

ray_offsets = np.einsum('oi,aij->aoj', ray_0_offsets, ray_rot_mats).reshape(-1, 2)
ray_directions = np.einsum('oi,aij->aoj', ray_0_directions, ray_rot_mats).reshape(-1, 2)

sigma = 1
kernel = lambda x: np.exp(-x**2/sigma**2/2)

xsc = (xs[1:] + xs[:-1]) / 2
ysc = (ys[1:] + ys[:-1]) / 2
b = np.stack(np.meshgrid(xsc, ysc), axis=-1).reshape(-1, 2)
a = ray_offsets
v = ray_directions
v = v / np.linalg.norm(v, axis=-1, keepdims=True)
p = ((b[None] - a[:, None]) * v[:, None]).sum(-1, keepdims=True) * v[:, None] + a[:, None]
d = np.linalg.norm(b - p, axis=-1)
d = kernel(d)
design_matrix = d.T
```

```
    np.save('data/design_matrix.npy', design_matrix)
    print(f'created and saved design matrix of shape {design_matrix.shape} at␣
↪data/design_matrix.npy')
```

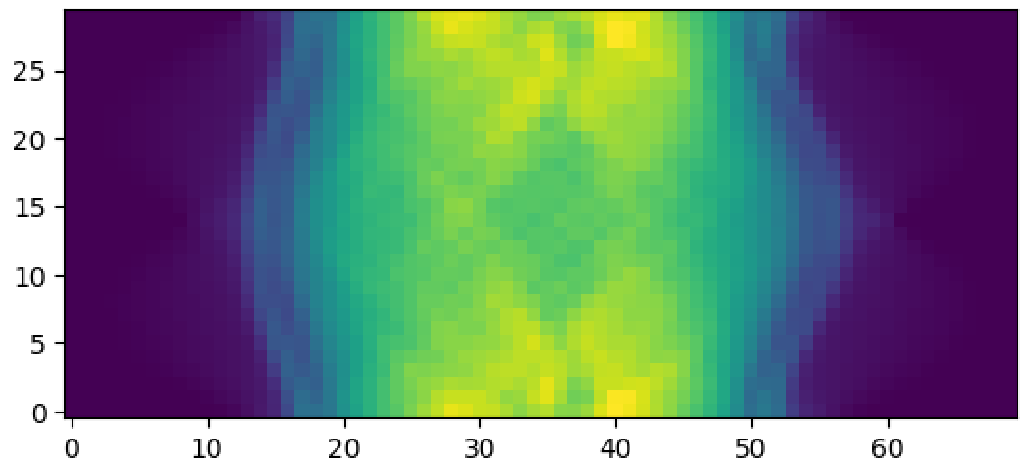
(a)

[11]:
```
design_matrix = np.load('data/design_matrix.npy')
res = (99, 117)

# TODO: visualize four random columns as images, using an image shape of (99,␣
↪117)
img_shape = (99, 117)

fig, axs = plt.subplots(1, 4, figsize=(16, 4))
for i, ax in zip(np.random.choice(np.arange(design_matrix.shape[1]), 4), axs):
    ax.imshow(design_matrix[:, i].reshape(*res));
    ax.set_title(f'column {i}')
```



Interpretation of a column of X: Every column of X describes the axis along which the detectors are positioned relative to the object to be reconstructed. Weighing the original object with a column of X gives the part of the object which can be projected onto the detector, the integral over this then gives the detector signal strength.

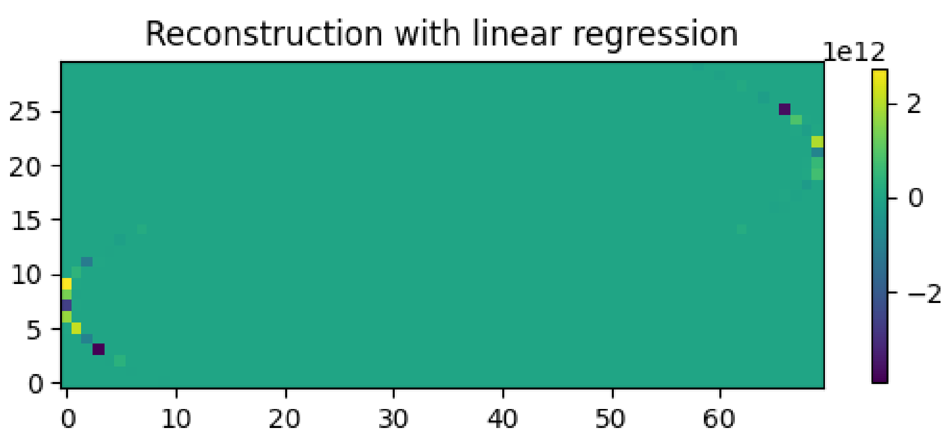
[12]:
```
sino = np.load('data/sino.npy')

