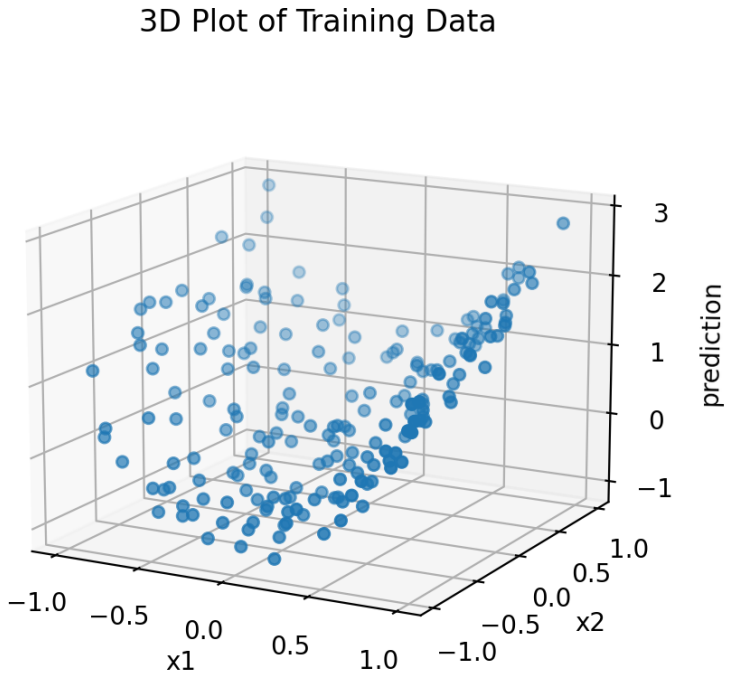
# CS7CS4/CSU44061 Machine Learning: Week 3 Assignment

Cian Higgins 29 October 2020

**# id: 8-16-8**

**(a)(i)** Data was read in with pandas and NumPy using the same code as for the previous two assignments. The data was then plotted as a 3D scatter plot with the first feature labelled '*x1*', second feature labelled '*x2*', and the target *y* values labelled 'prediction'. The matplotlib scatter function was used.




The training data appears to be lying on a curve. Upon inspection of the data, there are angles at which the data looks like a U-shaped cross section.

**(i)(b)** A short function was written that takes the current two features as input and adds extra polynomial features i.e. all combinations of powers of the two features up to the power of five. A new array of twenty-one features is returned.

```
def make_poly_features(x):
  poly = PolynomialFeatures(degree=5)
  new_x = poly.fit_transform(x)
  return new_x
```

These polynomial features were then used to train lasso regression models with a wide range of C values. The range of C values used was from one to ten thousand. Lasso regression uses L1 regularisation that adds the absolute value of the magnitude of coefficients. Adding this as a penalty term to the cost function means L1 can shrink features to zero which effectively eliminates them. In other words, specific features can be selected from a large group. To

demonstrate this, a small C was chosen which sets all of the trained model's parameters to zero. A value of C=1 worked for this.

```
polyX = make_poly_features(X)
C_params = [1, 10, 100, 1000, 10000]
lasso_models = []
for C in C_params:
  alpha = 1 / (2 * C)
  lasso = Lasso(alpha=alpha)
  lasso.fit(polyX, y)
  lasso_models.append(lasso)
  print(f"C={C},alpha={alpha},intercept={lasso.intercept_},coefs={lasso.coef_}")
```

**Parameter reporting for *lasso models* with varying C value:**

| | | |
|---|---|---|
| **C**=1 | **alpha**=0.5 | **intercept**=0.5826239294516646 |
| **coefs**=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] | | |
| **C**=10 | **alpha**=0.05 | **intercept**=0.16233727120458935 |
| **coefs**=[0, 0, 0.82752488, 1.4878169, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] | | |
| **C**=100 | **alpha**=0.005 | **intercept**=0.0007961265934705031 |
| **coefs**=[0, -0.00894803, 0.96801671, 1.96981562, 0, 0.04205435, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] | | |
| **C**=1000 | **alpha**=0.0005 | **intercept**=-0.0312414178102407 |
| **coefs**=[0, -0.09448614, 1.09351592, 2.03906152, 0.00219519, 0.07695824, 0, 0, 0.18077365, -0.22172446, -0.01161905, 0.03001595, 0, 0, 0, 0.02197548, -0.19293302, 0, 0.36454576, 0, 0] | | |
| **C**=10000 | **alpha**=0.00005 | **intercept**=-0.047234444757519034 |
| **coefs**=[0, -0.01586776, 1.27789678, 2.19383562, -0.02667148, 0.07452678, -0.4458174, -0.01032642, 0.16344467, -0.95675615, -0.18976269, 0.11428838, -0.01353057, -0.01290002, 0.00657811, 0.44030125, -0.46097273, 0.05354609, 0.72762214, 0.02738836, 0.54718837] | | |

