Machine Learning I - Practical I

Name: {Abhijit Kumar Baruah}

Course: {Quantitative Data Science Methods: Psychometrics, Econometrics and Machine Learning}

This notebook provides you with the assignments and the overall code structure you need to complete the assignment. Each exercise indicates the total number of points allocated. There are also questions that you need to answer in text form. Please use full sentences and reasonably correct spelling/grammar.

Regarding submission & grading:

- Solutions can be uploaded to ILIAS until the indicated deadline. Please upload a copy of this notebook and a PDF version of it after you ran it.
- Please hand in your own solution. You are encouraged to discuss your code with classmates and help each other, but after that, please sit down for yourself and write your own code.
- We will grade not only based on the outputs, but also look at the code. So please use comments make us understand what you intended to do :)
- For plots you create yourself, all axes must be labeled.
- DO NOT IN ANY CASE change the function interfaces.
- If you are not familiar with python, then this may be of help. This is a colab notebook which is part of the Convolutional Neural Networks for Visual Recognition course (CS231n) in Stanford. It goes through some of the basic elements of Python and Numpy. You can run it directly in the browser without having Python installed:

  https://colab.research.google.com/github/cs231n/cs231n.github.io/blob/master/python-colab.ipynb#scrollTo=U1PvreR9L9eW (https://colab.research.google.com/github/cs231n/cs231n.github.io/blob/master/python-colab.ipynb#scrollTo=U1PvreR9L9eW)

- If you are not familiar with python, but used MATLAB before, check out this reference pages listing what you want to use as python equivalent of a certain MATLAB command:

  https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html (https://docs.scipy.org/doc/numpy/user/numpy-for-matlab-users.html)

  http://www.eas.uccs.edu/~mwickert/ece5650/notes/NumPy2MATLAB.pdf (http://www.eas.uccs.edu/~mwickert/ece5650/notes/NumPy2MATLAB.pdf)

  http://mathesaurus.sourceforge.net/matlab-python-xref.pdf (http://mathesaurus.sourceforge.net/matlab-python-xref.pdf)

  or, if you prefer to read a longer article, try:

  https://realpython.com/matlab-vs-python/#learning-about-pythons-mathematical-libraries (https://realpython.com/matlab-vs-python/#learning-about-pythons-mathematical-libraries)

In [1]:
```
%matplotlib notebook

import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from scipy.io import loadmat
from scipy import stats
import copy
import pylab
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
```

The dataset

The dataset consists of over 20.000 materials and lists their physical features. From these features, we want to learn how to predict the critical temperature, i.e. the temperature we need to cool the material to so it becomes superconductive. First load and familiarize yourself with the data set a bit.

In [2]:
```
data=pd.read_csv('superconduct_train.csv')
print(data.shape)
```

```
(21263, 82)
```

In [3]:
```
data.head()
```

Out[3]:

	number_of_elements	mean_atomic_mass	wtd_mean_atomic_mass	gmean_atomic_mass	wtd_g
0	4	88.944468	57.862692	66.361592	
1	5	92.729214	58.518416	73.132787	
2	4	88.944468	57.885242	66.361592	
3	4	88.944468	57.873967	66.361592	
4	4	88.944468	57.840143	66.361592	

5 rows × 82 columns

Because the dataset is rather large, we prepare a small subset of the data as training set, and another subset as test set. To make the computations reproducible, we set the random seed.

In [4]:
```
target_clm = 'critical_temp' # the critical temperature is our target variable
n_trainset = 200 # size of the training set
n_testset = 500 #size of the test set
```

In [5]:
```
# set random seed to make sure every test set is the same
np.random.seed(seed=1)

idx = np.arange(data.shape[0])
idx_shuffled = np.random.permutation(idx) # shuffle indices to split into trai
ning and test set

test_idx = idx_shuffled[:n_testset]
train_idx = idx_shuffled[n_testset:n_testset+n_trainset]
train_full_idx = idx_shuffled[n_testset:]

X_test = data.loc[test_idx, data.columns != target_clm].values
y_test = data.loc[test_idx, data.columns == target_clm].values
print('Test set shapes (X and y)', X_test.shape, y_test.shape)

X_train = data.loc[train_idx, data.columns != target_clm].values
y_train = data.loc[train_idx, data.columns == target_clm].values
print('Small training set shapes (X and y):',X_train.shape, y_train.shape)

X_train_full = data.loc[train_full_idx, data.columns != target_clm].values
y_train_full = data.loc[train_full_idx, data.columns == target_clm].values
print('Full training set shapes (X and y):',X_train_full.shape, y_train_full.s
hape)
```

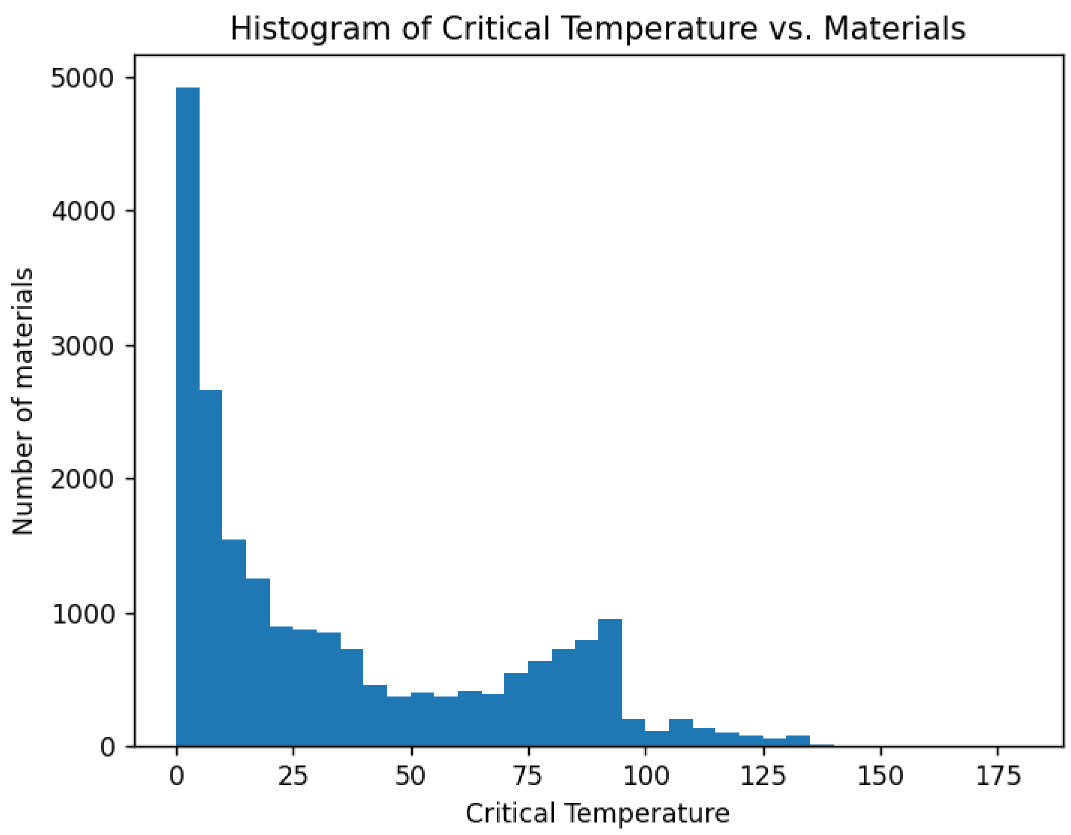
```
Test set shapes (X and y) (500, 81) (500, 1)
Small training set shapes (X and y): (200, 81) (200, 1)
Full training set shapes (X and y): (20763, 81) (20763, 1)
```

Task 1: Plot the dataset [5 pts]

To explore the dataset, use `X_train_full` and `y_train_full` for two descriptive plots:

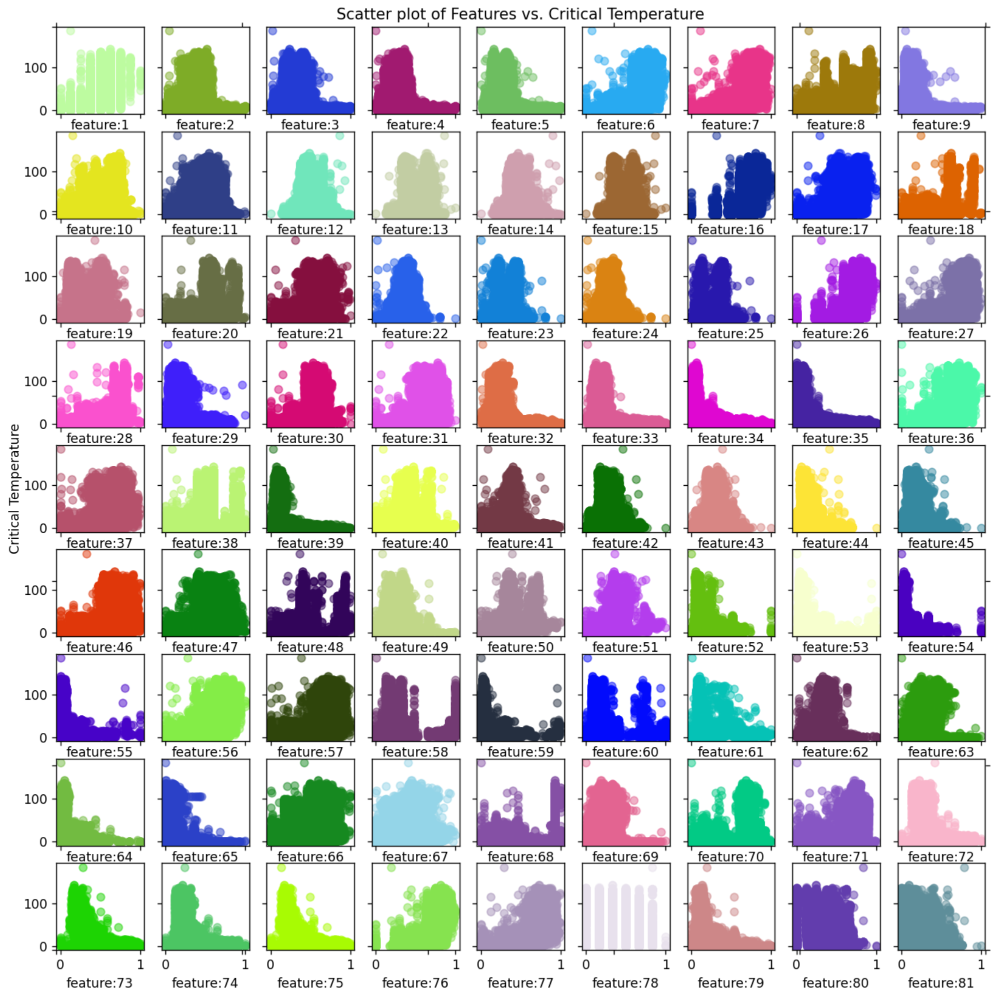
- **Histogram** of the target variable. Use `plt.hist` .
- **Scatterplots** relating the target variable to one of the feature values. For this you will need 81 scatterplots. Arrange them in one big figure with 9x9 subplots. Use `plt.scatter` . You may need to adjust the marker size and the alpha blending value.

In [6]:
```
# Histogram of the target variable

plt.hist(y_train_full, bins=range(0,185,5)) # histogram of target variable y_t
rain_full is plotted with specific bins
plt.xlabel('Critical Temperature') # x-axis is labelled
plt.ylabel('Number of materials') # y-axis is labelled
plt.title('Histogram of Critical Temperature vs. Materials') # histogram is ti
tled
```



Out[6]:
```
Text(0.5, 1.0, 'Histogram of Critical Temperature vs. Materials')
```

In [7]:
```
# Scatter plots of the target variable vs. features

# ADD YOUR CODE HERE
# First we make our 9 by 9  grid. Individual grids will contain a scatter plot
of feature vs. critical temperature
# Our axes ideally should have the same limits, ticks, and scale of axes as th
e shared axes.
# We do this to make a reliable comparison among the plots
fig, axes = plt.subplots(9, 9, sharex=True, sharey=True, figsize=(13, 13))

# We then add the largest axis and hide frame.
# On this axis and within this frame we will have all the subplots.
fig.add_subplot(111, frameon=False)

# hide tick and tick label of the axes and add y-label to the largest axis.
plt.tick_params(labelcolor='none', top='off', bottom='off', left='off', right=
'off')
plt.grid(False)
#plt.xlabel("Physical Features")
plt.ylabel("Critical Temperature")
plt.title("Scatter plot of Features vs. Critical Temperature")

# Now we are ready to make our scatter plots. But, as our axes are shared,
# we need to normalize X_train_full by columns first
# Here, use the package sklearn and its associated preprocessing utilities to
 normalize the data
import pandas as pd
from sklearn import preprocessing
x = X_train_full # we do not want to change the original X_train_full
min_max_scaler = preprocessing.MinMaxScaler()
normalized_X_train_full = min_max_scaler.fit_transform(x)
df = pd.DataFrame(normalized_X_train_full)

# we start plotting the target variable vs 1st feature and loop over the rest
# We assign a random color to the scatter plots and an alpha blending value
feature = 0
for n in range(81):
    axes[n // 9, n % 9].scatter(normalized_X_train_full[:, feature], y_train_f
ull, color = np.random.rand(3,), alpha=0.5)
    axes[n // 9, n % 9].set_xlabel('feature:{}'.format(feature+1))
    feature += 1
```



Which material properties may be useful for predicting superconductivity? What other observations can you make?

In [8]:
```
# To answer the above question, we would need to extract the names of the mate
rial properties corresponding to the different
# scatter plots above. We write the following code.
index = 1
for col in data.columns[1:81]:
    print(col, "(", index//9 ,",", index%9,")")
    index += 1
```

```
mean_atomic_mass ( 0 , 1 )
wtd_mean_atomic_mass ( 0 , 2 )
gmean_atomic_mass ( 0 , 3 )
wtd_gmean_atomic_mass ( 0 , 4 )
entropy_atomic_mass ( 0 , 5 )
wtd_entropy_atomic_mass ( 0 , 6 )
range_atomic_mass ( 0 , 7 )
wtd_range_atomic_mass ( 0 , 8 )
std_atomic_mass ( 1 , 0 )
wtd_std_atomic_mass ( 1 , 1 )
mean_fie ( 1 , 2 )
wtd_mean_fie ( 1 , 3 )
gmean_fie ( 1 , 4 )
wtd_gmean_fie ( 1 , 5 )
entropy_fie ( 1 , 6 )
wtd_entropy_fie ( 1 , 7 )
range_fie ( 1 , 8 )
wtd_range_fie ( 2 , 0 )
std_fie ( 2 , 1 )
wtd_std_fie ( 2 , 2 )
mean_atomic_radius ( 2 , 3 )
wtd_mean_atomic_radius ( 2 , 4 )
gmean_atomic_radius ( 2 , 5 )
wtd_gmean_atomic_radius ( 2 , 6 )
entropy_atomic_radius ( 2 , 7 )
wtd_entropy_atomic_radius ( 2 , 8 )
range_atomic_radius ( 3 , 0 )
wtd_range_atomic_radius ( 3 , 1 )
std_atomic_radius ( 3 , 2 )
wtd_std_atomic_radius ( 3 , 3 )
mean_Density ( 3 , 4 )
wtd_mean_Density ( 3 , 5 )
gmean_Density ( 3 , 6 )
wtd_gmean_Density ( 3 , 7 )
entropy_Density ( 3 , 8 )
wtd_entropy_Density ( 4 , 0 )
range_Density ( 4 , 1 )
wtd_range_Density ( 4 , 2 )
std_Density ( 4 , 3 )
wtd_std_Density ( 4 , 4 )
mean_ElectronAffinity ( 4 , 5 )
wtd_mean_ElectronAffinity ( 4 , 6 )
gmean_ElectronAffinity ( 4 , 7 )
wtd_gmean_ElectronAffinity ( 4 , 8 )
entropy_ElectronAffinity ( 5 , 0 )
wtd_entropy_ElectronAffinity ( 5 , 1 )
range_ElectronAffinity ( 5 , 2 )
wtd_range_ElectronAffinity ( 5 , 3 )
std_ElectronAffinity ( 5 , 4 )
wtd_std_ElectronAffinity ( 5 , 5 )
mean_FusionHeat ( 5 , 6 )
wtd_mean_FusionHeat ( 5 , 7 )
gmean_FusionHeat ( 5 , 8 )
wtd_gmean_FusionHeat ( 6 , 0 )
entropy_FusionHeat ( 6 , 1 )
wtd_entropy_FusionHeat ( 6 , 2 )
range_FusionHeat ( 6 , 3 )
```

```
wtd_range_FusionHeat ( 6 , 4 )
std_FusionHeat ( 6 , 5 )
wtd_std_FusionHeat ( 6 , 6 )
mean_ThermalConductivity ( 6 , 7 )
wtd_mean_ThermalConductivity ( 6 , 8 )
gmean_ThermalConductivity ( 7 , 0 )
wtd_gmean_ThermalConductivity ( 7 , 1 )
entropy_ThermalConductivity ( 7 , 2 )
wtd_entropy_ThermalConductivity ( 7 , 3 )
range_ThermalConductivity ( 7 , 4 )
wtd_range_ThermalConductivity ( 7 , 5 )
std_ThermalConductivity ( 7 , 6 )
wtd_std_ThermalConductivity ( 7 , 7 )
mean_Valence ( 7 , 8 )
wtd_mean_Valence ( 8 , 0 )
gmean_Valence ( 8 , 1 )
wtd_gmean_Valence ( 8 , 2 )
entropy_Valence ( 8 , 3 )
wtd_entropy_Valence ( 8 , 4 )
range_Valence ( 8 , 5 )
wtd_range_Valence ( 8 , 6 )
std_Valence ( 8 , 7 )
wtd_std_Valence ( 8 , 8 )
```

YOUR ANSWER HERE

From the different scatter plots above, the ones that possibly make the most sense are in coordinates: $(3,5),(3,6),(3,7),(4,2),(5,8),(7,0),(7,1),(8,2)$ and $(8,6)$. In each of these scatter plots a decreasing trend in $temperatures$ is seen after an initial spike. The correspoding material properties are: wtd_mean_Density, gmean_Density, wtd_gmean_Density, wtd_range_Density, gmean_FusionHeat, gmean_ThermalConductivity, wtd_gmean_ThermalConductivity, wtd_gmean_Valence and wtd_range_Valence.

