Linear regression

February 14, 2021

1 Linear regression

Partly adapted from Deisenroth, Faisal, Ong (2020).

The purpose of this notebook is to practice implementing some linear algebra (equations provided) and to explore some properties of linear regression.

We will solely rely on the Python packages numpy and matplotlib, and you are not allowed to use any package that has a complete linear regression framework implemented (e.g., scikit-learn).

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

We consider a linear regression problem of the form

$$y = \boldsymbol{x}^T\boldsymbol{\beta} + \epsilon\,, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

where $\boldsymbol{x} \in \mathbb{R}^D$ are inputs and $y \in \mathbb{R}$ are noisy observations. The parameter vector $\boldsymbol{\beta} \in \mathbb{R}^D$ parametrizes the function.

We assume we have a training set $(\boldsymbol{x}_n, y_n)$, $n = 1, \ldots, N$. We summarize the sets of training inputs in $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$ and corresponding training targets $\mathcal{Y} = \{y_1, \ldots, y_N\}$, respectively.

In this tutorial, we are interested in finding parameters $\boldsymbol{\beta}$ that map the inputs well to the ouputs.

From our lectures, we know that the parameters $\boldsymbol{\beta}$ found by the following equation are optimal:

$$\min_{\boldsymbol{\beta}} \|\mathcal{Y} - \mathcal{X}\boldsymbol{\beta}\|^2 = \min_{\boldsymbol{\beta}} \mathrm{L}_{\mathrm{LS}}(\boldsymbol{\beta})$$

where L_{LS} is the (ordinary) least squares loss function. The solution is

$$\boldsymbol{\beta}^* = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \ \in \mathbb{R}^D\,,$$

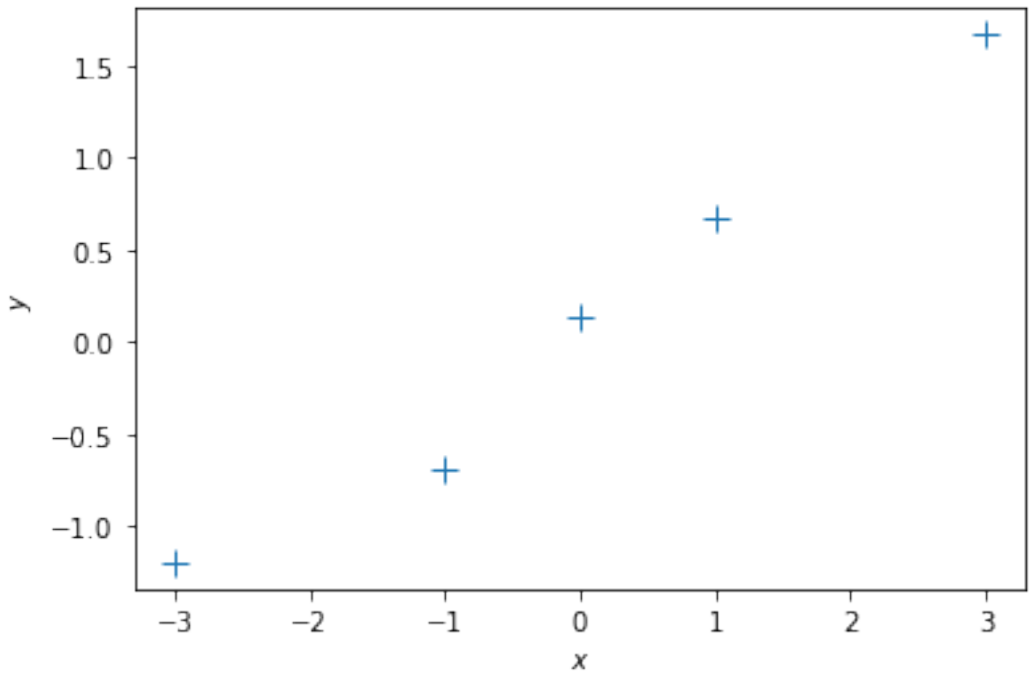
where

$$\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N]^T \in \mathbb{R}^{N\times D}\,, \quad \boldsymbol{y} = [y_1, \ldots, y_N]^T \in \mathbb{R}^N\,.$$

We will start with a simple training set, that we define by ourselves.

