Exercises week 41

FYS-STK4155

Even Sletteng Garvang

Introduction

I have made my own implementations of gradient descent (GD) and stochastic gradient descent (SGD) with and without memory and with different adaptive gradient techniques. The exploration of GD and validation of the implementation can be found at https://github.com/evengar/fys-stk4155/blob/main/Project2/w41-exercise.pdf (work-in-progress). Here I focus on applying the techniques to a simple data set with polynomial features using gradients from ordinary least squares (OLS) and Ridge regression.

The functions for gradient descent and utility functions are contained in `gdfuns.py` and `utils.py`, respectively, available at https://github.com/evengar/fys-stk4155/tree/main/Project2

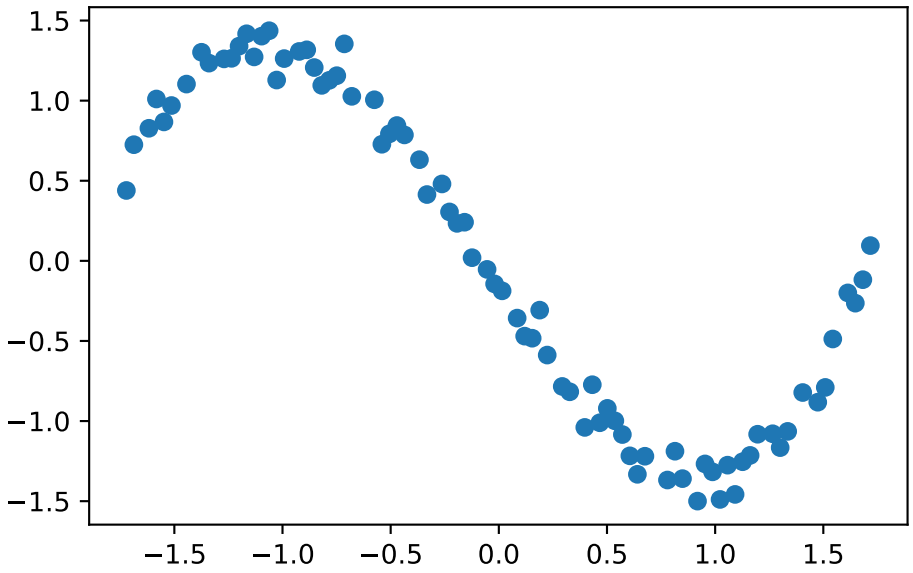
Preparation

Import libraries, gradient descent functions and utility functions.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

from gdfuns import *
from utils import *
```

Generate data.

```
np.random.seed(8923)

# generate data
n = 100
x = np.linspace(-2, 2, n)
f_x = data_function(x)
y = f_x + np.random.normal(0, 1, n)
y = y.reshape(-1,1)

# create design matrix of polynomials
# for now 3rd order reflecting data function
X = PolynomialFeatures(10).fit_transform(x.reshape(-1, 1))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)

scalerx = StandardScaler()
X_train_scaled = scalerx.fit_transform(X_train)
X_test_scaled = scalerx.transform(X_test)

scalery = StandardScaler()
y_train_scaled = scalerx.fit_transform(y_train)
y_test_scaled = scalerx.transform(y_test)


plt.plot(X_train_scaled[:,1], y_train_scaled, "o")
plt.show()
```



Here we have polynomials up to degree 10, while our data is generated from degree 3. We can see how our gradient descent handles different degrees.

GD and SGD

OLS gradient

Polynomial degree

```
def GD_poly_mse(
    X_train, X_test, y_train, y_test,
    maxpoly, minpoly = 1, gd_fun = GD, **kwargs
    ):
    polys = np.arange(minpoly-1, maxpoly) + 1
    mse = np.zeros(len(polys))

    for i in range(len(polys)):
        xtrain, xtest = sub_to_poly(X_train, X_test, polys[i])
        eta = 0.2
        n_iter = 1000
        theta = gd_fun(xtrain, y_train_scaled, **kwargs)
        y_pred = xtest @ theta
```

```
        mse[i] = mean_squared_error(y_test_scaled, y_pred)
    return mse
```

GD and SGD without momentum

```
maxpoly = 10
eta = 0.1
n_iter = 200

# sgd parameters
M = 5
n_epochs = 50

mse_gd = GD_poly_mse(
    X_train_scaled, X_test_scaled, y_train, y_test,
    maxpoly, eta = eta, n_iter = n_iter
    )

mse_sgd = GD_poly_mse(
    X_train_scaled, X_test_scaled, y_train, y_test,
    maxpoly, gd_fun=SGD, eta = eta, n_epochs = n_epochs, M = M
    )

polys = np.arange(maxpoly) + 1
plt.plot(polys, mse_gd, label = f"GD, {n_iter} iterations")
plt.plot(polys, mse_sgd, label = f"SGD, {n_epochs} epochs")
plt.legend()
plt.xlabel("Polynomial degree")
plt.ylabel("MSE")
plt.title(fr"$\eta$ = {eta}")
plt.show()
```

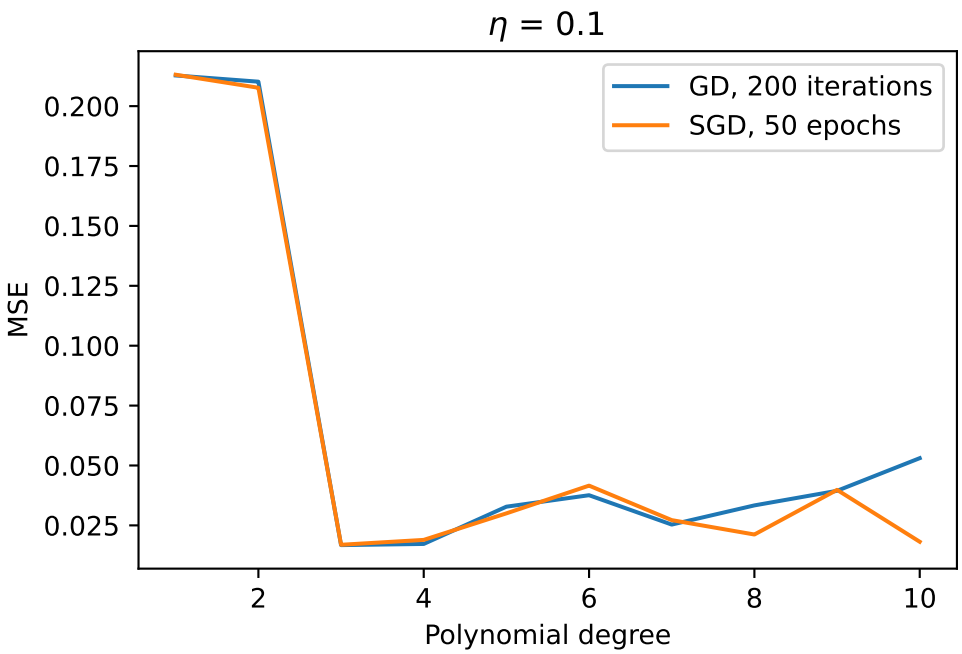


Figure 1

The algorithms give fairly good MSE, but increasing iterations lowers MSE, suggesting that it does not have time to converge in the specified number of iterations. Try with momentum.