As the C value increases, the alpha value decreases because $\alpha = 1/(2C)$ in sklearn. The increase in the C value makes the penalty term smaller, which allows more of the features to be assigned coefficients. Lower C values make the penalty term bigger which shrinks the less important feature coefficients to zero; thus removing features altogether. The models with very low C fail to capture the behaviour of the data properly and give poor predictions. This is underfitting. The models with very large C values are overfit i.e. the models have fit themselves to noise and so they generalise poorly.

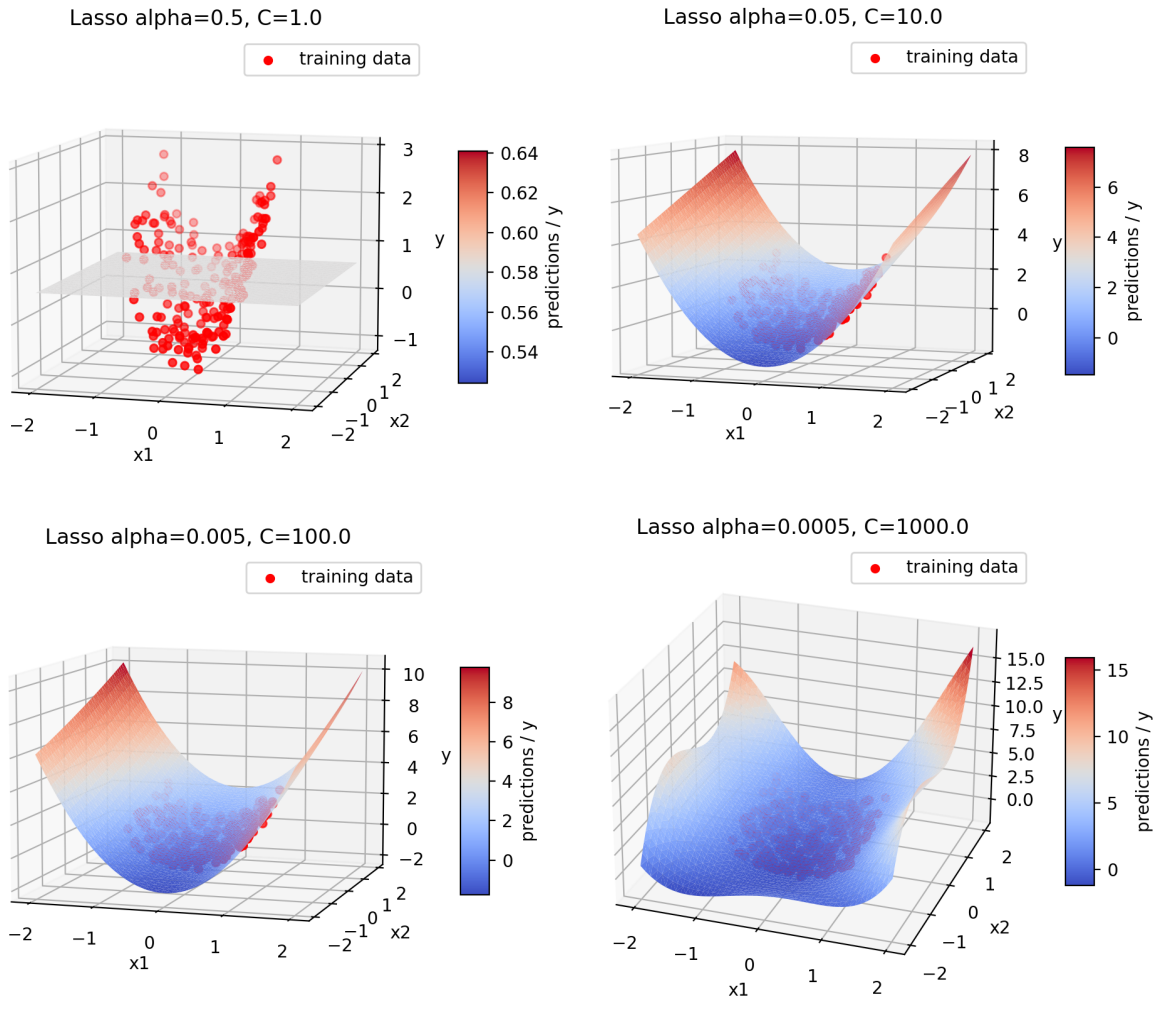
**(i)(c)** Predictions for the target variable were then generated for each of the previously trained models. The code given on the assignment sheet was used to produce a grid of feature values. The array of feature values was then passed to the function from part (i)(b) to calculate

its polynomial features. A reasonable grid range was chosen so that the training data could clearly be seen in the plots, as specified. The range $[-2,\ 2]$ was chosen.