# visualize sinogram as image
n_parallel_rays = 70
n_angles = 30
plt.imshow(sino.reshape(n_angles, n_parallel_rays), origin='lower')
plt.show();
```



(b)

[13]:
```
# TODO: solve the reconstruction with linear regression and visualize the result
X = design_matrix
beta = np.linalg.inv(X @ X.T) @ X @ sino.T
```

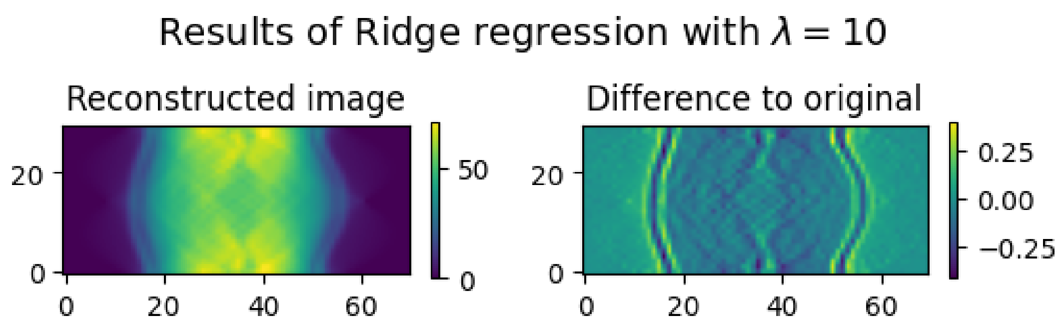
[14]:
```
reconst = beta.T @ X
plt.imshow(reconst.reshape(n_angles, n_parallel_rays), origin='lower')
plt.colorbar(shrink=.44)
plt.title('Reconstruction with linear regression')
plt.show()
```

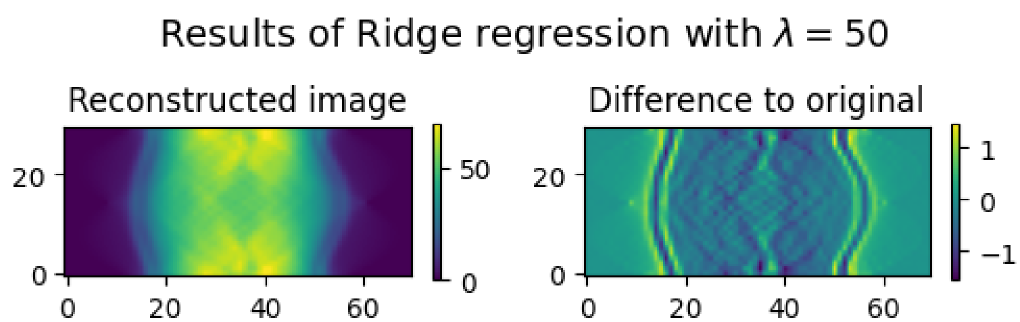


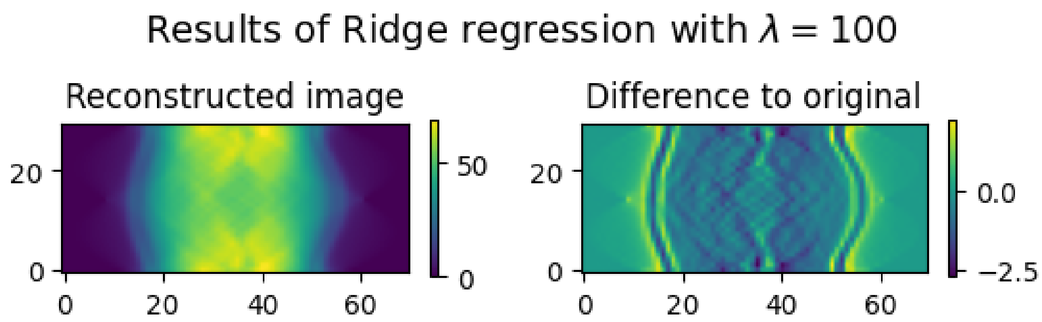
[15]:
```
# TODO: solve the reconstruction with ridge regression and visualize the result
# Optional: try out different regularization strengths and oberve the influence
X = design_matrix
```

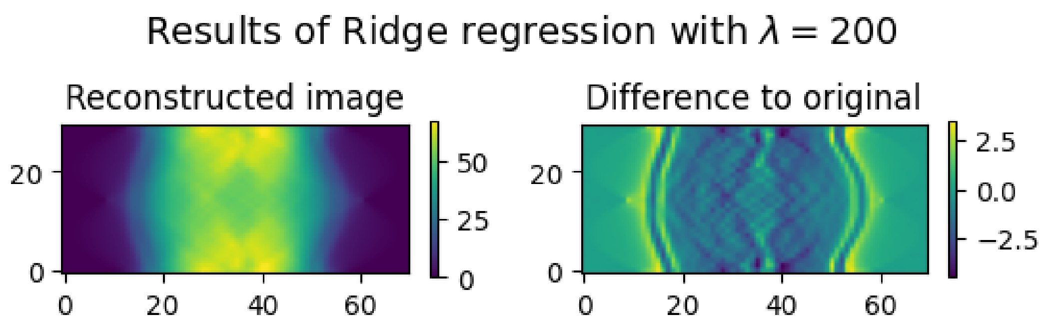
```
#beta_r = np.linalg.inv(X @ X.T + lam*np.identity(X.shape[0])) @ X @ sino.T
```

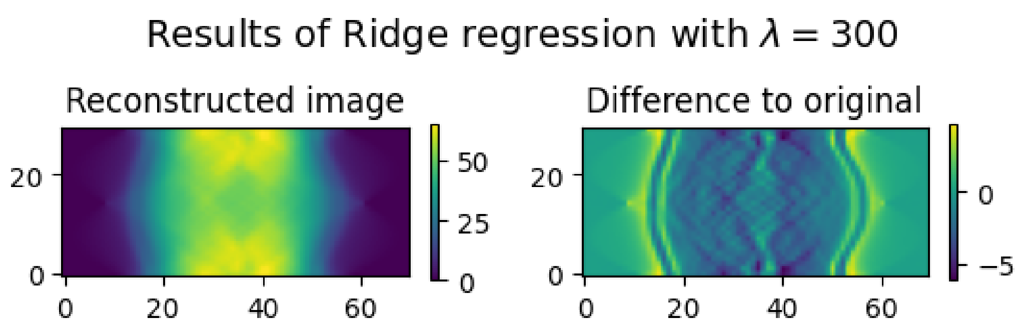
[16]:
```
for lam in [10, 50, 100, 200, 300]:
    beta_r = np.linalg.inv(X @ X.T + lam*np.identity(X.shape[0])) @ X @ sino.T
    reconst = beta_r.T @ X
    plt.subplot(1,2,1)
    plt.title('Reconstructed image')
    plt.imshow(reconst.reshape(n_angles, n_parallel_rays), origin='lower')
    plt.colorbar(shrink=0.22)
    #plt.show()
    plt.subplot(1,2,2)
    plt.title('Difference to original')
    plt.imshow((reconst-sino).reshape(n_angles, n_parallel_rays),␣
 ↪origin='lower')
    plt.colorbar(shrink=0.22)
    plt.suptitle(f"Results of Ridge regression with $\lambda={lam}$", y=0.7,␣
 ↪fontsize=14)
    plt.show()
```











We observe that:

- The plain Linear regression fails phenomenally at the task of reconstruction the image, because of the very inaccurate matrix inversion
- For all values of $\lambda \in \{10, 50, 100, 200, 300\}$, the ridge regression give optically fine results, for the smallest value of $\lambda = 10$, we get the least deviation from the original image.
- The shape of the difference does not vary by a lot, only its magnitude changes meaningfully with λ