```
maxpoly = 10
eta = 0.1
n_iter = 200

# sgd parameters
M = 5
n_epochs = 50

# momentum parameter
gamma = 0.3

mse_gd = GD_poly_mse(
    X_train_scaled, X_test_scaled, y_train, y_test,
    maxpoly, eta = eta, n_iter = n_iter,
    momentum = True, gamma = gamma
    )

mse_sgd = GD_poly_mse(
```

```
    X_train_scaled, X_test_scaled, y_train, y_test,
    maxpoly, gd_fun=SGD, eta = eta, n_epochs = n_epochs, M = M,
    momentum = True, gamma = gamma
    )

polys = np.arange(maxpoly) + 1
plt.plot(polys, mse_gd, label = f"GD, {n_iter} iterations")
plt.plot(polys, mse_sgd, label = f"SGD, {n_epochs} epochs")
plt.legend()
plt.xlabel("Polynomial degree")
plt.ylabel("MSE")
plt.title(fr"$\eta$ = {eta}")
plt.show()
```

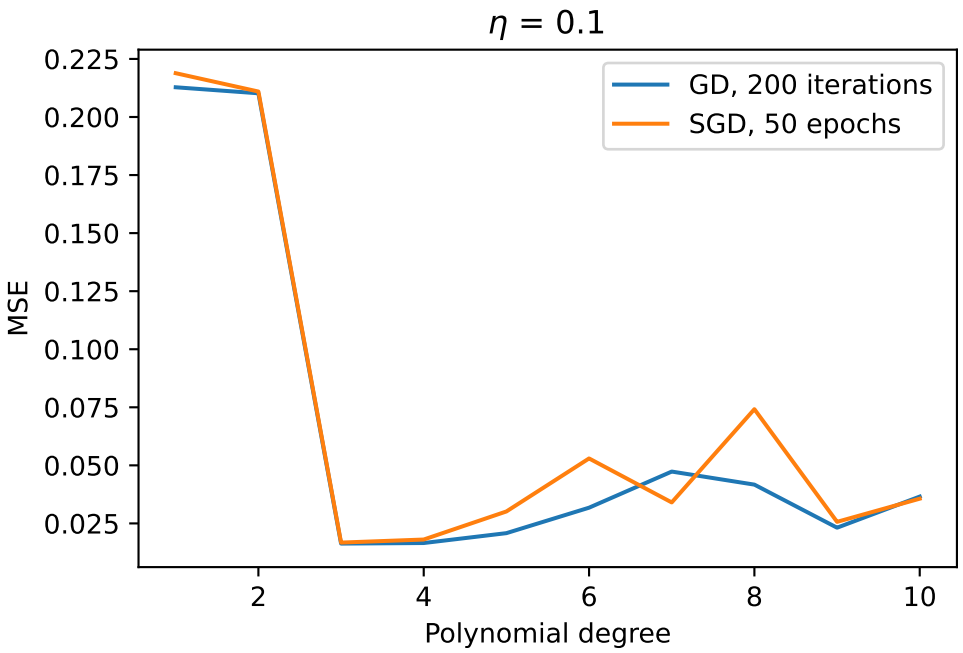


Figure 2

Learning rates

```
def eta_poly_grid(
    X_train, X_test, y_train, y_test,
```

```
    etas, maxpoly, minpoly = 1, **kwargs
    ):
    npoly = maxpoly - (minpoly - 1)
    mse = np.zeros( (npoly, len(etas)) )
    for i in range(len(etas)):
        mse_eta = GD_poly_mse(
            X_train_scaled, X_test_scaled, y_train, y_test,
            maxpoly, minpoly=minpoly, eta = etas[i], **kwargs)
        mse[:,i] = mse_eta
    return mse

def find_minimum(mat, varx, vary):
    min_index = np.unravel_index(mat.argmin(), mse.shape)
    varx_min = varx[min_index[1]]
    vary_min = vary[min_in]
    print("Optimal eta:", etas[min_index[1]])
    print("Optimal polydegree:", polys[min_index[0]])
    print("Gives MSE:", mse[min_index])
```

Keeping iterations the same as before, we see how learning rates affect MSE. I do it only with momentum here to keep number of figures down.

```
etas = np.linspace(0.05, 0.2, 20)
maxpoly = 10
minpoly = 3
polys = np.arange(minpoly-1, maxpoly) + 1
n_iter = 200

# momentum parameter
gamma = 0.3

mse = eta_poly_grid(
    X_train_scaled, X_test_scaled, y_train, y_test,
    etas, maxpoly, n_iter = n_iter, minpoly=minpoly,
    momentum = True, gamma = gamma
    )
```

```
sns.heatmap(mse)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(polys)) + 0.5, polys)
plt.xlabel(r"Learning rate $\eta$")
```