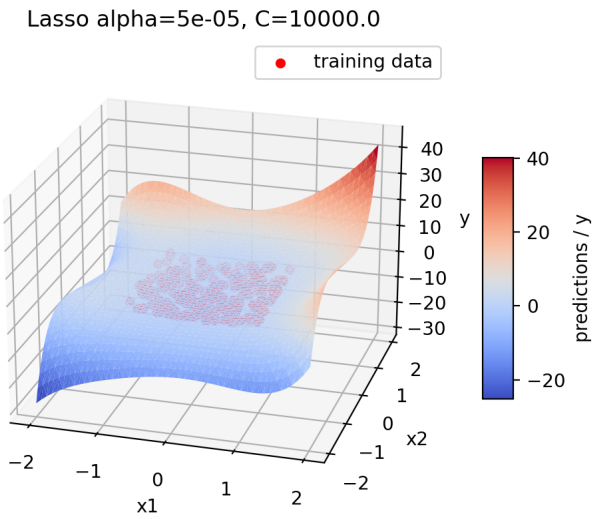
```
Xtest = []
grid = np.linspace(-2, 2)
for i in grid:
  for j in grid:
      Xtest.append([i, j])
Xtest = np.array(Xtest)

polyXtest = make_poly_features(Xtest)
for model in models:
  y_pred = model.predict(polyXtest)
```

These predictions were plotted as surfaces along with the training data. The plots:




The plots show that as C increases, the training data is fitted more closely and tightly. When C=1 and all coefficients are zero, the model is an underfit, very few of the points lie on the plane. When C=10 the model is a good fit (a curve) that also generalises well; in the case of C=100 the fit is also good, but the curve is steeper. When C reaches 1000 the model begins to overfit and fits closely to noise in the training data. It does not generalise well and performs badly with new data. For C=10000, the model is extremely overfit *(see overleaf).*




As mentioned, the training data is very closely fitted when C=10000. New data does not perform well on this model. The model makes predictions in an extreme range of -20 to 40. It does not do a good job of generalising the given training data.

**(i)(d)** ***Underfitting*** is when the model used is overly simplistic and as such cannot properly capture the behaviour of the data. Predictions on these models are poor. The model does not incorporate enough features to model the training data effectively, nor does it generalise well for new data. Underfitting is relatively easy to identify using performance metrics (accuracy, mean square error, etc.) as these will have poor results.

***Overfitting*** is when the model is too closely fitted to the training data. There comes a point when the model starts to learn the detail and noise in the data. This results in poor generalisation which means performance on unseen data will be low. Overfitting is recognised when the model performs highly on training data, but badly on unseen test data.

Lasso and ridge regression models have the ***hyperparameter C*** which is utilised to limit and constrain how much detail the model learns. Choosing a good C value is intrinsic to supervised learning algorithms. A number of values can be cross validated to find the C value that strikes the right balance between over- and underfitting.

**(i)(e)** The same functions and code used at parts (b) and (c) were used for a range of ridge regression models, however a different range of C values were chosen to demonstrate a range of coefficient values for the ridge models with L2-regularisation.

**Parameter reporting for *ridge models* with varying C value:**

| **C=0.001** | **alpha=500** | **intercept=0.5632007703724272** |
|---|---|---|
| **coefs=[0, 0.0137402, 0.10577897, 0.07024486, -0.00076802, -0.00670964, 0.01165274, 0.02451149, 0.00502244, 0.06378183, 0.06028513, -0.00204984, 0.02015041, -0.00449762, -0.00462376, 0.01128194, 0.01136593, 0.0029058, 0.014666, 0.00310044, 0.04778585]** | | |
| **C=0.01** | **alpha=50** | **intercept=0.4039406962187021** |
| **coefs=[0, 0.01607182, 0.42685391, 0.44464955, -0.00325623, -0.01153924, 0.0318227, 0.07211634, 0.02248684, 0.21251729, 0.37272107, -0.01103207, 0.13354269, -0.02588964, -0.00072368, 0.04131494, 0.02116766, 0.01820276, 0.03047764, 0.01893788, 0.14582786]** | | |