Another observation is that in every subplot, there is an element that reached the maximum critical temperature.

Task 2: Implement your own OLS estimator [10 pts]

We want to use linear regression to predict the critical temperature. Implement the ordinary least squares estimator without regularization 'by hand':

$$w = (X^TX)^{-1}X^Ty$$

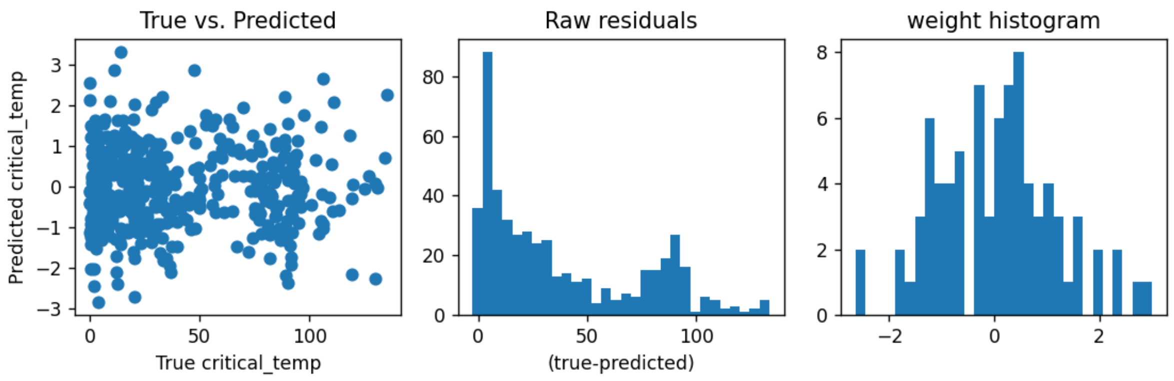
To make life a bit easier, we provide a function that can be used to plot regression results. In addition it computes the mean squared error and the squared correlation between the true and predicted values.

In [9]:
```
def plot_regression_results(y_test,y_pred,weights):
    '''Produces three plots to analyze the results of linear regression:
        -True vs predicted
        -Raw residual histogram
        -Weight histogram

    Inputs:
        y_test: (n_observations,) numpy array with true values
        y_pred: (n_observations,) numpy array with predicted values
        weights: (n_weights) numpy array with regression weights'''

    print('MSE: ', mean_squared_error(y_test,y_pred))
    print('r^2: ', r2_score(y_test,y_pred))

    fig,ax = plt.subplots(1,3,figsize=(9,3))
    #predicted vs true
    ax[0].scatter(y_test,y_pred)
    ax[0].set_title('True vs. Predicted')
    ax[0].set_xlabel('True %s' % (target_clm))
    ax[0].set_ylabel('Predicted %s' % (target_clm))

    #residuals
    error = np.squeeze(np.array(y_test)) - np.squeeze(np.array(y_pred))
    ax[1].hist(np.array(error),bins=30)
    ax[1].set_title('Raw residuals')
    ax[1].set_xlabel('(true-predicted)')

    #weight histogram
    ax[2].hist(weights,bins=30)
    ax[2].set_title('weight histogram')

    plt.tight_layout()
```

As an example, we here show you how to use this function with random data.

In [10]:
```
# weights is a vector of length 82: the first value is the intercept (beta0),
 then 81 coefficients
weights = np.random.randn(82)

# Model predictions on the test set
y_pred_test = np.random.randn(y_test.size)

plot_regression_results(y_test, y_pred_test, weights)
```

```
MSE:  2646.872810973404
r^2:  -1.113204595044524
```

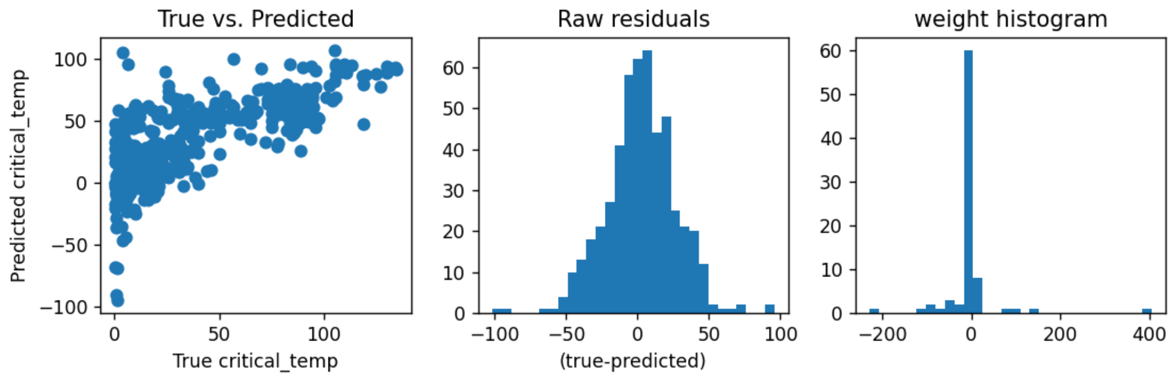


Implement OLS linear regression yourself. Use `X_train` and `y_train` for estimating the weights, and use `X_test` to compute test set predictions. `OLS_regression` is supposed to return those. Call our plotting function with the regession result to compute the MSE and r^2. You should get mean squared error of 599.7.

In [11]:
```
def OLS_regression(X_test, X_train, y_train):
    '''Computes OLS weights for linear regression without regularization on th
e training set and
       returns weights and testset predictions.

       Inputs:
         X_test: (n_observations, 81), numpy array with predictor values of th
e test set
         X_train: (n_observations, 81), numpy array with predictor values of t
he training set
         y_train: (n_observations,) numpy array with true target values for th
e training set

       Outputs:
         weights: The weight vector for the regerssion model including the off
set
         y_pred: The predictions on the TEST set

       Note:
         Both the training and the test set need to be appended manually by a
 columns of 1s to add
         an offset term to the linear regression model.

    '''

    # ADD YOUR CODE HERE
    # We first appended manually on both the training and test set by a column
s of 1s
    # to add an offset term to the linear regression model.
    n1, m1 = X_train.shape # dimensions of the X_train
    n2, m2 = X_test.shape # dimensions of the X_test
    X_train_1 = np.ones((n1,1)) # column vector of ones to append to the X_tra
in
    X_test_1 = np.ones((n2,1)) # column vector of ones to append to the X_test
    X_train_new = np.hstack((X_train_1, X_train)) # new X_train with the offse
t terms
    X_test_new = np.hstack((X_test_1, X_test)) # new X_test with the offset te
rms


    X_transpose = np.transpose(X_train_new) # transpose of the new X_train set
to be used in computation

    # trial code using matrix multiplications (bad for computation)
    # This helps us keep track of what needs to be done with expanded matrix m
ultiplications
    # X_tX = np.matmul(X_transpose, X_train_new)
    # weights = np.matmul(np.matmul(np.linalg.inv(X_tX), X_transpose), y_trai
n)
    # y_pred = np.matmul(X_test_new, weights)

    X_tX = np.zeros((m1+1,m1+1)) # initializing the matrix X'X
    X_ty = np.zeros((m1+1)) # initializing the matrix X'y
    weights = np.zeros((m1+1)) # initializing the vector of weights to be upda
ted
    for i in range(m1+1):
        for j in range(m1+1):
            X_tX[i,j] = np.dot(X_transpose[i,:],X_train_new[:,j]) # calculate
 X'X
        X_ty[i] = np.dot(X_transpose[i,:],y_train) # calculate X'y

    X_tX_inv = np.linalg.inv(X_tX) # calculate the inverse of X'X
    for i in range(m1+1):
        weights[i] = np.dot(X_tX_inv[i,:],X_ty) # calculate element-wise (X'X)
^{-1}X'y as a dot product

    # So we have our weights vector. All that remains is to calculate predicte
d y values.
    y_pred = np.matmul(X_test_new, weights)
    # another element-wise dot product can be done above as well.But, as both
 new X_test and weights
    # are well defined, we can safely proceed with just matrix multiplication
 for y.

    return weights, y_pred
```

In [12]:
```
weights, y_pred = OLS_regression(X_test, X_train, y_train)
plot_regression_results(y_test, y_pred, weights)
```

```
MSE:  599.7397626108208
r^2:  0.5211810643390854
```



What do you observe? Is the linear regression model good?

YOUR ANSWER HERE

The linear regression model is definitely good. In contrast to the linear regression model example in Figure 3 where the high value of MSE and the negative r^2 value indicated that the linear regression model was worse that just a representative horizontal line at $y = 0$, we can actually make positive interpretations from Figure 4 pertaining to why the linear regression model for our data works well. The interpretations are as follows:

1. The value of our MSE is much lower than that in the example illustrated by Figure 3. An r^2 value of 0.5212 indicates that approximately 52.12 percent of our variance is explained by our linear regression model.
2. The True vs. Predicted plot shows us that we can possibly have a regression line joining $(0, 0)$ and $(150, 100)$.
3. The bell shaped plot of Raw Residuals indicate an underlying Normal distribution with mean zero and a finite variance.

All the points mentioned above imply that the linear regression model is good. However, there is one possible qualm that comes out of the Weight Histogram plot which is as follows: There are between $50 - 60$ weights with value close to zero. This means that we are likely overfitting the data and would eventually need to regularize with ridge regression and penalize some weights.