```
plt.ylabel("Maximum polydegree")
plt.show()
```

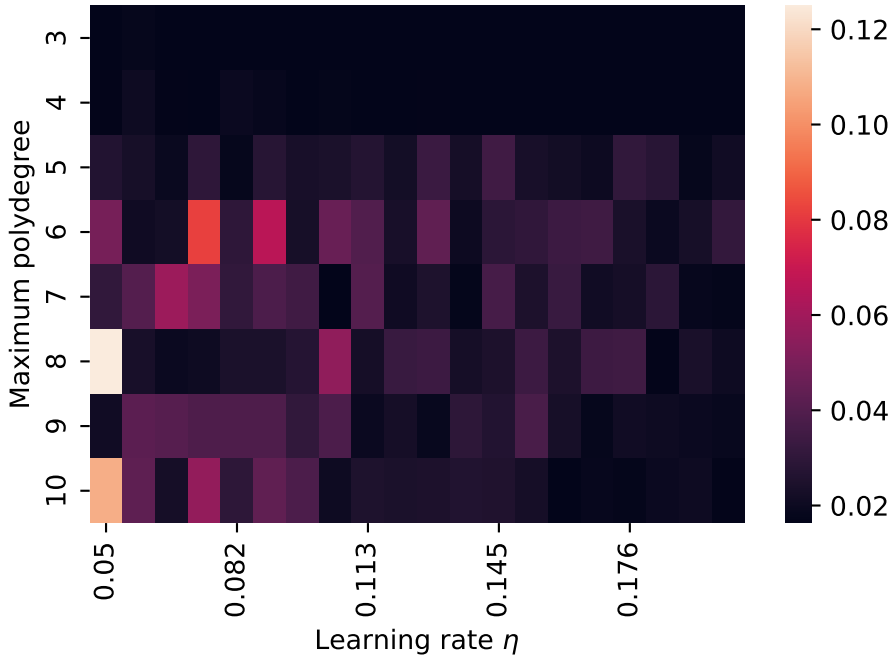


Figure 3: Heatmap of MSE for gradient descent (200 iterations) with varying learning rates and polynomial degrees.

```
min_index = np.unravel_index(mse.argmin(), mse.shape)
print("Optimal eta:", etas[min_index[1]])
print("Optimal polydegree:", polys[min_index[0]])
print("Gives MSE:", mse[min_index])
```

```
Optimal eta: 0.09736842105263159
Optimal polydegree: 3
Gives MSE: 0.016309067557554822
```

The same, but with SGD:

```
etas = np.linspace(0.05, 0.2, 20)
maxpoly = 10
minpoly = 3
```

```
polys = np.arange(minpoly-1, maxpoly) + 1

# sgd parameters
M = 5
n_epochs = 50

# momentum parameter
gamma = 0.3

mse = eta_poly_grid(
    X_train_scaled, X_test_scaled, y_train, y_test,
    etas, maxpoly, gd_fun=SGD, n_epochs = n_epochs, M=M, minpoly=minpoly,
    momentum = True, gamma = gamma
    )
```

```
sns.heatmap(mse)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(polys)) + 0.5, polys)
plt.xlabel(r"Learning rate $\eta$")
plt.ylabel("Maximum polydegree")
plt.show()
```

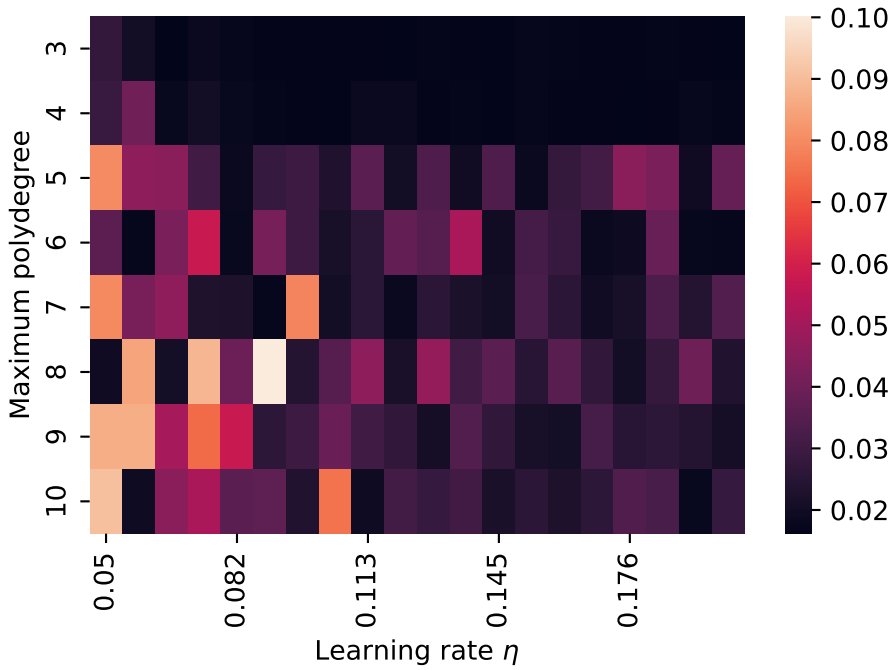


Figure 4: Heatmap of MSE for stochastic gradient descent (M=5, 50 epochs) with varying learning rates and polynomial degrees.

```
min_index = np.unravel_index(mse.argmin(), mse.shape)
print("Optimal eta:", etas[min_index[1]])
print("Optimal polydegree:", polys[min_index[0]])
print("Gives MSE:", mse[min_index])
```

```
Optimal eta: 0.2
Optimal polydegree: 3
Gives MSE: 0.016152895902791366
```

GD and SGD performs similarly for this data and the OLS gradient.

Ridge gradient

We repeat some of the steps above, but with gradient from the Ridge cost function. We settle for a constant polynomial degree of 10, and instead look at the interaction of learning rate and λ. We also do all gradient descents in this part with momentum.