[2]:
```
# Define training set
X = np.array([-3, -1, 0, 1, 3]).reshape(-1,1) # 5x1 vector, N=5, D=1
y = np.array([-1.2, -0.7, 0.14, 0.67, 1.67]).reshape(-1,1) # 5x1 vector
```

```
# Plot the training set
plt.figure()
plt.plot(X, y, '+', markersize=10)
plt.xlabel("$x$")
plt.ylabel("$y$");
```



1.1 1. Maximum likelihood

We will start with what you have gotten to know as the statistical interpretation of linear regression, i.e., the maximum likelihood estimation of the parameters $\boldsymbol{\beta}$. In maximum likelihood estimation, we find the parameters $\boldsymbol{\beta}^{\mathrm{ML}}$ that maximize the likelihood

$$p(\mathcal{Y}|\mathcal{X}, \boldsymbol{\beta}) = \prod_{n=1}^{N} p(y_n|\boldsymbol{x}_n, \boldsymbol{\beta}) \,.$$

From the lecture we know that the maximum likelihood estimator is given by

$$\boldsymbol{\beta}^{\mathrm{ML}} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y} \,.$$

Let us compute the maximum likelihood estimate for the given training set.

[3]:
```
## EDIT THIS FUNCTION (done)
def max_lik_estimate(X, y):

    # X: N x D matrix of training inputs
    # y: N x 1 vector of training targets/observations
```

```
    # returns: maximum likelihood parameters (D x 1)

    N, D = X.shape
    beta_ml = np.linalg.solve(X.T @ X, X.T @ y) ## <-- EDIT THIS LINE (done)
    return beta_ml
```

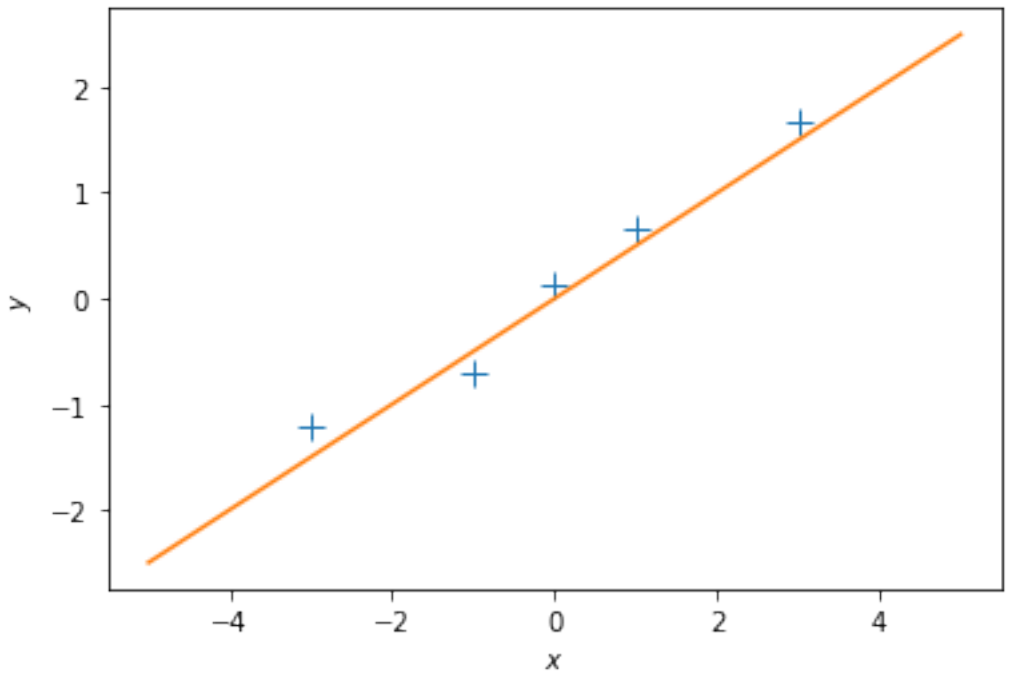
[4]:
```
# get maximum likelihood estimate
beta_ml = max_lik_estimate(X,y)
```

Now, make a prediction using the maximum likelihood estimate that we just found.