Task 3: Compare your implementation to sklearn [5 pts]

Now, familarize yourself with the sklearn library. In the section on linear models:

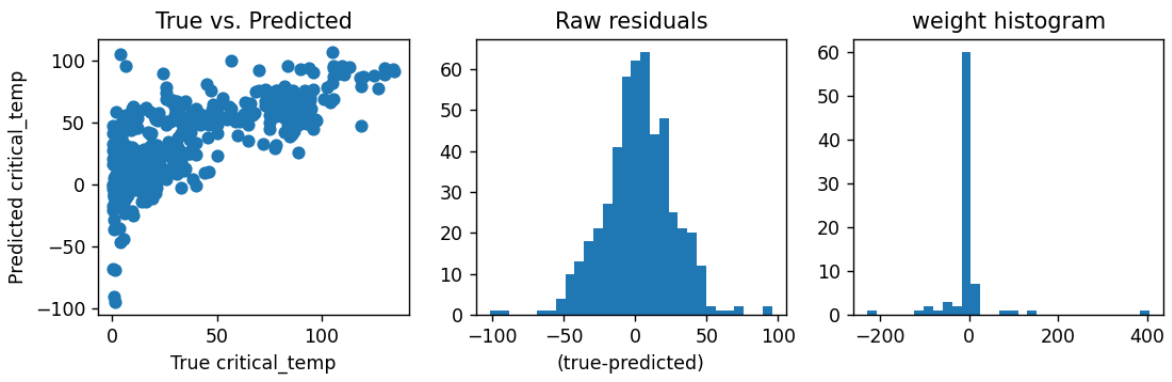
https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model)

you will find `sklearn.linear_model.LinearRegression` , the `sklearn` implementation of the OLS estimator. Use this sklearn class to implement OLS linear regression. Again obtain estimates of the weights on `X_train` and `y_train` and compute the MSE and r^2 on `X_test` .

In [13]:
```
def sklearn_regression(X_test, X_train, y_train):
    '''Computes OLS weights for linear regression without regularization using
the sklearn library on the training set and
       returns weights and testset predictions.

       Inputs:
         X_test: (n_observations, 81), numpy array with predictor values of th
e test set
         X_train: (n_observations, 81), numpy array with predictor values of t
he training set
         y_train: (n_observations,) numpy array with true target values for th
e training set

       Outputs:
         weights: The weight vector for the regerssion model including the off
set
         y_pred: The predictions on the TEST set


       Note:
         The sklearn library automatically takes care of adding a column for t
he offset.

    '''

    # ADD YOUR CODE HERE
    reg = linear_model.LinearRegression().fit(X_train, y_train)
    weights = np.transpose(reg.coef_)
    y_pred = reg.predict(X_test)

    return weights, y_pred
```

In [14]:
```
weights, y_pred = sklearn_regression(X_test, X_train, y_train)
plot_regression_results(y_test, y_pred, weights)
```

```
MSE:  599.7397626182633
r^2:  0.5211810643331436
```

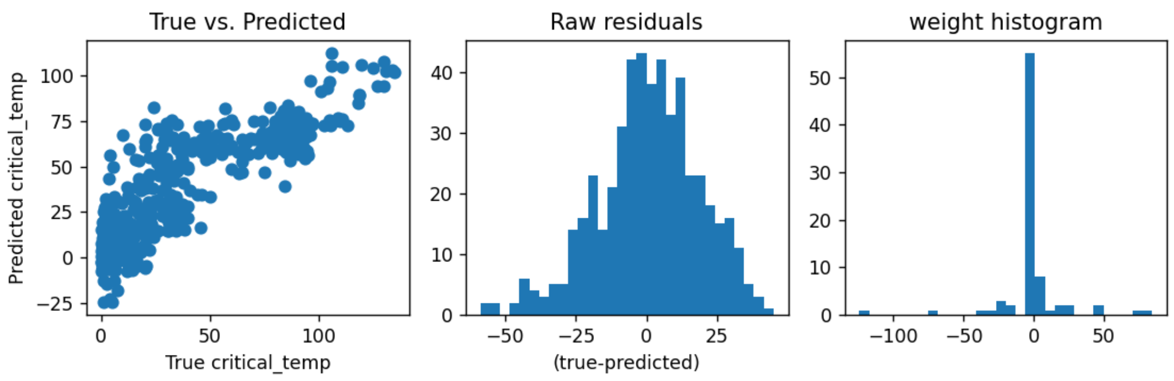


If you implemented everything correctly, the MSE is again 599.74.

Fit the model using the larger training set, `X_train_full` and `y_train_full` , and again evaluate on `X_test` .

In [15]:
```
weights, y_pred = sklearn_regression(X_test, X_train_full, y_train_full)
plot_regression_results(y_test, y_pred, weights)
```

```
MSE:  329.8607010410344
r^2:  0.7366465263179107
```



How does test set performance change? What else changes?

YOU ANSWER HERE

Our performance increases as seen from the lower MSE value when using the larger training set. A higher r^2 value of 0.7366 implies that 73.66 variance is explained by our linear regression model which is respectable. The True vs. Predicted plot show an underlying linearity. The residuals plot reveal a more symmetric bell shaped curve, i.e. Normal distribution centered around zero with finite variance. The weight histogram shows a slight improvement with a fewer amount of weights around zero in comparison to that of the smaller training set. However, the tendency to overfit still remains. We need regularization with ridge regression to address this concern, as also seen in the case with smaller training set.

All of the above results are as expected as we are training our linear regression model using more data, resulting in better estimates of weights.

Task 4: Regularization with ridge regression [15 pts]

We will now explore how a penalty term on the weights can improve the prediction quality for finite data sets. Implement the analytical solution of ridge regression

$$w = (X^T X + \alpha I_D)^{-1} X^T y$$

as a function that can take different values of α, the regularization strength, as an input. In the lecture, this parameter was called λ, but this is a reserved keyword in Python.

In [16]:
```
def ridge_regression(X_test, X_train, y_train, alpha):
    '''Computes OLS weights for regularized linear regression with regularizat
ion strength alpha
       on the training set and returns weights and testset predictions.

       Inputs:
         X_test: (n_observations, 81), numpy array with predictor values of th
e test set
         X_train: (n_observations, 81), numpy array with predictor values of t
he training set
         y_train: (n_observations,) numpy array with true target values for th
e training set
         alpha: scalar, regularization strength

       Outputs:
         weights: The weight vector for the regerssion model including the off
set
         y_pred: The predictions on the TEST set

       Note:
         Both the training and the test set need to be appended manually by a
 columns of 1s to add
         an offset term to the linear regression model.

    '''

    # ADD YOUR CODE HERE
    # We first appended manually on both the training and test set by a column
s of 1s
    # to add an offset term to the linear regression model.
    # We do so in the exact manner as in the function OLS_regression.
    n1, m1 = X_train.shape
    n2, m2 = X_test.shape
    X_train_1 = np.ones((n1,1))
    X_test_1 = np.ones((n2,1))
    X_train_new = np.hstack((X_train_1, X_train))
    X_test_new = np.hstack((X_test_1, X_test))

    X_transpose = np.transpose(X_train_new) # This is X'

    # We initialize the matrices X'X, X'y and weights with the appropriate dim
ensions.
    X_tX = np.zeros((m1+1,m1+1))
    X_ty = np.zeros((m1+1))
    weights = np.zeros((m1+1))

    # Calculate X'X and X'y element-wise using dot product of relevent vectors
    for i in range(m1+1):
        for j in range(m1+1):
            X_tX[i,j] = np.dot(X_transpose[i,:],X_train_new[:,j])
        X_ty[i] = np.dot(X_transpose[i,:],y_train)

    # Add the regularization term as a diagonal matrix to X'X
    X_tX += alpha*np.diag(np.ones(m1+1))

    # Then take its inverse
    X_tX_inv = np.linalg.inv(X_tX)

    # Finally, compute the weight vector element-wise
    for i in range(m1+1):
        weights[i] = np.dot(X_tX_inv[i,:],X_ty)

    # and determine the predicted y-values using a simple matrix multiplicatio
n
    y_pred = np.matmul(X_test_new, weights)

    # P.S. y_pred could be found using another dot product operation over loop
s which would potentially
    # take as much time as a numpy matrix multiplication.

    return weights, y_pred
```

Now test a range of log-spaced αs (~10-20), which cover several orders of magnitude, e.g. from 10^-7 to 10^7.