```
def GD_lambda_mse(
    X_train, X_test, y_train, y_test, lmbs,
    gd_fun = GD, gradient_fun=gradient_ridge, **kwargs
    ):
    mse=np.zeros(len(lmbs))
    for i in range(len(lmbs)):
        theta = gd_fun(
            X_train, y_train, gradient_fun=gradient_ridge,
            gradient_args={"lmb":lmbs[i]}, **kwargs
            )
        y_pred = X_test @ theta
        mse[i] = mean_squared_error(y_test, y_pred)
    return mse
```

```
maxpoly = 10
xtrain, xtest = sub_to_poly(X_train_scaled, X_test_scaled, maxpoly)

eta = 0.1
n_iter = 200
lmbs=np.logspace(-12,0,20)

# sgd parameters
M = 5
n_epochs = 50

# momentum parameter
gamma = 0.3

mse_gd_ridge = GD_lambda_mse(
    xtrain, xtest, y_train_scaled, y_test_scaled, lmbs, eta = eta,
    n_iter = n_iter, momentum = True, gamma = gamma
    )

mse_sgd_ridge = GD_lambda_mse(
    xtrain, xtest, y_train_scaled, y_test_scaled, lmbs, eta = eta,
    n_epochs=n_epochs, M=M, gd_fun=SGD, momentum = True, gamma = gamma
    )


plt.plot(lmbs, mse_gd_ridge, label = f"GD, {n_iter} iterations")
plt.plot(lmbs, mse_sgd_ridge, label = f"SGD, {n_epochs} epochs")
plt.legend()
```

```
plt.xlabel(r"$\lambda$")
plt.ylabel("MSE")
plt.semilogx()
plt.show()
```

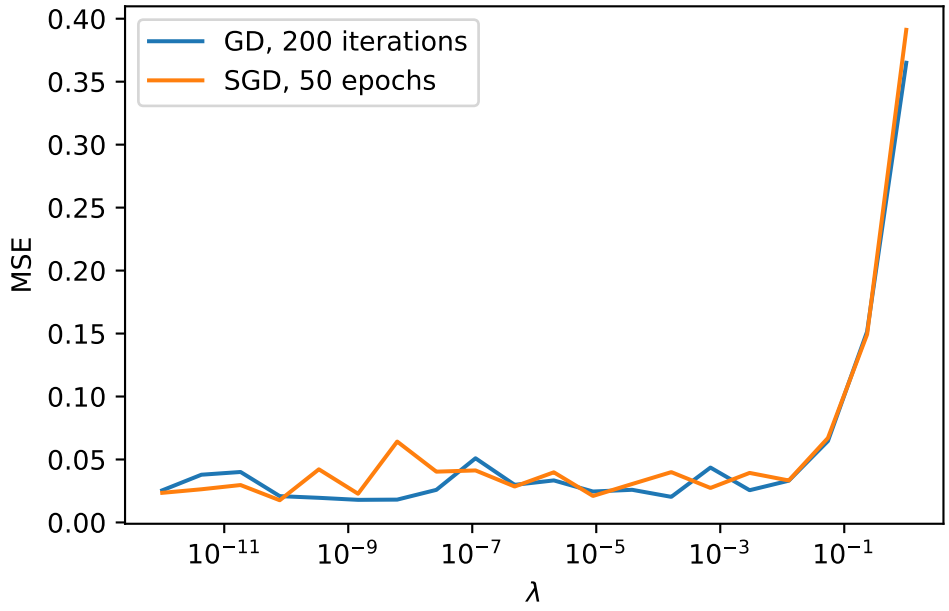


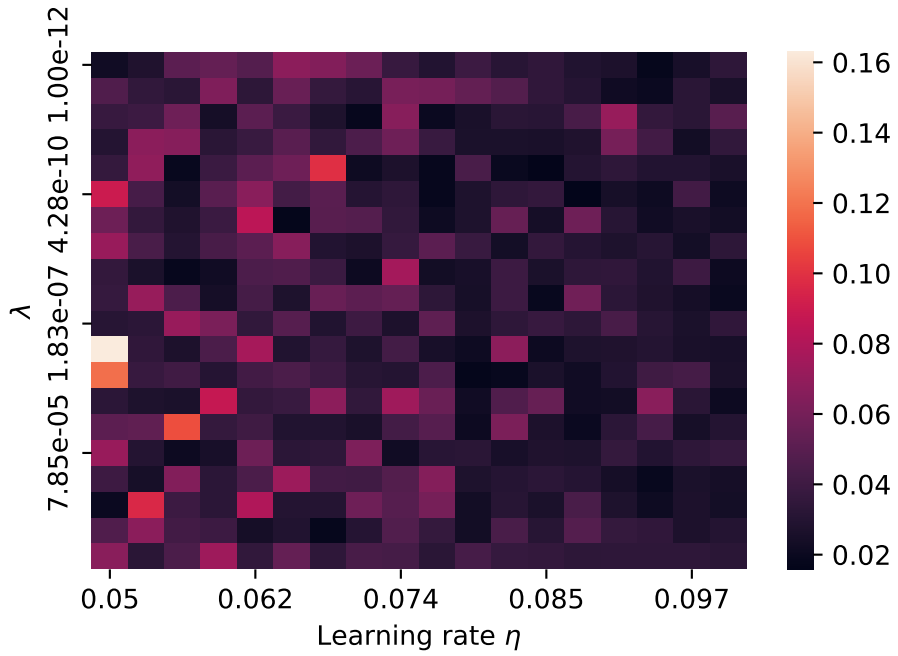
Figure 5: Mean squared error (MSE) from gradient descent (GD) and stochastic gradient descent (SGD) with Ridge gradient. Learning rate $\eta = 0.1$.

No obvious patterns in lambdas, other than that it seems better to keep it below ca. 10^{-3}. Try a grid of lambdas and learning rates:

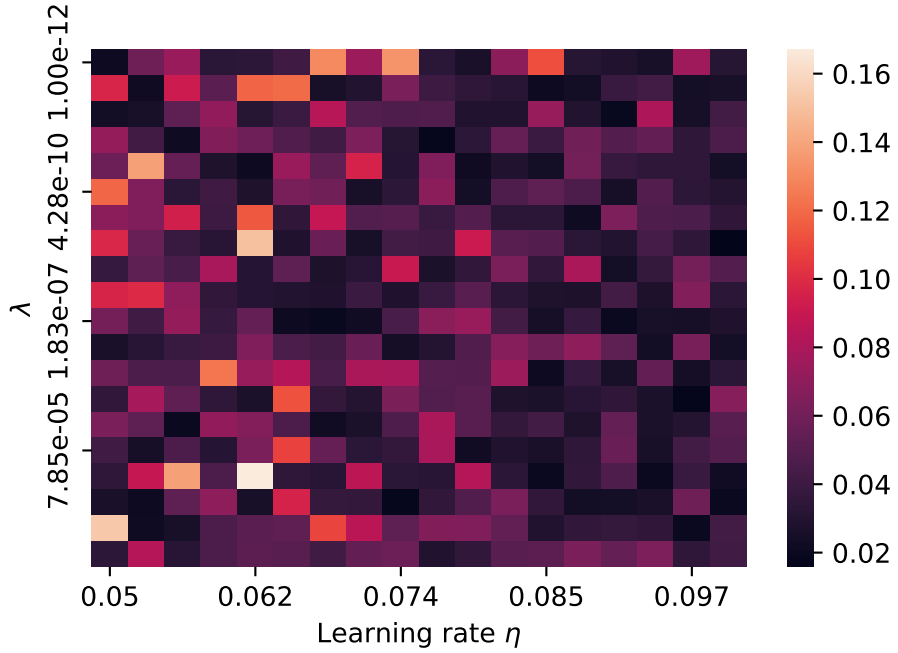
```
def eta_lambda_grid(
    X_train, X_test, y_train, y_test,
    etas, lmbs, **kwargs
    ):
    mse = np.zeros( (len(lmbs), len(etas)) )
    for i in range(len(etas)):
        mse_eta = GD_lambda_mse(
            X_train_scaled, X_test_scaled, y_train, y_test,
            lmbs, eta = etas[i], **kwargs
            )
        mse[:,i] = mse_eta
    return mse
```