**Parameter reporting for *ridge models* with varying C value, *continued*:**

| C=0.1 | alpha=5 | intercept=0.14329962818432168 |
|---|---|---|
| coefs=[0, -0.049201133, 0.79371043, 1.0538482, 0.022572868, -0.00025405302, -0.013352628, 0.098701171, 0.06621804, 0.15186267, 0.70346187, 0.017741352, 0.23046497, -0.0087724433, 0.015422233, 0.028092364, -0.0014840858, 0.031381711, 0.010207303, 0.058422679, 0.035000686] | | |
| C=1 | alpha=0.5 | intercept=0.017129879975055662 |
| coefs=[0, -0.071235882, 1.0369565, 1.6848395, -0.016370986, 0.050944725, -0.089945495, 0.076309811, 0.14456735, -0.13592085, 0.31629274, 0.054450572, 0.086616827, 0.02979, 0.0082762277, 0.089613388, -0.16417308, 0.013097099, 0.23021062, 0.040844685, -0.003427464] | | |
| C=10 | alpha=0.05 | intercept=-0.040114431403830864 |
| coefs=[0, -0.034711381, 1.2184414, 2.1259902, -0.02804907, 0.07758655, -0.35549988, 0.011045798, 0.17434108, -0.72435669, -0.11779976, 0.10816568, -0.0058240314, -0.0065854124, 0.0034886293, 0.3585502, -0.41499416, 0.042184602, 0.63686942, 0.016007094, 0.36996751] | | |
| C=100 | alpha=0.005 | intercept=-0.0476889006898491 |
| coefs=[0, 0.00028042197, 1.3020347, 2.2026412, -0.031416674, 0.071790299, -0.51317991, -0.04783455, 0.1386974, -1.0396382, -0.199862, 0.12811069, -0.019367395, -0.017876356, 0.012103412, 0.49677197, -0.45467441, 0.076450347, 0.77302475, 0.045986875, 0.61067393] | | |

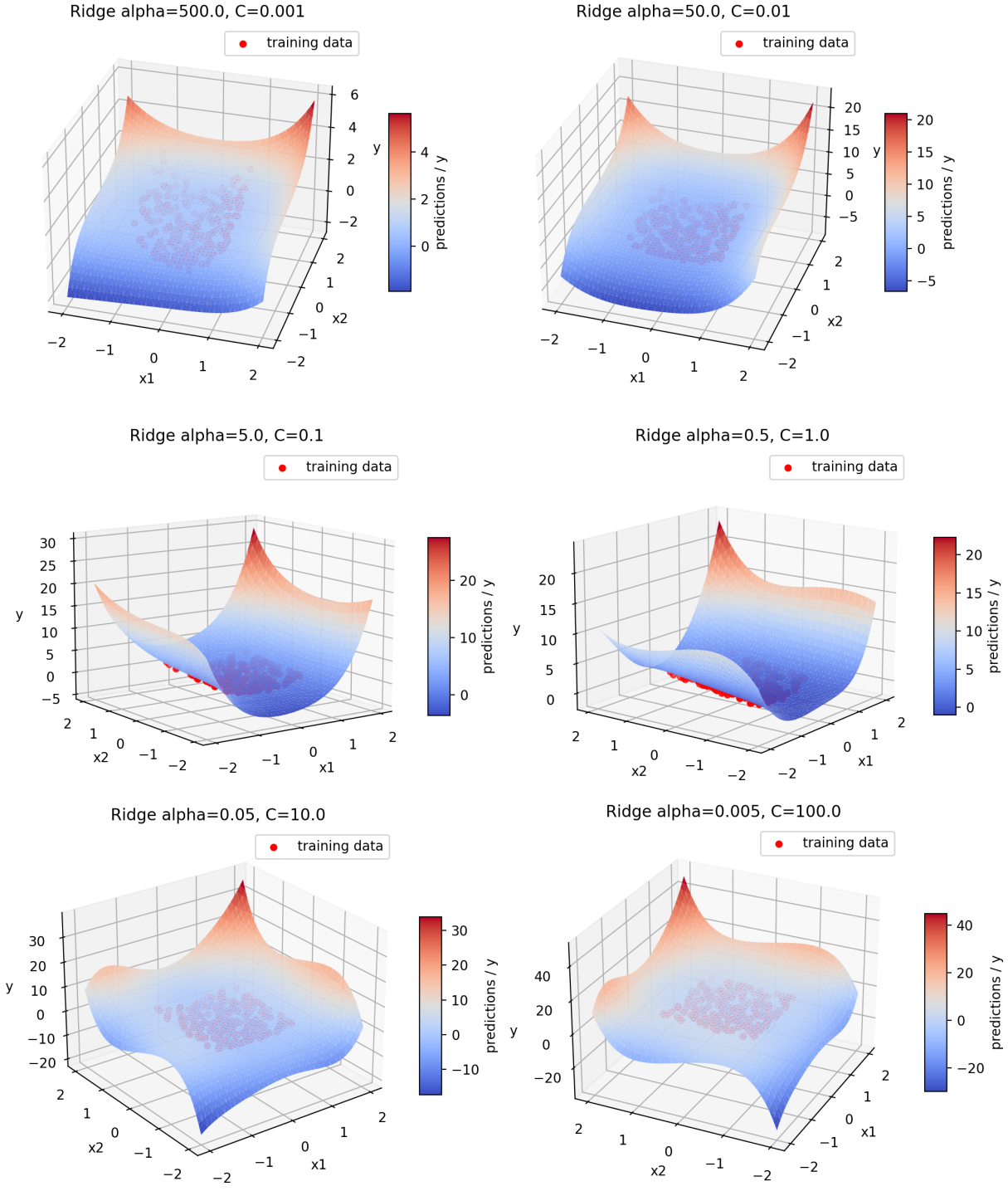
Code executed to obtain the above parameter values:

```
ridge_models = []
ridge_c_params = [0.001, 0.01, 0.1, 1, 10, 100]
for C in ridge_c_params:
  alpha = 1 / (2 * C)
  ridge = Ridge(alpha=alpha)
  ridge.fit(polyX, y)
  ridge_models.append(ridge)
```