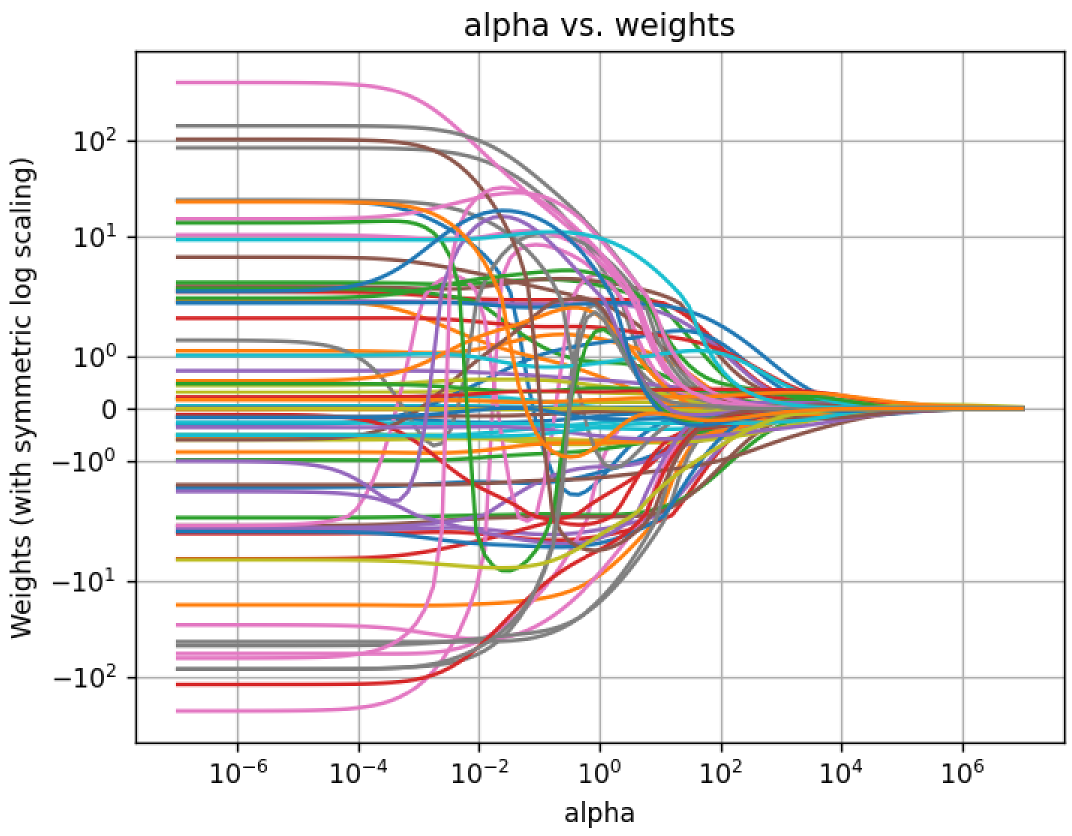
- For each α, you will get one model with one set of weights.
- For each model, compute the error on the test set.

Store both the errors and weights of all models for later use. You can use the function `mean_squared_error` from sklearn (imported above) to compute the MSE.

In [17]:
```
alphas = np.logspace(-7,7,100)

# ADD YOUR CODE HERE

# First we initialize an empty errors vector which will update and store the m
ean squared errors
errors = []

# Then for each alpha we have a column weight vector of length 82
# So, for 100 such alphas we have initialize a weight matrix of dimension (82,
100).
# This weight matrix will store along columns the computed weights per alpha.
weight_matrix = np.zeros((82,100))

# loop over the number of alphas
for i in range(len(alphas)):
    # call ridge_regression with appropriate alpha value and update the column
s of weight matrix
    weight_matrix[:,i], y_pred = ridge_regression(X_test, X_train, y_train, al
phas[i])
    # calculate the mean squared error and add the value to our errors vector
    errors.append(mean_squared_error(y_test,y_pred))
```

Make a single plot that shows for each coefficient how it changes with α, i.e. one line per coefficient. Also think about which scale (linear or log) is appropriate for your α-axis. You can set this using `plt.xscale(...)` .

In [18]:
```
# ADD YOUR CODE HERE
plt.figure()
plt.xscale('log')
plt.xlabel("alpha")
plt.yscale('symlog')
plt.ylabel("Weights (with symmetric log scaling)")
plt.title("alpha vs. weights")
for i in range(82):
    plt.plot(alphas, weight_matrix[i,:], )
    # Each row of weight matrix shows the change in its respective weight with
change in alpha

plt.grid(True)
plt.show()
```



Why are the values of the weights largest on the left? Do they all change monotonically?

YOUR ANSWER HERE

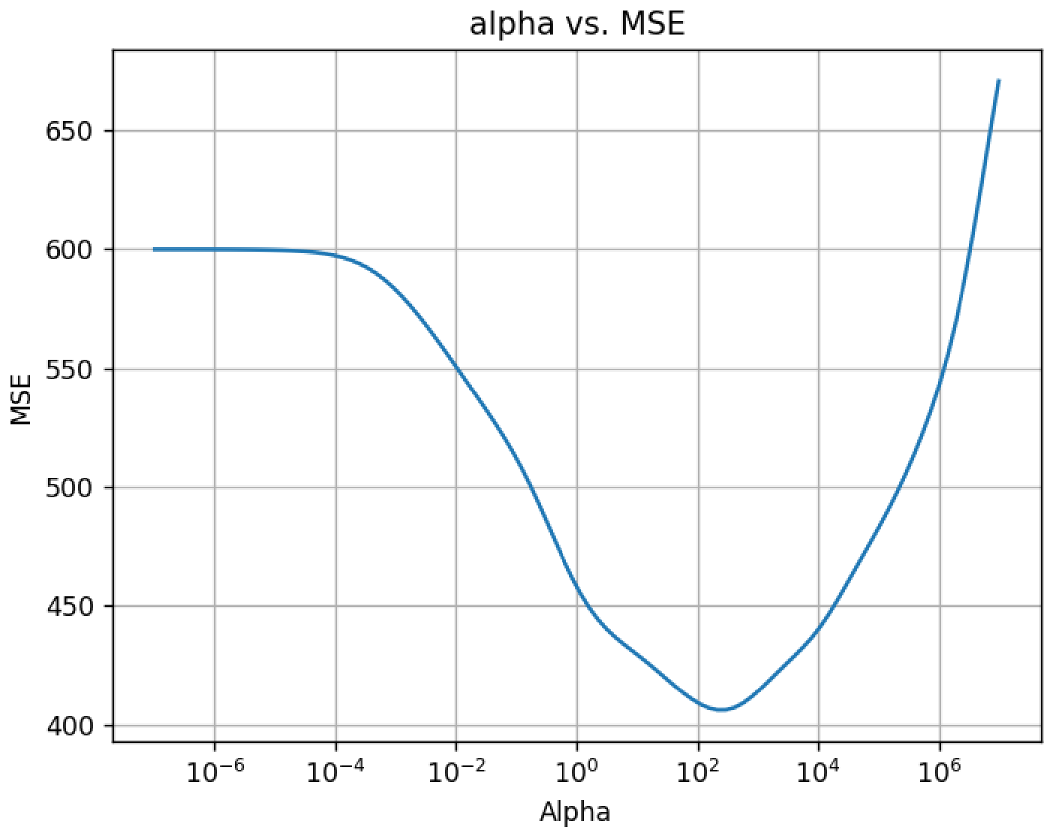
Each coloured curve in the above "alpha vs. weights" plot shows us how a single weight (or coefficient) changes with changes in α. The values of the weights are largest on the left because this portion of the plot shows the behaviour of the weights when the regularization strength is very small (or close to zero). In this region, the weights after regularization would still be close to those of without regularization. As we move towards the right and the regularization strength increases to higher orders of 10, the weights gradually get strongly penalized and become close to zero.

Although the net weight shows a monotonically decreasing behaviour and approaches zero, but the individual weights do not change monotonically as seen clearly from the plot. There are multiple weights that rise and fall (or fall and rise) i.e. have multiple local maxima(or minima) as α increases from 10^{-7} to 10^{7}.

The ideal weights would lie somewhere in between for some α in the range $[10^{-7}, 10^{7}]$.

Plot how the performance (i.e. the error) changes as a function of α. Again, use appropriate scaling of the x-axis. As a sanity check, the MSE value for very small αs should be close to the test-set MSE of the unregularized solution, i.e. 599.

In [19]:
```
# ADD YOUR CODE HERE
plt.figure()
plt.plot(alphas,errors)
plt.xscale('log')
plt.xlabel("Alpha")
plt.ylabel("MSE")
plt.title("alpha vs. MSE")
plt.grid(True)
plt.show()
```



Which value of α gives the minimum MSE? Is it better than the unregularized model? Why should the curve reach ~600 on the left?

YOUR ANSWER HERE

In [20]:
```
min_MSE = np.min(errors)
print("Minimum MSE is:", min_MSE)
print("Corresponding alpha value is:", alphas[errors.index(min(errors))])
```

```
Minimum MSE is: 406.2638214315345
Corresponding alpha value is: 215.44346900318777
```

The value of $\alpha = 215.44$ gives the minimum MSE of 406.26. This is certainly better than the unregularized model which has an MSE of 599.74 ~600.

The reason why the plot of MSE reaches ~600 or, in other words, close to the MSE value of the unregularized model, is that α values are significantly small towards the left. This implies that the regularization strength is highly diminished and the values of weights become closer to the weights in the unregularized model. So the MSE value in the regularized model with very small regularization strength also approaches the MSE value of the unregularized model which is ~600.

Further, we see that as α increases, the MSE curve dips to a minima for an ideal regularization strength, α of 215.44. But as we keep increasing α the weights starts approaching zero as they are heavily penalized and gradually become redundant. This results in an increasing MSE towards the right.

Now implement the same model using sklearn. Use the `linear_model.Ridge` object to do so.