```
eta = 0.2
n_iter = 200
lmbs=np.logspace(-12,-2,20)
etas = np.linspace(0.05, 0.1, 18)

# sgd parameters
M = 5
n_epochs = 50

# momentum parameter
gamma = 0.3

mse_gd_ridge = eta_lambda_grid(
    xtrain, xtest, y_train_scaled, y_test_scaled,
    etas, lmbs, n_iter = n_iter,
    momentum = True, gamma = gamma
    )

mse_sgd_ridge = eta_lambda_grid(
    xtrain, xtest, y_train_scaled, y_test_scaled,
    etas, lmbs, n_epochs=n_epochs, M=M, gd_fun=SGD,
    momentum = True, gamma = gamma
    )
```

```
lmb_lab = ["{0:.2e}".format(x) for x in lmbs]

sns.heatmap(mse_gd_ridge)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(lmbs))[::5] + 0.5, lmb_lab[::5])
plt.xlabel(r"Learning rate $\eta$")
plt.ylabel(r"$\lambda$")
plt.show()

sns.heatmap(mse_sgd_ridge)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(lmbs))[::5] + 0.5, lmb_lab[::5])
plt.xlabel(r"Learning rate $\eta$")
plt.ylabel(r"$\lambda$")
plt.show()
```



(a) Gradient descent, 200 iterations.



(b) Stochastic gradient descent, $M = 5$, 50 epochs.

Figure 6: Heatmap of MSE for gradient descent with momentum ($\gamma = 0.3$) with varying learning rates and λs.

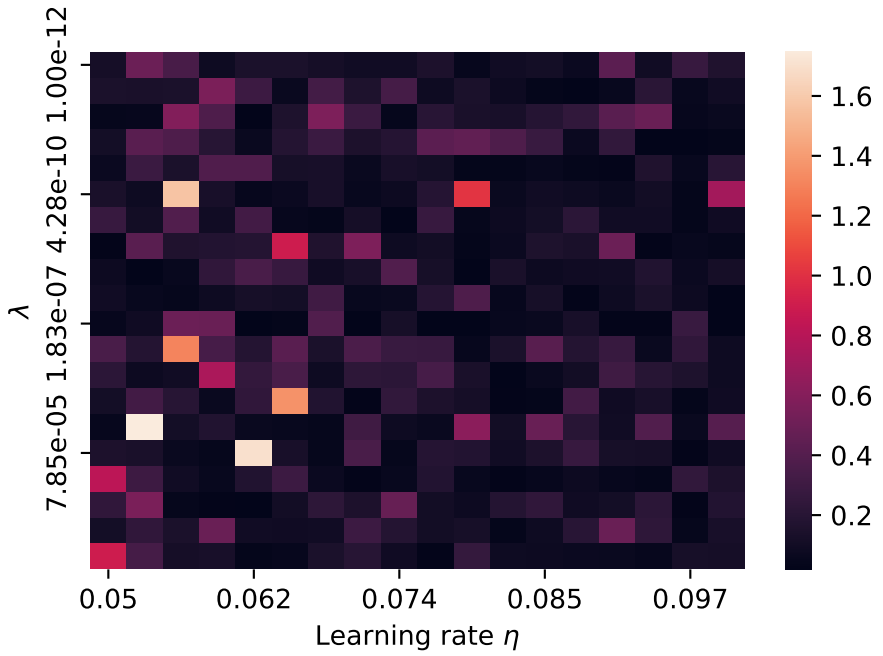
ADAgrad

Same as above, but with ADAgrad (with momentum)

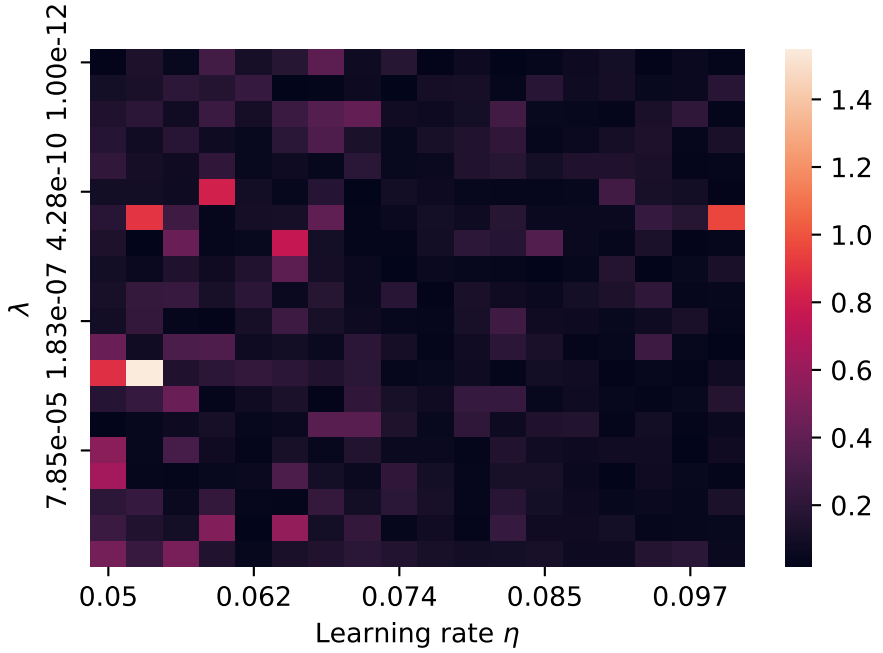
```
eta = 0.2
n_iter = 200
lmbs=np.logspace(-12,-2,20)
etas = np.linspace(0.05, 0.1, 18)