It is clear from the reported values that ridge regression applies a different penalty than lasso. The penalty used for ridge is called quadratic or L2-regularisation. The C hyperparameter is once again used to control the effect of the regularisation. Note that even a very small C value does not cause feature sparsity – as is the case with L1 – but instead encourages the coefficients to have small values. Ridge regression only shrinks coefficients close to zero, but never exactly zero. Larger C values allow the model to assign large coefficients to certain parameters, e.g. when C=100, $\theta 3$ and $\theta 4$ have values 1.3 and 2.2, respectively. Larger coefficients mean these features have a greater influence on the model than others.

The ridge regression models were again plotted using matplotlib, *please see the next page.*

**Ridge regression plots:**




Smaller C values make for coefficients which are close to zero. The result of C=0.001 is a model with reduced complexity, and it appears to be a mostly flat plot, predictions lie mostly in the small range of 0 – 4. The models are underfit. As the C value increases the prediction range gets wider, at the same time the coefficients are less restricted (less L2 penalty) and the shapes look more complex. The models best generalise the training data when C is 0.1 and 1. Beyond this, plots begin to overfit the training data and predictions outside of the training range are assigned extreme positive and negative values.

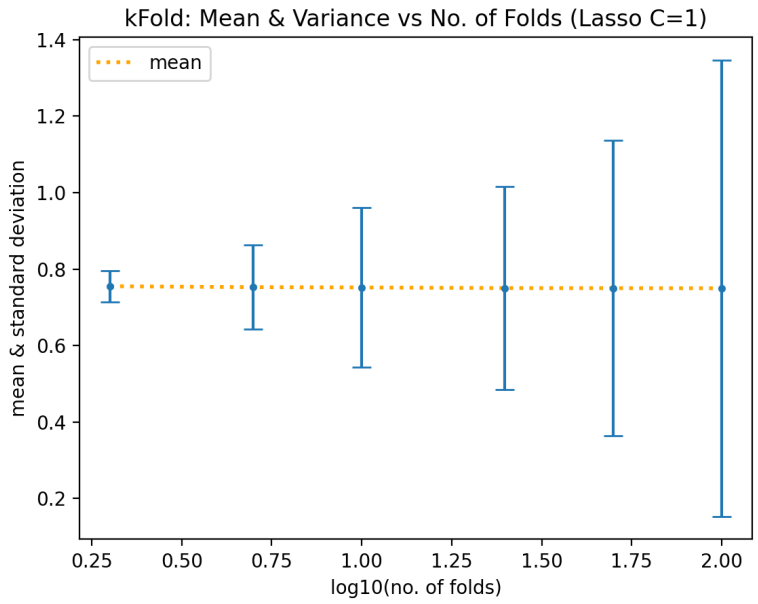
**ii)(a)** A function was defined that takes the number of folds, k, and a particular model and then computes the mean and variance of the prediction error (MSE) for each number of folds.

```
def k_fold_cross_val(k, model):
  k_fold = KFold(n_splits=k)
  sq_errs = []
  for train, test in k_fold.split(polyX):
      curr_model = model.fit(polyX[train], y[train])
      y_pred = curr_model.predict(polyX[test])
      sq_err = mean_squared_error(y[test], y_pred)
      sq_errs.append(sq_err)
  mean = np.mean(sq_errs)
  std = np.std(sq_errs)
  return mean, std
```