[5]:
```
## EDIT THIS FUNCTION (done)
def predict_with_estimate(X_test, beta):

    # X_test: K x D matrix of test inputs
    # beta: D x 1 vector of parameters
    # returns: prediction of f(X_test); K x 1 vector

    prediction = X_test @ beta ## <-- EDIT THIS LINE (done)

    return prediction
```

Let's see whether we got something useful:

[6]:
```
# define a test set
X_test = np.linspace(-5,5,100).reshape(-1,1) # 100 x 1 vector of test inputs

# predict the function values at the test points using the maximum likelihood␣
 ↪estimator
ml_prediction = predict_with_estimate(X_test, beta_ml)

# plot
plt.figure()
plt.plot(X, y, '+', markersize=10)
plt.plot(X_test, ml_prediction)
plt.xlabel("$x$")
plt.ylabel("$y$");
```



Questions

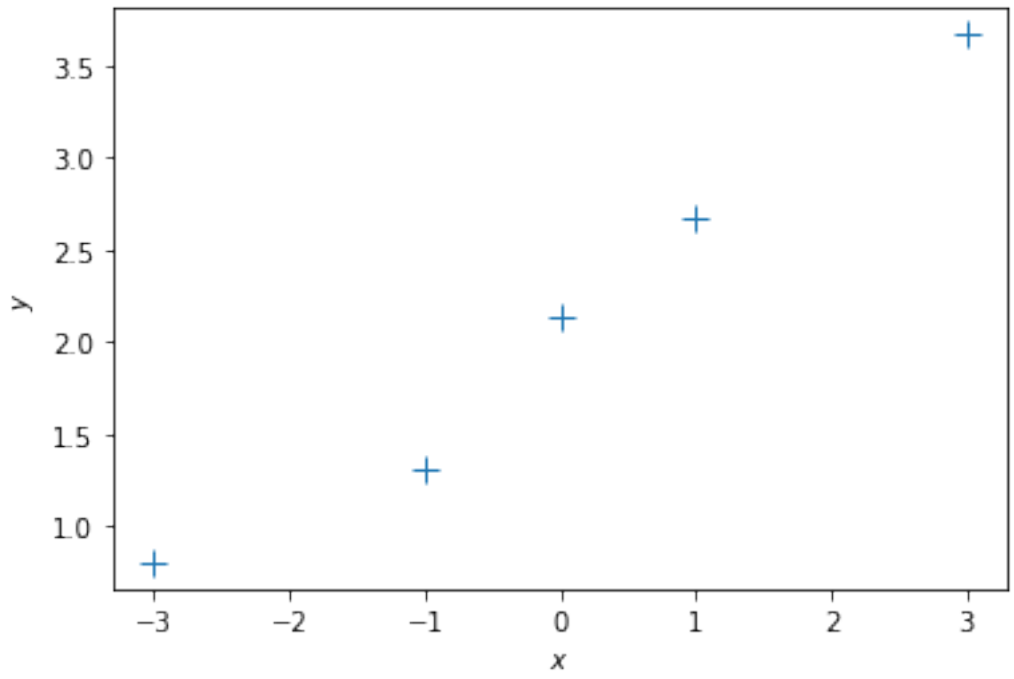
1. Does the solution above look reasonable?
2. Play around with different values of β. How do the corresponding functions change?
3. Modify the training targets $\mathcal{Y}$ and re-run your computation. What changes?

Answers

1. Yes (appears to pass closely to all points)
2. slope of line changes
3. new appropriate LS estimator obtained (NB need to change linspace option to plot full line)