In [21]:
```
def ridge_regression_sklearn(X_test, X_train, y_train,alpha):
    '''Computes OLS weights for regularized linear regression with regularizat
ion strength alpha using the sklearn
       library on the training set and returns weights and testset prediction
s.

       Inputs:
         X_test: (n_observations, 81), numpy array with predictor values of th
e test set
         X_train: (n_observations, 81), numpy array with predictor values of t
he training set
         y_train: (n_observations,) numpy array with true target values for th
e training set
         alpha: scalar, regularization strength

       Outputs:
         weights: The weight vector for the regerssion model including the off
set
         y_pred: The predictions on the TEST set

       Note:
         The sklearn library automatically takes care of adding a column for t
he offset.


    '''

    # ADD YOUR CODE HERE

    # We write the code in a manner that will allow us to pass a vector of alp
has and not just a single alpha
    a = alpha # store alpha in 'a' to avoid conflict with the alpha in linear_
model.Ridge
    m = len(alpha) # length of our alpha vector
    n, p = X_train.shape # dimension of X_train including the offset

    # Now we initialize our weights matrix such that for each i-th alpha, the
 weights are stored in the i-th column
    weights = np.ones((p,m))

    # For simplicity, we fit linear ridge model for our first alpha
    i = 0
    lmr = linear_model.Ridge(alpha = a[i]) # call linear_model.Ridge from skle
arn
    lmr.fit(X_train, y_train) # fit the training set
    weights[:,i] = lmr.coef_ # update the first weight column and
    y_pred = lmr.predict(X_test) # get predicted y-values as column vector

    # Now we loop over the remaining alphas
    for i in range(1,m):
        lmr = linear_model.Ridge(alpha = a[i])
        lmr.fit(X_train, y_train)
        weights[:,i] = lmr.coef_
        y_pred = np.hstack((y_pred,lmr.predict(X_test))) # we use hstack to st
ack the new predictions columnwise
        # This gives us a y_pred matrix where for each i-th alpha, the predict
```

```
ions are stored along the i-th column

    return weights, y_pred
```

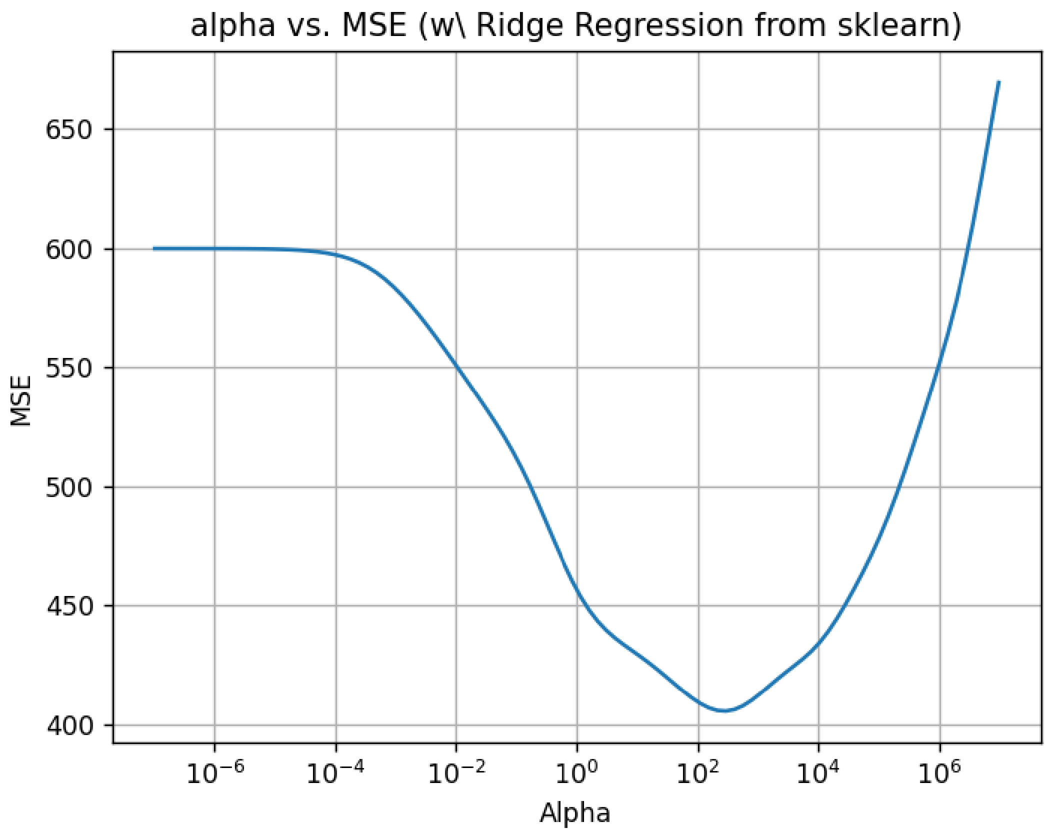
In [22]:
```
weights, y_pred = ridge_regression_sklearn(X_test, X_train, y_train,alphas)
```

In [23]:
```
# We check the correctness of our dimensions
weights.shape, y_pred.shape
```

Out[23]:
```
((81, 100), (500, 100))
```

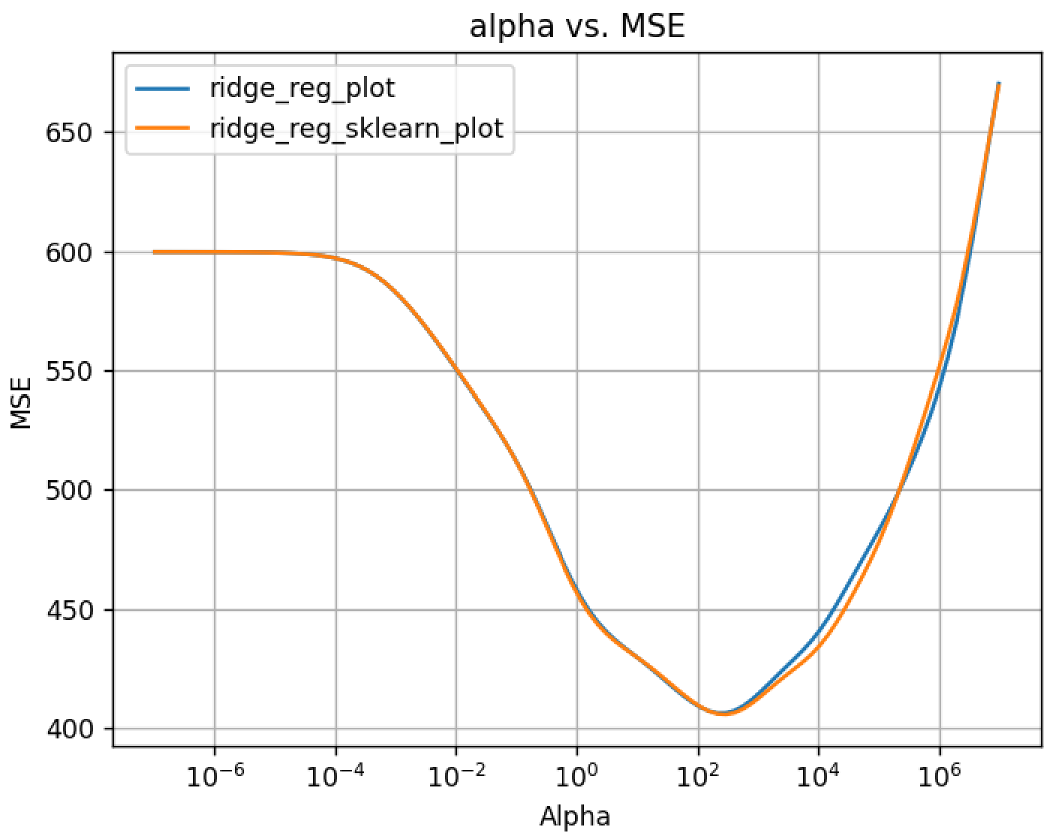
This time, only plot how the performance changes as a function of α.

In [24]:
```
# ADD YOUR CODE HERE
errors_rreg_sklearn = [] # initialize an empty errors vector

# We now loop over our alphas
for i in range(len(alphas)):
    # each loop calculates the mean squared error between y-test and predicted
y-value specific to the alpha
    # then appends the errors vector
    errors_rreg_sklearn.append(mean_squared_error(y_test,y_pred[:,i]))
plt.figure()
plt.plot(alphas,errors_rreg_sklearn)
plt.xscale('log')
plt.xlabel("Alpha")
plt.ylabel("MSE")
plt.title("alpha vs. MSE (w\ Ridge Regression from sklearn)")
plt.grid(True)
plt.show()
```



Note: Don't worry if the curve is not exactly identical to the one you got above. The loss function we wrote down in the lecture has α defined a bit differently compared to sklearn. However, qualitatively it should look the same.

In [25]:
```
# We check the differences and similarities in the curves here in conformation
with the above Note.
plt.figure()
ridge_reg_plot = plt.plot(alphas,errors)
ridge_reg_sklearn_plot = plt.plot(alphas,errors_rreg_sklearn)
plt.xscale('log')
plt.xlabel("Alpha")
plt.ylabel("MSE")
plt.legend(["ridge_reg_plot","ridge_reg_sklearn_plot"])
plt.title("alpha vs. MSE")
plt.grid(True)
plt.show()
```



We indeed see that the two curves are not exactly identical but qualitatively similar as mentioned in the note.