This function was used for a range of different folds and the returned means and standard deviations were then visualised with the matplotlib error bar plot function.

```
folds = [2, 5, 10, 25, 50, 100]
means = []
std_devs = []
for fold in folds:
  lassoC1 = Lasso(alpha=0.5)
  values = k_fold_cross_val(fold, lassoC1)
  means.append(values[0])
  std_devs.append(values[1])
```

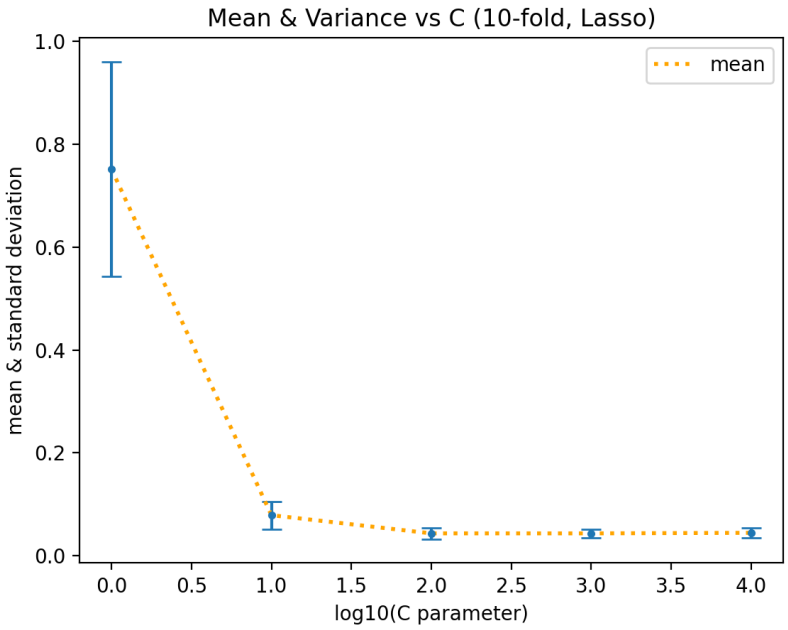
The error bar plot shows clearly that the variance increases as the number of folds increases. As the number of folds increases, the size of each sample taken from the whole dataset gets smaller. Within these smaller samples there is a larger spread and noise becomes more pronounced. More computational power is also needed as every sample is being fit to the model and predictions are also being made each time. Since the prediction mean stays constant regardless of the number of folds, a small number of folds should be chosen to save computation time and effort. We should choose k=10, as this gives us enough data to be sure of how robust the model and its predictions are. We also have a good ratio of training data to test data, 90% training and 10% test data on each fold. Although the variance isn't as small as possible, k=10 is small enough to average the results and smooth out noise and fluctuations.



**(ii)(b)** A for-loop was written to use the cross-validation function from part (a) on each lasso model. Reminder that the C values of the lasso models are `[1, 10, 100, 1000, 10000]`.

```
lasso_means = []
lasso_std_devs = []

for lasso_mod in lasso_models:
  results = k_fold_cross_val(10, lasso_mod)
  lasso_means.append(results[0])
  lasso_std_devs.append(results[1])
```




A wide range of C values was chosen to show the progression of the mean of the prediction error and the reduction in variance. The same values that were used for the plots were also used here so both could be inspected to make an informed decision on the C value.