# sgd parameters
M = 5
n_epochs = 50

# momentum parameter
gamma = 0.3

mse_gd_ridge = eta_lambda_grid(
    xtrain, xtest, y_train_scaled, y_test_scaled,
    etas, lmbs, n_iter = n_iter,
    momentum = True, gamma = gamma,
    adaptive_fun=AdaGrad
    )

mse_sgd_ridge = eta_lambda_grid(
    xtrain, xtest, y_train_scaled, y_test_scaled,
    etas, lmbs, n_epochs=n_epochs, M=M, gd_fun=SGD,
    momentum = True, gamma = gamma,
    adaptive_fun=AdaGrad
    )
```

```
lmb_lab = ["{0:.2e}".format(x) for x in lmbs]

sns.heatmap(mse_gd_ridge)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(lmbs))[::5] + 0.5, lmb_lab[::5])
plt.xlabel(r"Learning rate $\eta$")
plt.ylabel(r"$\lambda$")
plt.show()

sns.heatmap(mse_sgd_ridge)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(lmbs))[::5] + 0.5, lmb_lab[::5])
```

```
plt.xlabel(r"Learning rate $\eta$")
plt.ylabel(r"$\lambda$")
plt.show()
```



(a) Gradient descent, 200 iterations.



(b) Stochastic gradient descent, $M = 5$, 50 epochs.

Figure 7: Heatmap of MSE for gradient descent with the AdaGrad algorithm with momentum ($\gamma = 0.3$) with varying learning rates and λs.

Interestingly, SGD performs significantly better than GD for adagrad (for these hyperparameter ranges).

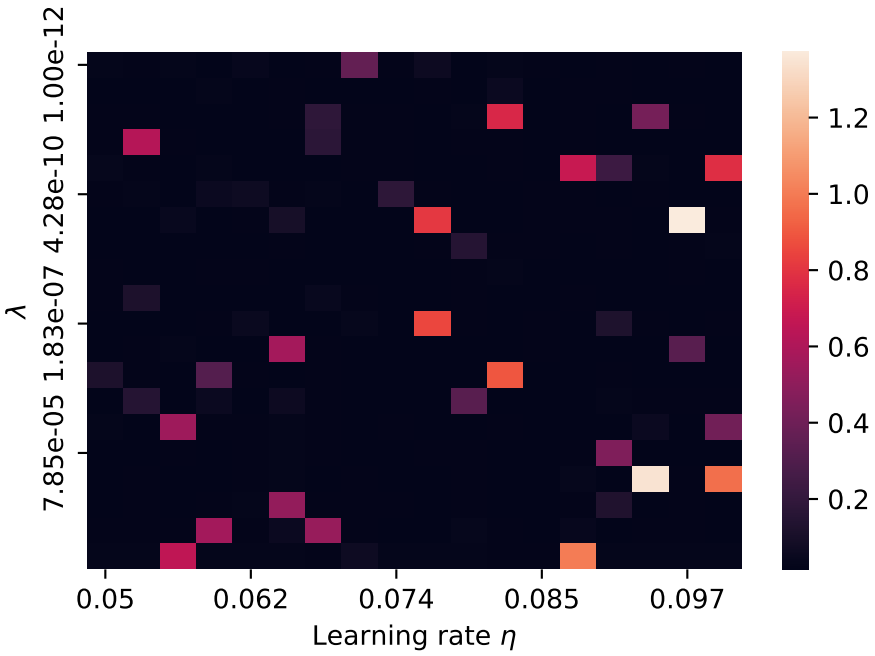
RMSprop

Same as above, but with RMSprop (with momentum)

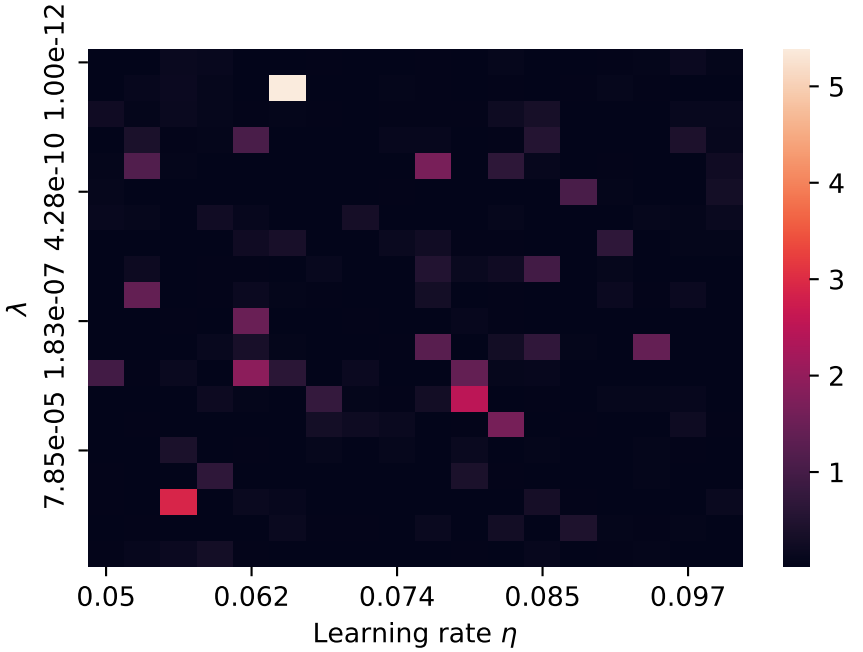
```
eta = 0.2
n_iter = 200
lmbs=np.logspace(-12,-2,20)
etas = np.linspace(0.05, 0.1, 18)