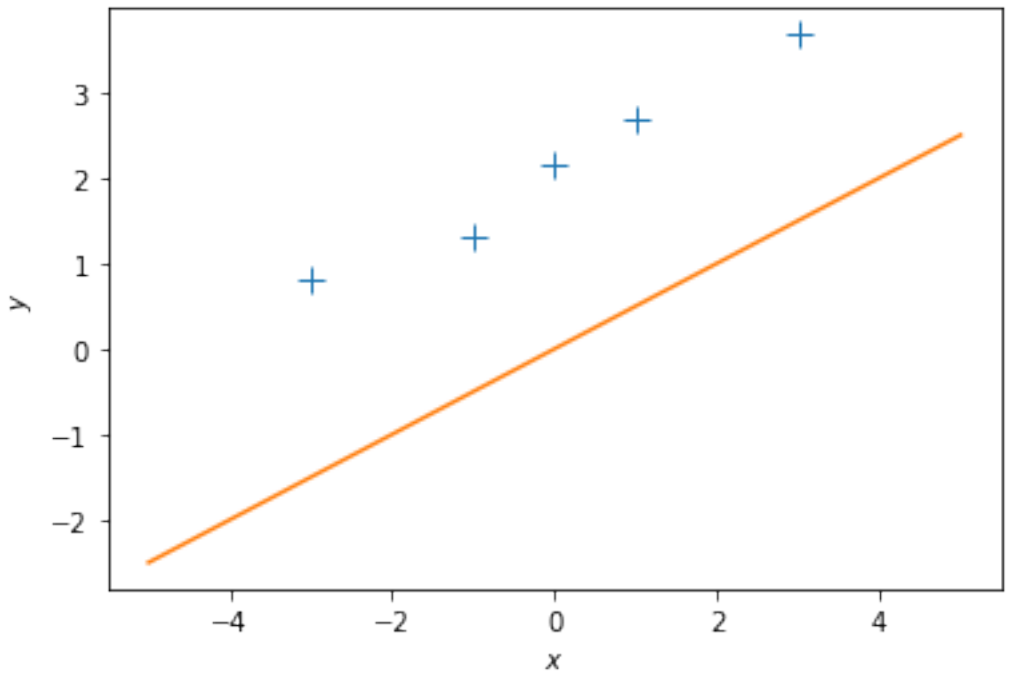
Let us now look at a different training set, where we add 2.0 to every y-value, and compute the maximum likelihood estimate.

[7]:
```
ynew = y + 2.0

plt.figure()
plt.plot(X, ynew, '+', markersize=10)
plt.xlabel("$x$")
plt.ylabel("$y$");
```



[8]:
```
# get maximum likelihood estimate
beta_ml = max_lik_estimate(X, ynew)
print(beta_ml)

# define a test set
X_test = np.linspace(-5,5,100).reshape(-1,1) # 100 x 1 vector of test inputs

# predict the function values at the test points using the maximum likelihood␣
 ↪estimator
ml_prediction = predict_with_estimate(X_test, beta_ml)

# plot
plt.figure()
plt.plot(X, ynew, '+', markersize=10)
plt.plot(X_test, ml_prediction)
plt.xlabel("$x$")
plt.ylabel("$y$");
```

```
[[0.499]]
```



Question:

1. This maximum likelihood estimate doesn't look too good: The orange line is too far away from the observations although we just shifted them by 2. Why is this the case?

- This model fixes intercept at (0,0), doesn't account for non-zero y intercept

2. How can we fix this problem?

- Adding an extra beta_0 parameter for the intercept

Let us now define a linear regression model that is slightly more flexible:

$$y = \beta_0 + \boldsymbol{x}^T \boldsymbol{\beta}_1 + \epsilon\,, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

Here, we added an offset (also called intercept) parameter β_0 to our original model.

Question:

1. What is the effect of this bias parameter, i.e., what additional flexibility does it offer?

- Line can intercept the y axis at the point for which error is minimised

If we now define the inputs to be the augmented vector $\boldsymbol{x}_{\text{aug}} = \begin{bmatrix} 1 \\ \boldsymbol{x} \end{bmatrix}$, we can write the new linear regression model as

$$y = \boldsymbol{x}_{\text{aug}}^T \boldsymbol{\beta}_{\text{aug}} + \epsilon\,, \quad \boldsymbol{\beta}_{\text{aug}} = \begin{bmatrix} \beta_0 \\ \boldsymbol{\beta}_1 \end{bmatrix}\,.$$

[9]:
```
N, D = X.shape
X_aug = np.hstack([np.ones((N,1)), X]) # augmented training inputs of size N x (D+1)
beta_aug = np.zeros((D+1, 1)) # new beta vector of size (D+1) x 1
```

Let us now compute the maximum likelihood estimator for this setting.