Task 5: Cross-validation [15 pts]

Until now, we always estimated the error on the test set directly. However, we typically do not want to tune hyperparameters of our inference algorithms like α on the test set, as this may lead to overfitting. Therefore, we tune them on the training set using cross-validation. As discussed in the lecture, the training data is here split in `n_folds` -ways, where each of the folds serves as a held-out dataset in turn and the model is always trained on the remaining data. Implement a function that performs cross-validation for the ridge regression parameter α. You can reuse functions written above.

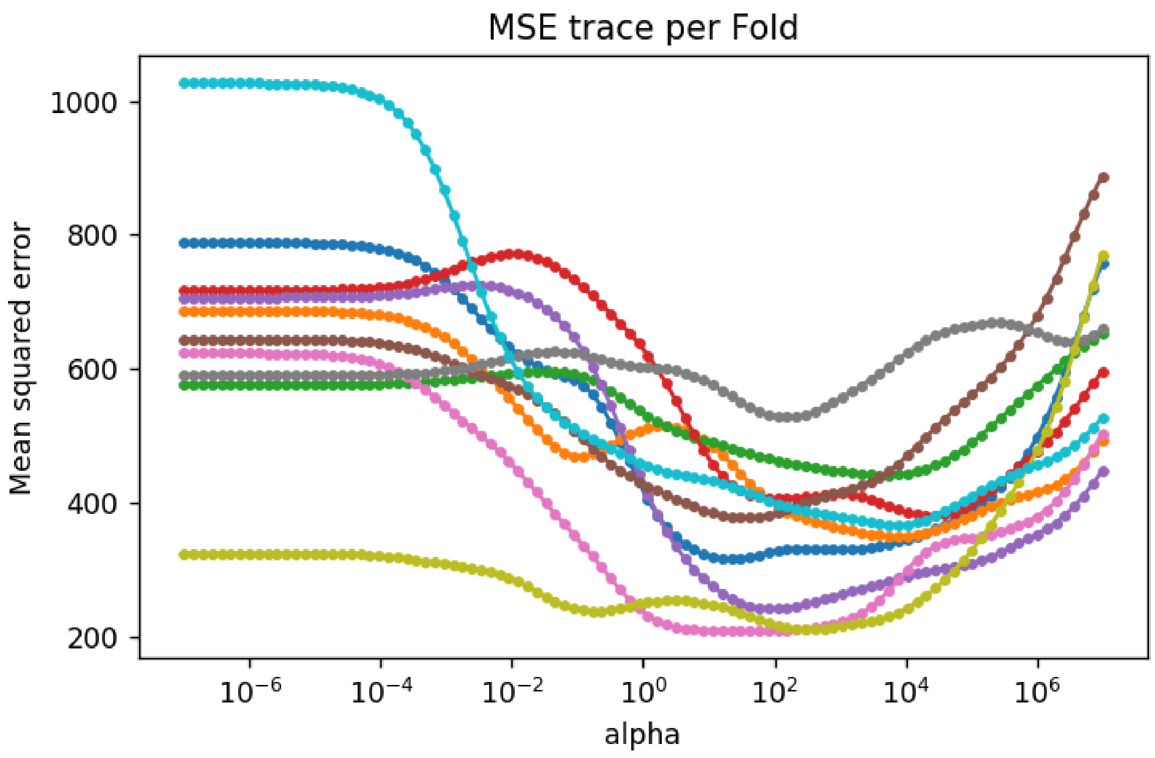
In [26]:
```
def ridgeCV(X, y, n_folds, alphas):
    '''Runs a n_fold-crossvalidation over the ridge regression parameter alph
a.
       The function should train the linear regression model for each fold on
 all values of alpha.

      Inputs:
        X: (n_obs, n_features) numpy array - predictor
        y: (n_obs,) numpy array - target
        n_folds: integer - number of CV folds
        alphas: (n_parameters,) - regularization strength parameters to CV ove
r

      Outputs:
        cv_results_mse: (n_folds, len(alphas)) numpy array, MSE for each cross
-validation fold

      Note:
        Fix the seed for reproducibility.

        '''

    cv_results_mse = np.zeros((n_folds, len(alphas)))
    np.random.seed(seed=2)


    # ADD YOUR CODE HERE

    # We see that the function ridgeCV only returns the MSE per fold per alph
a.
    # This simplifies our code as we do not need to initialize our weights and
y_pred matrices before hand
    nx, px = X.shape # dimension of X
    ny, py = y.shape # dimension of y (we do not essentially require this as t
he row sizes will be same for X and y)

    # We calculate on the size of the block that will act as the test set insi
de each fold, given that we have n folds.
    block_size = (n_folds/100)*nx

    # we now loop over the number of folds
    for fold in range(1,n_folds+1):
        start = int((n_folds-fold)*block_size) # starting row index for the te
st set inside the fold
        end = int((n_folds-fold+1)*block_size) # ending row index for the test
set inside the fold
        X_test = X[start:end,:] # We select the X_test from the training set i
nside the fold
        y_test = y[start:end,:] # We select the corresponding y_test values fr
om the y set inside the fold

        # Then the portion of the data left after selecting X_test and y_test
 are clubbed to form the X_train and y_train
        X_train = np.vstack((X[0:start,:],X[end:nx,:]))
        y_train = np.vstack((y[0:start,:],y[end:nx,:]))
        # Now run ridge regression from sklearn using the previously defined f
unction
        weights, y_pred = ridge_regression_sklearn(X_test, X_train, y_train, a
lphas)

        # Create empty errors vector for each fold and append the mean squared
errors per fold for each alpha
        errors = []
        for i in range(len(alphas)):
            errors.append(mean_squared_error(y_test,y_pred[:,i]))

        # Finally update the i-th row of cv_results_mse with the errors.
        # This i-th row will contain the MSEs from the i-th fold corresponding
to different alpha values.
        cv_results_mse[fold-1,:] = errors


    return cv_results_mse
```

Now we run 10-fold cross-validation using the training data of a range of αs.

In [27]:
```
alphas = np.logspace(-7,7,100)
mse_cv = ridgeCV(X_train, y_train, n_folds=10, alphas=alphas)
```

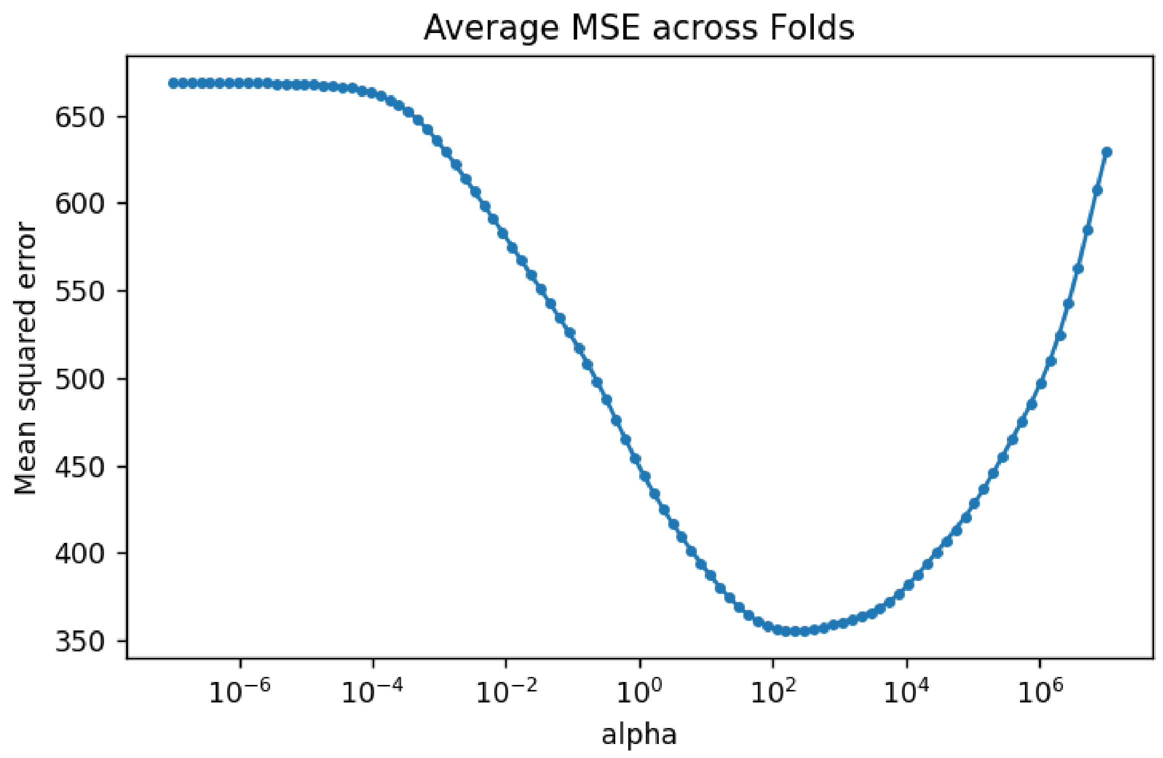
We plot the MSE trace for each fold separately:

In [28]:
```
plt.figure(figsize=(6,4))
plt.plot(alphas, mse_cv.T, '.-')
plt.xscale('log')
plt.xlabel('alpha')
plt.ylabel('Mean squared error')
plt.title('MSE trace per Fold')
plt.tight_layout()
```



We also plot the average across folds:

In [29]:
```
plt.figure(figsize=(6,4))
plt.plot(alphas, np.mean(mse_cv,axis=0), '.-')
plt.xscale('log')
plt.xlabel('alpha')
plt.ylabel('Mean squared error')
plt.title('Average MSE across Folds')
plt.tight_layout()
```



What is the optimal α? Is it similar to the one found on the test set? Do the cross-validation MSE and the test-set MSE match well or differ strongly?

In [30]:
```
mean_mse_cv = np.mean(mse_cv,axis=0)
min_mse_cv = np.amin(mean_mse_cv)
min_index = np.where(mean_mse_cv == np.amin(mean_mse_cv))
print('The minimum MSE from 10-folds crossvalidation is',min_mse_cv)
print('The optimal alpha is:',alphas[min_index])
```

```
The minimum MSE from 10-folds crossvalidation is 355.3370369076891
The optimal alpha is: [215.443469]
```

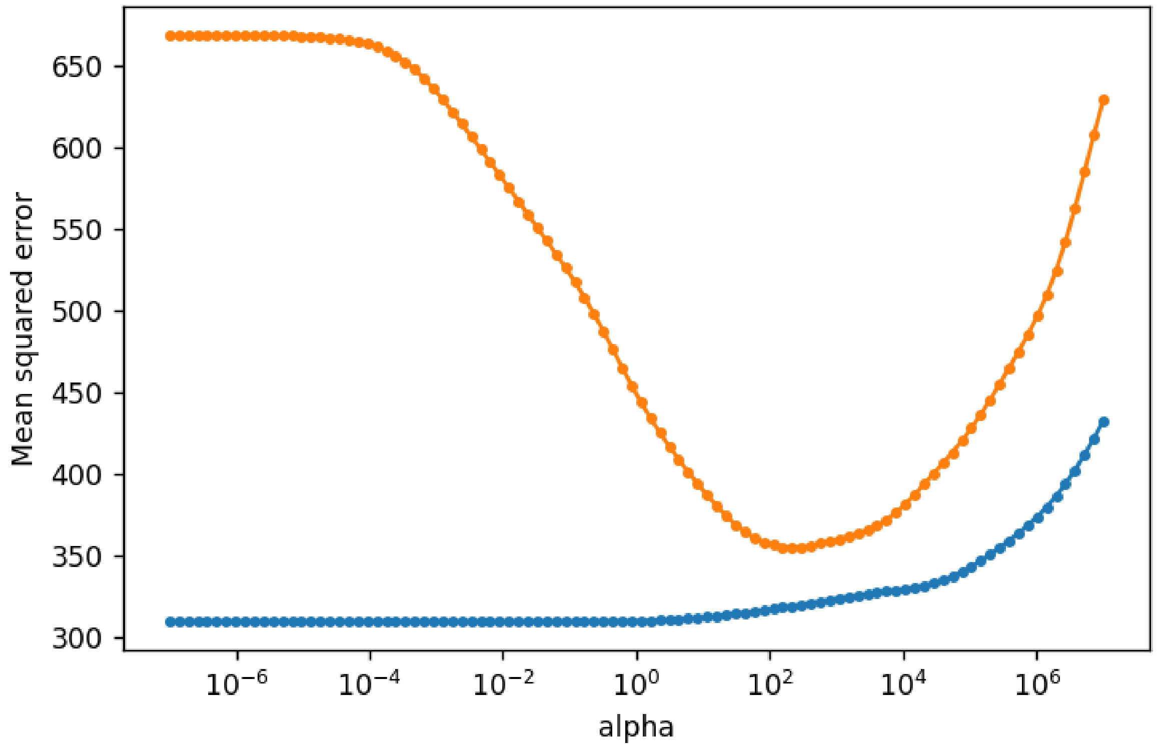
YOUR ANSWER HERE

The optimal α is 215.44. This is similar to the one found on the test set. However, the cross-validation MSE (355.34) is much smaller than the test-set MSE (406.26). So, they differ strongly.

We will now run cross-validation on the full training data. This will take a moment, depending on the speed of your computer. Afterwards, we will again plot the mean CV curves for the full data set (blue) and the small data set (orange).

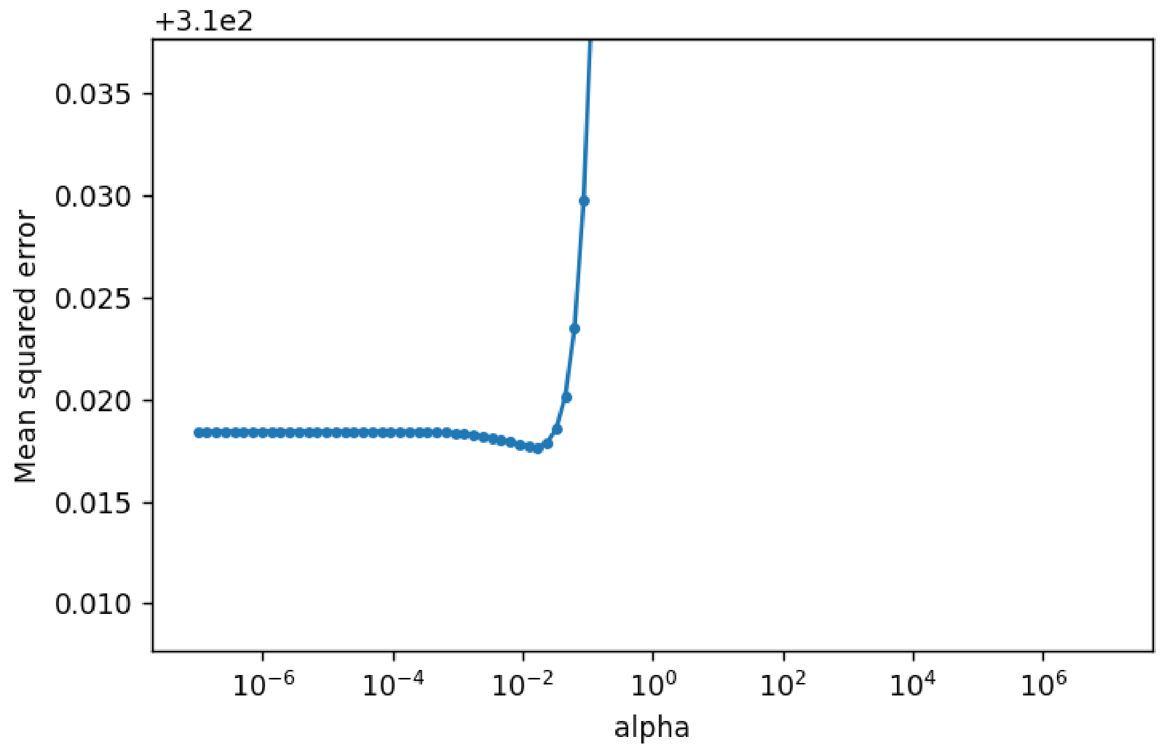
In [31]:
```
alphas = np.logspace(-7,7,100)
mse_cv_full = ridgeCV(X_train_full, y_train_full, n_folds=10, alphas=alphas)
```

In [32]:
```
plt.figure(figsize=(6,4))
plt.plot(alphas, np.mean(mse_cv_full,axis=0), '.-')
plt.plot(alphas, np.mean(mse_cv,axis=0), '.-')
plt.xscale('log')
plt.xlabel('alpha')
plt.ylabel('Mean squared error')
plt.tight_layout()
```



We determine the (very subtle) minimum of the blue curve and zoom in around it to be able to see it:

In [33]:
```
plt.figure(figsize=(6,4))
plt.plot(alphas, np.mean(mse_cv_full,axis=0), '.-')
plt.xscale('log')
minValue = np.min(np.mean(mse_cv_full,axis=0))
plt.ylim([minValue-.01, minValue+.02])
plt.xlabel('alpha')
plt.ylabel('Mean squared error')
plt.tight_layout()
```



Why does the CV curve on the full data set look so different? What is the optimal value of α and why is it so much smaller than on the small training set?

In [34]:
```
mean_mse_cv_full = np.mean(mse_cv_full,axis=0)
min_index_full = np.where(mean_mse_cv_full == np.amin(mean_mse_cv_full))
print('The minimum MSE from 10-folds crossvalidation using the full training s
et is',minValue)
print('The optimal alpha using the full training set with cross validation i
s:',alphas[min_index_full])
```

```
The minimum MSE from 10-folds crossvalidation using the full training set is
310.01766033547494
The optimal alpha using the full training set with cross validation is: [0.01
707353]
```

YOUR ANSWER HERE A short directed answer would be: more data -> more precise estimates -> less sampling error->less overfitting -> less punishment

In a bit more detail, we would have the following explanation: We have more data points while implementing Cross Validation on the full training data set. This results in more precise estimates of weights and less sampling error. So even for very low regularization strength our MSE vs. α curve is lower and flatter in comparison to when we use Cross Validation with ridge regression on the smaller training set.

Because we have low sampling error with the full training data set, we also have less overfitting. Consequentially, our optimal α using the full training set with cross validated ridge regression becomes much smaller than on the small training set as we do not need to punish our estimated weights and require only slight regularization.

However, as we continue to increase the regularization strength in the CV ridge regression on full training data, we begin to unnecessarily punish the estimated weights and the MSE starts increasing after the local point of minima. Because we already had good estimates of weights, so our overall MSE remains lower for the full data in comparison to the smaller data. But qualitively the two curves start to look similar for large α values.

```
In [ ]:
```