# sgd parameters
M = 5
n_epochs = 50

# momentum parameter
gamma = 0.3

# RMSProp parameter
rho=0.99

mse_gd_ridge = eta_lambda_grid(
    xtrain, xtest, y_train_scaled, y_test_scaled,
    etas, lmbs, n_iter = n_iter,
    momentum = True, gamma = gamma,
    adaptive_fun=RMSProp, rho=rho
    )

mse_sgd_ridge = eta_lambda_grid(
    xtrain, xtest, y_train_scaled, y_test_scaled,
     etas, lmbs, n_epochs=n_epochs, M=M, gd_fun=SGD,
     momentum = True, gamma = gamma,
     adaptive_fun=RMSProp, rho=rho
     )
```

```
lmb_lab = ["{0:.2e}".format(x) for x in lmbs]

sns.heatmap(mse_gd_ridge)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(lmbs))[::5] + 0.5, lmb_lab[::5])
plt.xlabel(r"Learning rate $\eta$")
```

```
plt.ylabel(r"$\lambda$")
plt.show()

sns.heatmap(mse_sgd_ridge)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(lmbs))[::5] + 0.5, lmb_lab[::5])
plt.xlabel(r"Learning rate $\eta$")
plt.ylabel(r"$\lambda$")
plt.show()
```



(a) Gradient descent, 200 iterations.



(b) Stochastic gradient descent, $M = 5$, 50 epochs.

Figure 8: Heatmap of MSE for gradient descent with the RMSProp algorithm with momentum ($\gamma = 0.3$) with varying learning rates and λs.

Here we have the opposite pattern of AdaGrad, and regular GD works best. Again this is for our subset of hyperparameters.

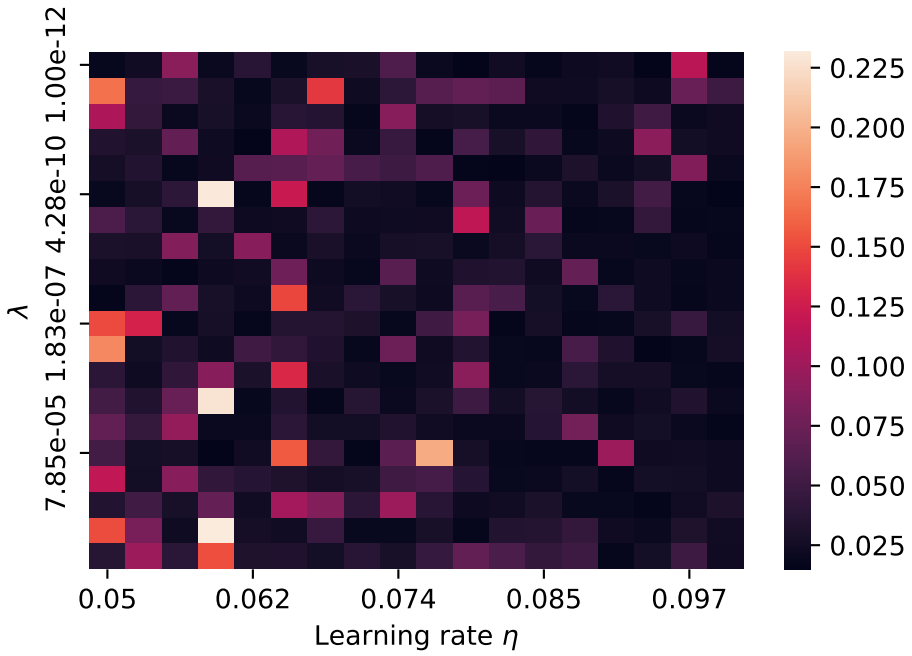
ADAM

Finally, we try ADAM. Without momentum since I was not sure how to implement it, or if it would benefit the algorithm if I did.

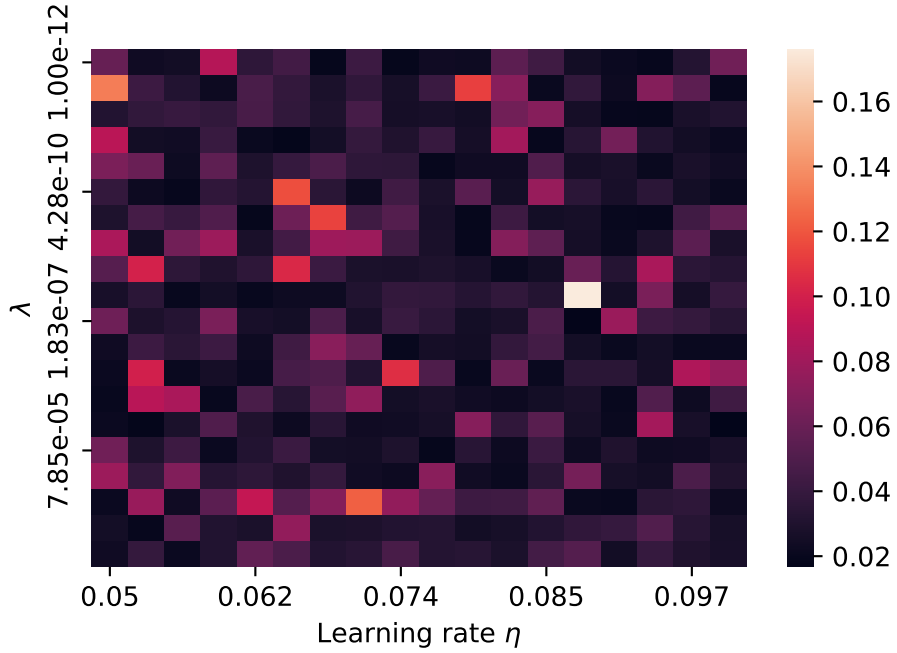
```
eta = 0.2
n_iter = 200
lmbs=np.logspace(-12,-2,20)
etas = np.linspace(0.05, 0.1, 18)