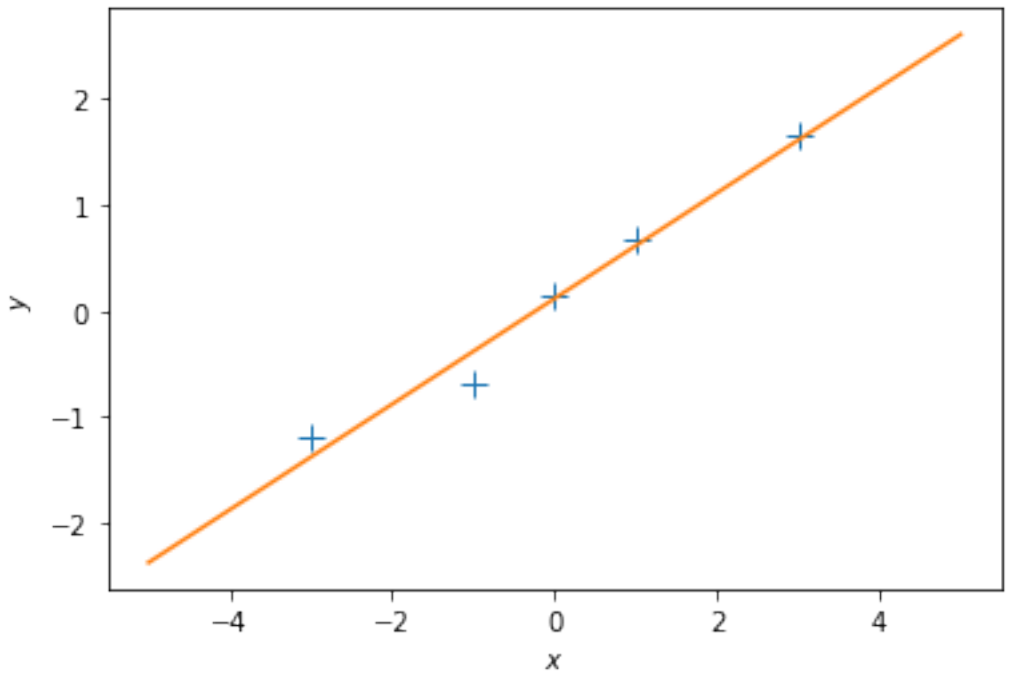
Hint: If possible, re-use code that you have already written.

[10]:
```
## EDIT THIS FUNCTION (done)
def max_lik_estimate_aug(X_aug, y):

    beta_aug_ml = max_lik_estimate(X_aug, y) ## <-- EDIT THIS LINE (done)

    return beta_aug_ml
```

[11]:
```
beta_aug_ml = max_lik_estimate_aug(X_aug, y)
```

Now, we can make predictions again:

[12]:
```
# define a test set (we also need to augment the test inputs with ones)
X_test_aug = np.hstack([np.ones((X_test.shape[0],1)), X_test]) # 100 x (D + 1) vector of test inputs

# predict the function values at the test points using the maximum likelihood estimator
ml_prediction = predict_with_estimate(X_test_aug, beta_aug_ml)

# plot
plt.figure()
plt.plot(X, y, '+', markersize=10)
plt.plot(X_test, ml_prediction)
plt.xlabel("$x$")
plt.ylabel("$y$");
```



It seems this has solved our problem! #### Question: 1. Play around with the first parameter of $\boldsymbol{\beta}_{\text{aug}}$ and see how the fit of the function changes. - changes intercept 2. Play around with the second parameter of $\boldsymbol{\beta}_{\text{aug}}$ and see how the fit of the function changes. - changes slope

1.2 2. Ridge regression

From our lectures, we know that ridge regression is an extension of linear regression with least squares loss function, including a (usually small) positive penalty term λ:

$$\min_{\boldsymbol{\beta}} \|\mathcal{Y} - \mathcal{X}\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2 = \min_{\boldsymbol{\beta}} \mathrm{L}_{\text{ridge}}(\boldsymbol{\beta})$$

where $\mathrm{L}_{\text{ridge}}$ is the ridge loss function. The solution is

$$\boldsymbol{\beta}^*_{\text{ridge}} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda I)^{-1}\boldsymbol{X}^T\boldsymbol{y}\,.$$

This time, we will define a very small training set of only two observations to demonstrate the advantages of ridge regression over least squares linear regression.