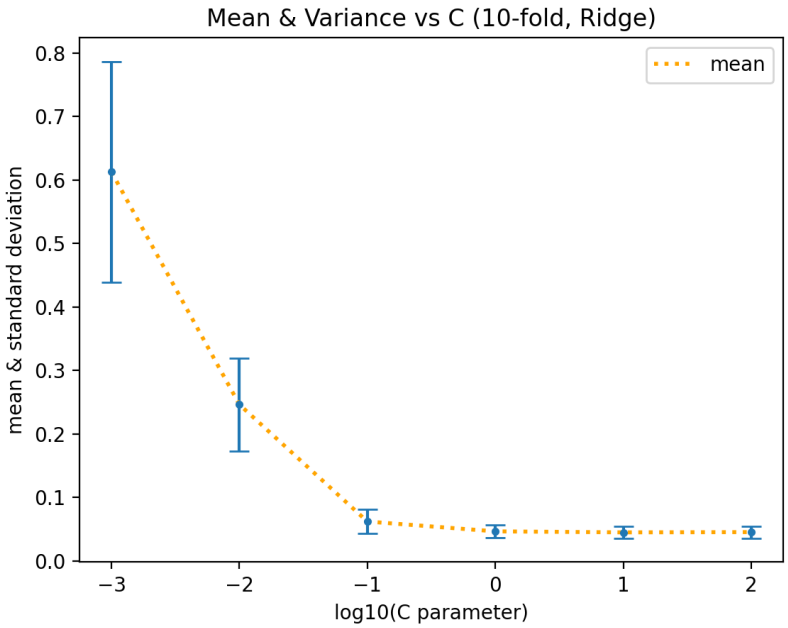
**(ii)(c)** Judging from the error bar plot, the value C=10 should be used in the model. The prediction error has a relatively small variance, but not too small to the point the model is overfit. To avoid overfitting, the simplest model possible i.e. the smallest C should be chosen. However, C=1 has a big variance and the mean of the prediction error is so high that it suggests the model is underfit. For this reason, we choose C=10 to try strike a balance between under- and overfitting – *bias-variance tradeoff*.

**(ii)(d)** Similarly, a for-loop was written to also perform cross-validation on the ridge models.

```
ridge_means = []
ridge_std_devs = []

for ridge_mod in ridge_models:
  results = k_fold_cross_val(10, ridge_mod)
  ridge_means.append(results[0])
  ridge_std_devs.append(results[1])
```

The means and variances of the prediction error were then plotted with matplotlib's error bar plot:




The range of values for C used `[0.001, 0.01, 0.1, 1, 10, 100]`. This range was chosen to give a good visual overview of the mean squared error. The smallest C value shows a wide variance, whereas the C values of 1, 10 and 100 have very small variance and similar means, this suggests that the model is overfitting with these C values.

Once again, a bias-variance tradeoff or a balance between under- and overfitting must be struck. The same strategy of choosing the smallest possible C value will be used, and if that value proves to be unsuitable, the next value up will be inspected. As mentioned, C=0.001 has a big variance and a high error rate (an underfit), so this will not be chosen. Next, C=0.01 is improved, but nevertheless has a high error rate and a big variance. Based off this, C=0.1 should be chosen as it has a respectable prediction error mean and the variance is not too large. The larger C values are clearly overfitting because the prediction error is incredibly low, and variance of the means is also very tight.

## Appendix

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

# read in data
df = pd.read_csv("week3.csv", comment='#', header=None)
print(df.head())
X1 = df.iloc[:, 0]
X2 = df.iloc[:, 1]
X = np.column_stack((X1, X2))
y = np.array(df.iloc[:, 2])

"""(i)(a) 3d plot
data lies on a curve - half cylinder """
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_title("3D Plot of Training Data")
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('prediction')
ax.scatter(X[:, 0], X[:, 1], y)
plt.show()


def make_poly_features(x):
  """
  (i)(b) makes polynomial features
  """
  poly = PolynomialFeatures(degree=5)
  new_x = poly.fit_transform(x)
  return new_x


""" train Lasso on a wide range of C values """
print("\n=== LASSO | L1 PENALTY ===")
polyX = make_poly_features(X)
C_params = [1, 10, 100, 1000, 10000]
lasso_models = []
for C in C_params:
  alpha = 1 / (2 * C)
  lasso = Lasso(alpha=alpha)
  lasso.fit(polyX, y)
  # pred = lasso.predict(polyX)
  lasso_models.append(lasso)
  print(f"C={C},alpha={alpha},intercept={lasso.intercept_},coefs={lasso.coef_}")


def test_plot_models(models):
  Xtest = []
  grid = np.linspace(-2, 2)  # adjusted grid to ensure training data can be seen
in plot
  for i in grid:
```