# sgd parameters
M = 5
n_epochs = 50

# Adam parameters
beta1 = 0.9
beta2 = 0.999

mse_gd_ridge = eta_lambda_grid(
    xtrain, xtest, y_train_scaled, y_test_scaled,
    etas, lmbs, n_iter = n_iter,
    adam=True, beta1 = beta1, beta2 = beta2
    )

mse_sgd_ridge = eta_lambda_grid(
    xtrain, xtest, y_train_scaled, y_test_scaled,
    etas, lmbs, n_epochs=n_epochs, M=M, gd_fun=SGD,
    adam=True, beta1 = beta1, beta2 = beta2
    )
```

```
lmb_lab = ["{0:.2e}".format(x) for x in lmbs]

sns.heatmap(mse_gd_ridge)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(lmbs))[::5] + 0.5, lmb_lab[::5])
plt.xlabel(r"Learning rate $\eta$")
plt.ylabel(r"$\lambda$")
plt.show()
```

```
sns.heatmap(mse_sgd_ridge)
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(lmbs))[::5] + 0.5, lmb_lab[::5])
plt.xlabel(r"Learning rate $\eta$")
plt.ylabel(r"$\lambda$")
plt.show()
```



(a) Gradient descent, 200 iterations.



(b) Stochastic gradient descent, $M = 5$, 50 epochs.

Figure 9: Heatmap of MSE for gradient descent with the ADAM algorithm with momentum ($\gamma = 0.3$) with varying learning rates and λs.

ADAM performed well across a wide range of learning rates. In contrast to RMSProp and AdaGrad, ADAM had similar performance for GD and SGD.

Conclusion

I have investigated gradient descent using different algorithms and a gradient derived from the Ridge cost function. I have only scratched the surface here, and in the future I want to assess how the hyperparameters of the different algorithms affect their performance. I also want to investigate the differences between the methods more directly, and for real data. Unfortunately I ran out of time for using on a weekly exercise ...

BONUS: Apply to Franke function

Create data, polynomial degree 15

```
# Make data.
x = np.arange(0, 1, 0.05)
y = np.arange(0, 1, 0.05)
x, y = np.meshgrid(x,y)


def FrankeFunction(x,y):
    term1 = 0.75*np.exp(-(0.25*(9*x-2)**2) - 0.25*((9*y-2)**2))
    term2 = 0.75*np.exp(-((9*x+1)**2)/49.0 - 0.1*(9*y+1))
    term3 = 0.5*np.exp(-(9*x-7)**2/4.0 - 0.25*((9*y-3)**2))
    term4 = -0.2*np.exp(-(9*x-4)**2 - (9*y-7)**2)
    return term1 + term2 + term3 + term4


z = FrankeFunction(x, y) + np.random.normal(0,0.1,x.shape)

xy = np.column_stack((x.flatten(), y.flatten()))
max_poly = 15
X = PolynomialFeatures(max_poly).fit_transform(xy)

np.random.seed(42)
y = z.flatten()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
# reshape y_test to 2D array, necessary for resampling later
y_test = y_test[:,np.newaxis]
```

```
y_train = y_train[:,np.newaxis]

# scale data

scalerX = StandardScaler()
X_train_scaled = scalerX.fit_transform(X_train)
X_test_scaled = scalerX.transform(X_test)

scalery = StandardScaler()
y_train_scaled = scalery.fit_transform(y_train)
y_test_scaled = scalery.transform(y_test)
```

Arbitrarily choose ADAM to use on the data

```
eta = 0.2
n_iter = 200
lmbs=np.logspace(-12,-2,20)
etas = np.linspace(0.05, 0.1, 18)

# sgd parameters
M = 5
n_epochs = 50

# Adam parameters
beta1 = 0.9
beta2 = 0.999
mse_sgd_ridge = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    etas, lmbs, n_epochs=n_epochs, M=M, gd_fun=SGD,
    adam=True, beta1 = beta1, beta2 = beta2
    )
```

```
from matplotlib.colors import LogNorm
lmb_lab = ["{0:.2e}".format(x) for x in lmbs]

sns.heatmap(mse_sgd_ridge, norm=LogNorm())
plt.xticks(np.arange(len(etas))[::4] + 0.5, np.round(etas, 3)[::4])
plt.yticks(np.arange(len(lmbs))[::5] + 0.5, lmb_lab[::5])
plt.xlabel(r"Learning rate $\eta$")
plt.ylabel(r"$\lambda$")
#plt.title(f"CV MSE ($k={k}$)")
plt.show()
```

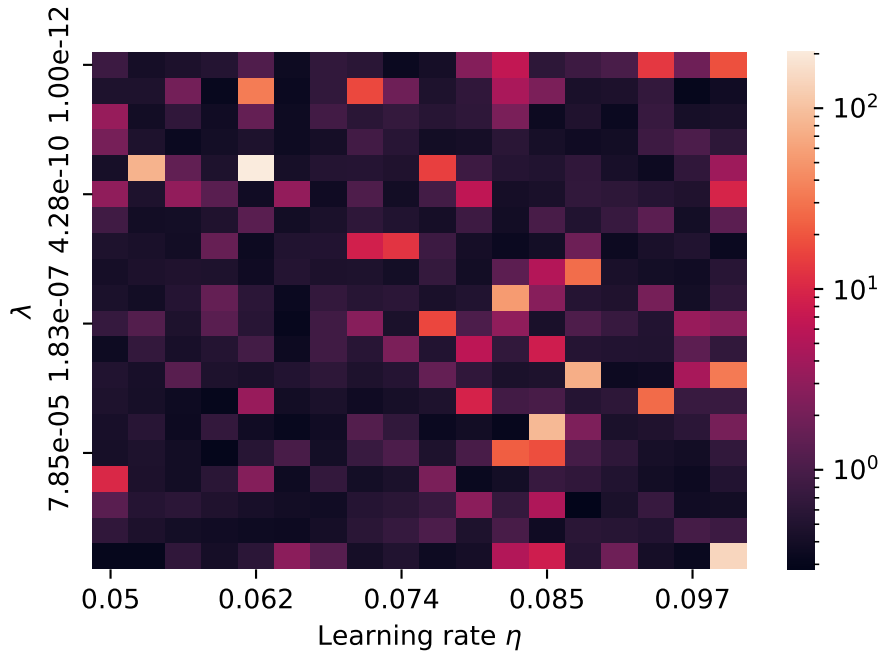


Figure 10: BONUS. the same methods applied to the Franke function. Note log scale on color gradient, unlike all previous figures …