[13]:
```
X_train = np.array([0.5, 1]).reshape(-1,1)
y_train = [0.5, 1]
X_test = np.array([0, 2]).reshape(-1,1)
```

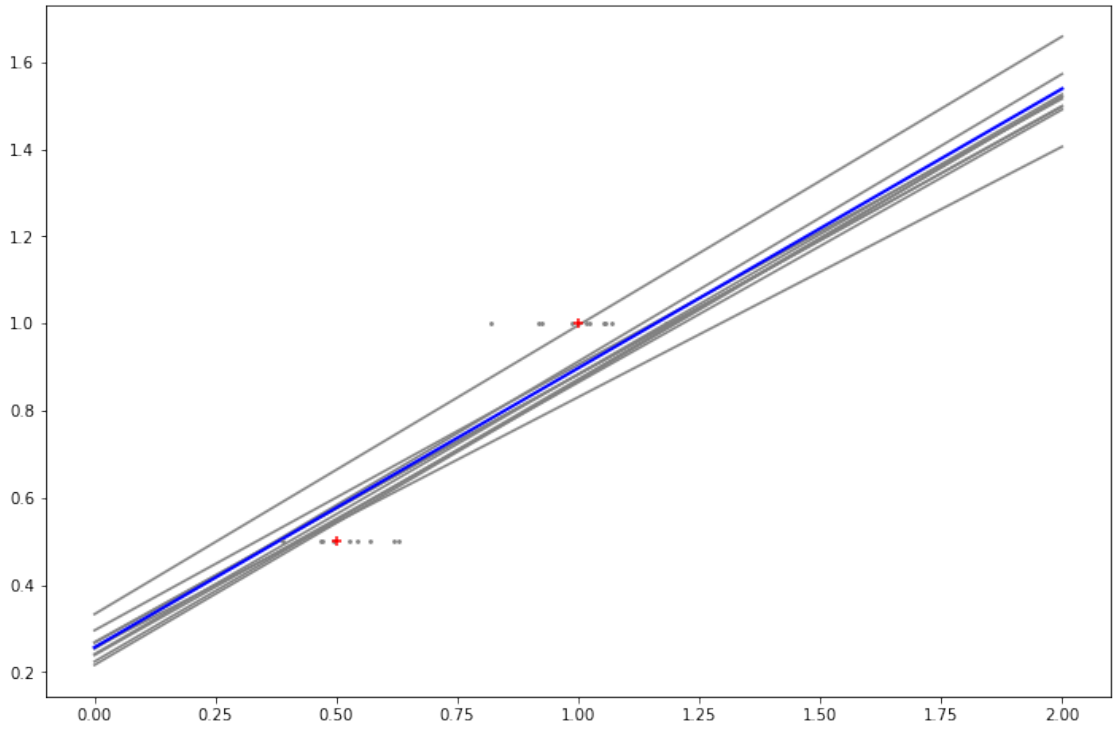
Let's define function similar to the one for least squares, but taking one additional argument, our penalty term λ.

Hint: we apply the same augmentation as above with least squares, so the offset is accurately captured.

[14]:
```
## EDIT THIS FUNCTION (done)
def ridge_estimate(X, y, penalty):

    # X: N x D matrix of training inputs
    # y: N x 1 vector of training targets/observations
    # returns: maximum likelihood parameters (D x 1)

    N, D = X.shape
    X_aug = np.hstack([np.ones((N,1)), X]) # augmented training inputs of size N x (D+1)
    N_aug, D_aug = X_aug.shape
    I = np.identity(D_aug)
    beta_ridge = np.linalg.solve(X_aug.T @ X_aug + penalty * I, X_aug.T @ y) ## <-- EDIT THIS LINE (done)
    return beta_ridge
```