```
        for j in grid:
            Xtest.append([i, j])
    Xtest = np.array(Xtest)

    polyXtest = make_poly_features(Xtest)
    for model in models:
        y_pred = model.predict(polyXtest)
        fig1 = plt.figure()
        ax = fig1.add_subplot(111, projection='3d')
        ax.scatter(X[:, 0], X[:, 1], y, label='training data', color='red')
        surface = ax.plot_trisurf(Xtest[:, 0], Xtest[:, 1], y_pred, cmap='coolwarm')
        fig1.colorbar(surface, shrink=0.5, aspect=8, label='predictions / y')
        ax.set_xlabel('x1')
        ax.set_ylabel('x2')
        ax.set_zlabel('y')
        ax.legend()
        alpha_param = model.__getattribute__("alpha")
        c_param = 1 / (2 * alpha_param)
        model_name = type(model).__name__
        ax.set_title(f'{model_name} alpha={alpha_param}, C={c_param}')
        plt.show()


""" (i)(c) 3d plots of the models from prev part """
test_plot_models(lasso_models)

""" (i)(d) under- and over-fitting """

""" (i)(e) repeat b and c for ridge models """
print("\n=== RIDGE | L2 PENALTY ===")
ridge_models = []
ridge_c_params = [0.001, 0.01, 0.1, 1, 10, 100]
for C in ridge_c_params:
    alpha = 1 / (2 * C)
    ridge = Ridge(alpha=alpha)
    ridge.fit(polyX, y)
    ridge_models.append(ridge)
    print(f"C={C},alpha={alpha},intercept={ridge.intercept_},coefs={ridge.coef_}")

test_plot_models(ridge_models)

""" (ii)(a) k fold cross validation, C=1 """
def k_fold_cross_val(k, model):
    k_fold = KFold(n_splits=k)
    print(f"=== KFOLD k={k} ===")
    sq_errs = []
    for train, test in k_fold.split(polyX):
        curr_model = model.fit(polyX[train], y[train])
        y_pred = curr_model.predict(polyX[test])
        sq_err = mean_squared_error(y[test], y_pred)
        sq_errs.append(sq_err)
    mean = np.mean(sq_errs)
    std = np.std(sq_errs)
    print(f"mean={mean}, variance={std}")
    return mean, std


folds = [2, 5, 10, 25, 50, 100]
means = []
```

```
std_devs = []
for fold in folds:
  lassoC1 = Lasso(alpha=0.5)
  values = k_fold_cross_val(fold, lassoC1)
  means.append(values[0])
  std_devs.append(values[1])

fig2 = plt.figure()
plt.errorbar(np.log10(folds), means, yerr=std_devs, fmt='.', capsize=5)
plt.plot(np.log10(folds), means, linestyle=':', label='mean', linewidth=2,
color='orange')
plt.legend()
plt.title("kFold: Mean & Variance vs No. of Folds (Lasso C=1)")
plt.xlabel("log10(no. of folds)")
plt.ylabel("mean & standard deviation")
plt.show()
"""add legend! labels titles etc to all plots"""

""" (ii)(b) LASSO 10-fold cross validation to choose C-parameter """
# models with C_params
lasso_means = []
lasso_std_devs = []
for lasso_mod in lasso_models:
  results = k_fold_cross_val(10, lasso_mod)
  lasso_means.append(results[0])
  lasso_std_devs.append(results[1])

c_logs = np.log10(C_params)

plt.errorbar(np.log10(C_params), lasso_means, yerr=lasso_std_devs, fmt='.',
capsize=5)
plt.plot(c_logs, lasso_means, linestyle=':', label='mean', linewidth=2,
color='orange')
plt.legend()
plt.title("Mean & Variance vs C (10-fold, Lasso)")
plt.xlabel("log10(C parameter)")
plt.ylabel("mean & standard deviation")
plt.show()

""" (ii)(d) repeat for RIDGE 10-fold cross validation to choose C-parameter """
ridge_means = []
ridge_std_devs = []
for ridge_mod in ridge_models:
  results = k_fold_cross_val(10, ridge_mod)
  ridge_means.append(results[0])
  ridge_std_devs.append(results[1])

c_logs = np.log10(ridge_c_params)

plt.errorbar(c_logs, ridge_means, yerr=ridge_std_devs, fmt='.', capsize=5)
plt.plot(c_logs, ridge_means, linestyle=':', label='mean', linewidth=2,
color='orange')
plt.legend()
plt.title("Mean & Variance vs C (10-fold, Ridge)")
plt.xlabel("log10(C parameter)")
plt.ylabel("mean & standard deviation")
plt.show()
```