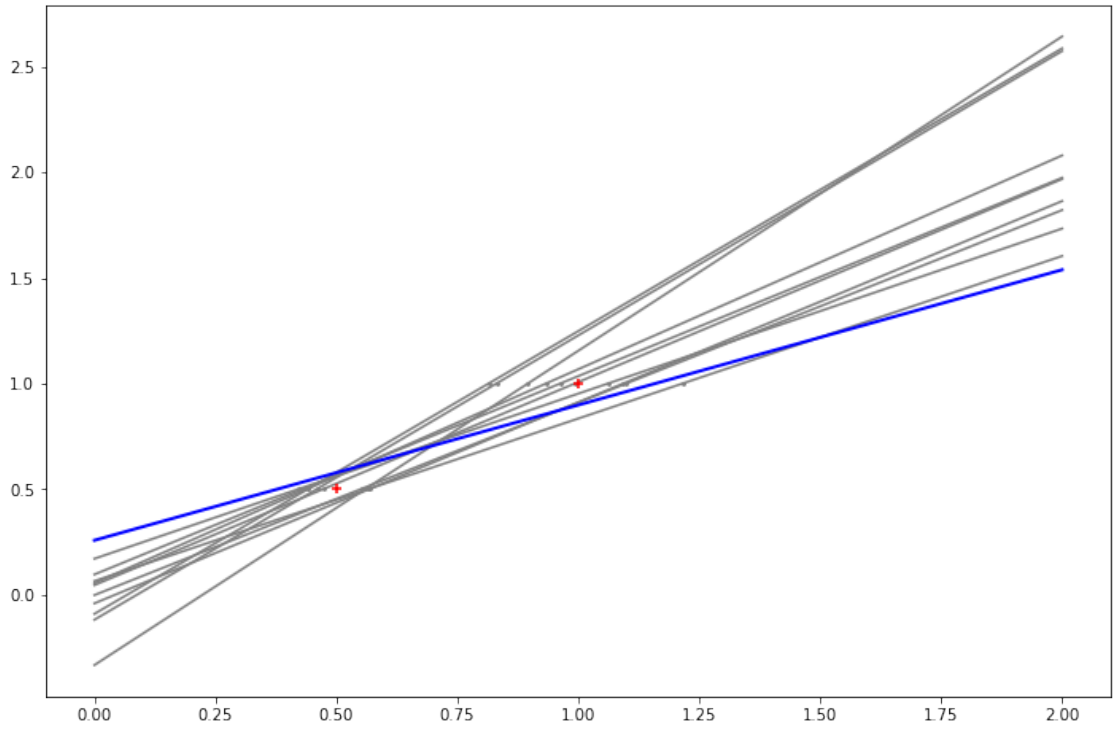
Now, we add a bit of Gaussian noise to our training set and apply ridge regression. We should do it a couple of times to be sure about the results (here 10 times).

[15]:
```
penalty_term = 0.1
fig, ax = plt.subplots(figsize=(12, 8))
X_test_aug = np.hstack([np.ones((X_test.shape[0],1)), X_test])

for _ in range(10):
  this_X = 0.1 * np.random.normal(size=(2, 1)) + X_train

  beta_ridge = ridge_estimate(this_X, y_train, penalty=penalty_term)
  ridge_prediction = predict_with_estimate(X_test_aug, beta_ridge)

  ax.plot(X_test, ridge_prediction, color='gray')
  ax.scatter(this_X, y_train, s=3, c='gray', marker='o', zorder=10)

beta_ridge = ridge_estimate(X_train, y_train, penalty=penalty_term)
ridge_prediction_X = predict_with_estimate(X_test_aug, beta_ridge)

ax.plot(X_test, ridge_prediction_X, linewidth=2, color='blue')
ax.scatter(X_train, y_train, s=30, c='red', marker='+', zorder=10);
```



Let's compare this to ordinary least squares:

[16]:
```
fig, ax = plt.subplots(figsize=(12, 8))
X_test_aug = np.hstack([np.ones((X_test.shape[0],1)), X_test])

for _ in range(10):
  this_X = 0.1 * np.random.normal(size=(2, 1)) + X_train
  N, D = this_X.shape
  this_X_aug = np.hstack([np.ones((N,1)), this_X])

  beta_aug_ml = max_lik_estimate_aug(this_X_aug, y_train)
  ml_prediction = predict_with_estimate(X_test_aug, beta_aug_ml)

  ax.plot(X_test, ml_prediction, color='gray')
  ax.scatter(this_X, y_train, s=3, c='gray', marker='o', zorder=10)

beta_aug_ml = max_lik_estimate_aug(X_train, y_train)
ml_prediction_X = predict_with_estimate(X_test_aug, beta_ridge)

ax.plot(X_test, ml_prediction_X, linewidth=2, color='blue')
ax.scatter(X_train, y_train, s=30, c='red', marker='+', zorder=10);
```



Questions

1. What differences between the two solutions above can you see?

- Ordinary least squares is much more variable, the lines we obtain from different error terms are very different, but seems to go trhough points exactly (0 bias, high variance)
- Ridge estimation has solutions that are less variable but they not necessarily go through the points exactly (slight bias, low variance)

2. Play around with different values of the penalty term λ. How do the corresponding functions change? Which values provide the most reasonable results?

- lambda = 0 recovers the least squares estimate
- as lambda increases, the bias increases vastly, so lines are not near the 2 original points at all
- values around 0.1 seem to produce good results

[ ]: