Data Science 1: Machine Learning Concepts

Assignment by Péter Szilvási

1. Consider the simplest possible predicitive model

$$Y_i = \beta_0 + \epsilon_i$$

where ϵ_i, $i = 1, \ldots, n$ are independent and identically distributed random variables with $E(\epsilon_i) = 0$ and $Var(\epsilon_i) = \sigma^2$. (Thus, in this model we try to predict Y using only a constant, i.e., $f(x) = \beta_0$ regardless of any X variables we might have at our disposal.) The ridge estimator of β_0 solves

$$\min_b \left[\sum_{i=1}^{n} (Y_i - b)^2 + \lambda b^2 \right]$$

for some $\lambda \geq 0$. In the special case $\lambda = 0$, the solution is of course the OLS estimator. It is easy to show that the general solution to this problem is given by $\hat{\beta}_0^{ridge} = \sum_{i=1}^{n} Y_i/(n + \lambda)$.

a) How does the regularized estimator (predictor) $\hat{\beta}_0^{ridge}$ compare with the OLS estimator?

in this exercise's example, the OLS estimator β0 OLS is obtained when lambda is equal to zero. In this case the regularization term becomes zero, and the ridge regression solution reduces to the OLS estimator.

When lambda is bigger than zero, the regularization term penalizes large values of b, meaning that the ridge estimator will be shrunk towards zero compared to the OLS estimator.

b) Suppose that $\beta_0 = 1$ and $\epsilon \sim N(0, \sigma^2)$ with $\sigma^2 = 4$. Generate a sample of size $n = 20$ from the model and compute the predicted value $\hat{Y} = \hat{f}(x) = \hat{\beta}_0^{ridge}$ for a grid of λ values over the interval $[0, 20]$.

In [1]:
```
# libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# b)
beta_0 = 1
sigma_squared = 4
sample_size = 20
lambda_values = np.linspace(0, 20,20)
np.random.seed(20240314)
```

```
# Function to compute ridge estimator for beta_0
def ridge_estimator(Y, lambda_value):
    n = len(Y)
    return np.sum(Y) / (n + lambda_value)

# Generate epsilon from normal distribution
epsilon = np.random.normal(loc=0, scale=np.sqrt(sigma_squared), size=sample_size)

# Compute the response variable Y
Y = beta_0 + epsilon

# Compute ridge estimators for each lambda
ridge_estimates = [ridge_estimator(Y, lambda_val) for lambda_val in lambda_values]
```

In [2]:
```
epsilon.mean() # theoretically, on average should be zero
```

Out[2]:
```
0.3035650679183731
```

In [3]:
```
ridge_over_lambda_grid = pd.DataFrame({
    'Lambda':lambda_values,
    'Ridge Estimate':ridge_estimates
})
ridge_over_lambda_grid
```

Out[3]:

	Lambda	Ridge Estimate
0	0.000000	1.303565
1	1.052632	1.238387
2	2.105263	1.179416
3	3.157895	1.125806
4	4.210526	1.076858
5	5.263158	1.031989
6	6.315789	0.990709
7	7.368421	0.952605
8	8.421053	0.917324
9	9.473684	0.884562
10	10.526316	0.854060
11	11.578947	0.825591
12	12.631579	0.798959
13	13.684211	0.773992
14	14.736842	0.750537
15	15.789474	0.728463
16	16.842105	0.707650
17	17.894737	0.687993
18	18.947368	0.669398
19	20.000000	0.651783

c) Repeat part b), say, 1000 times so that you end up with 1000 estimates of β_0 for all the λ values that you have picked. For each value of λ, compute $bias^2[\hat{\beta}_0^{ridge}]$, $Var[\hat{\beta}_0^{ridge}]$ and $MSE[\hat{\beta}_0^{ridge}] = bias^2[\hat{\beta}_0^{ridge}] + Var[\hat{\beta}_0^{ridge}]$.

In [4]:
```
# Given values
n = 20
sigma_squared = 4
beta_0 = 1
lambda_values = np.linspace(0, 20, 20)
iterations = 1000

# Lists to store the results
bias_squared = []
variance = []
mse = []

# Monte Carlo simulation
for i, lambda_val in enumerate(lambda_values):
    beta_ridge_estimates = []
    for _ in range(iterations):
        # Generate sample
        Y = np.random.normal(beta_0, np.sqrt(sigma_squared), n)

        # Compute the ridge estimator
        beta_ridge = np.sum(Y) / (n + lambda_val)
        beta_ridge_estimates.append(beta_ridge)

    # Calculate bias, variance, and MSE
    bias_squared.append((np.mean(beta_ridge_estimates) - beta_0) ** 2)
    variance.append(np.var(beta_ridge_estimates))
    mse.append(bias_squared[-1] + variance[-1])

# Convert lists to arrays
bias_squared = np.array(bias_squared)
variance = np.array(variance)
mse = np.array(mse)
```

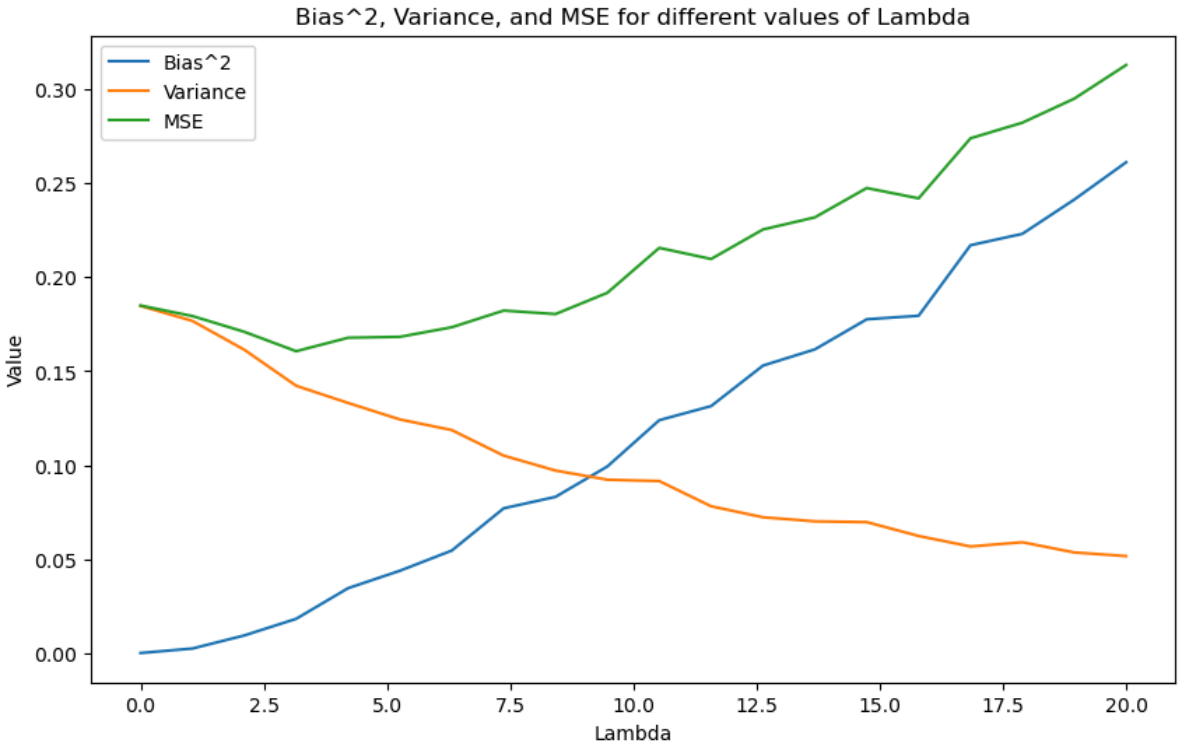
In [5]:
```
pd.DataFrame({
    'lambdas':lambda_values,
    'Bias^2':bias_squared,
    'Variance':variance,
    'MSE':mse
})
```

Out[5]:

	lambdas	Bias^2	Variance	MSE
0	0.000000	0.000196	0.184753	0.184949
1	1.052632	0.002585	0.176776	0.179361
2	2.105263	0.009531	0.161448	0.170979
3	3.157895	0.018377	0.142288	0.160665
4	4.210526	0.034658	0.133166	0.167824
5	5.263158	0.043941	0.124383	0.168324
6	6.315789	0.054672	0.118738	0.173410
7	7.368421	0.077130	0.105152	0.182282
8	8.421053	0.083226	0.097205	0.180432
9	9.473684	0.099407	0.092343	0.191750
10	10.526316	0.123995	0.091623	0.215619
11	11.578947	0.131495	0.078220	0.209715
12	12.631579	0.153041	0.072369	0.225410
13	13.684211	0.161643	0.070201	0.231844
14	14.736842	0.177651	0.069797	0.247448
15	15.789474	0.179525	0.062429	0.241954
16	16.842105	0.217026	0.056845	0.273871
17	17.894737	0.223085	0.059096	0.282181
18	18.947368	0.241291	0.053668	0.294959
19	20.000000	0.261120	0.051732	0.312852

d) Plot $bias^2[\hat{\beta}_0^{ridge}]$, $Var[\hat{\beta}_0^{ridge}]$ and $MSE[\hat{\beta}_0^{ridge}]$ as a function of λ and interpret the results. Can a ridge regression give a better prediction than OLS?

In [6]:
```
# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(lambda_values, bias_squared, label='Bias^2')
plt.plot(lambda_values, variance, label='Variance')
plt.plot(lambda_values, mse, label='MSE')
plt.xlabel('Lambda')
plt.ylabel('Value')
plt.legend()
plt.title('Bias^2, Variance, and MSE for different values of Lambda')
plt.show()
```



Interpretation:

In Ridge regression, if we increase the regularization parameter lambda, we are penalizing the model for having large coefficients.

Bias^2:

As lambda increases, the ridge estimator tends to shrink towards zero, resulting in an increase in bias because the estimator is biased towards zero due to the penalty term added to the loss function.

Variance:

As lambda increases, the variance tends to decrease, as a result of the coefficients shrinking towards zero. Here, our model is getting simpler and simpler as lambda increases and for a simpler model, the variance tends to decrease. Predictions are biased, but with low variance.

MSE:

The MSE is just an addition between the bias^2 and the variance. Ultimately, the MSE is a U-shaped curve, as if the model is too simmple, we are underfitting, while if it's too complex it is overfitting, the U-shape reflects this property of the model.

Can a Ridge Regression give better prediction than OLS?

In [7]:
```
ridge_vs_ols = pd.DataFrame({
    'lambda':lambda_values,
    'MSE':mse
})
ridge_vs_ols
```

Out[7]:

	lambda	MSE
0	0.000000	0.184949
1	1.052632	0.179361
2	2.105263	0.170979
3	3.157895	0.160665
4	4.210526	0.167824
5	5.263158	0.168324
6	6.315789	0.173410
7	7.368421	0.182282
8	8.421053	0.180432
9	9.473684	0.191750
10	10.526316	0.215619
11	11.578947	0.209715
12	12.631579	0.225410
13	13.684211	0.231844
14	14.736842	0.247448
15	15.789474	0.241954
16	16.842105	0.273871
17	17.894737	0.282181
18	18.947368	0.294959
19	20.000000	0.312852

In [8]:
```
# lambda values for ols and ridge
ols_lambda = round(ridge_vs_ols.loc[ridge_vs_ols['MSE'] == ridge_vs_ols['MSE'][0],
ridge_lambda = round(ridge_vs_ols.loc[ridge_vs_ols['MSE'] == ridge_vs_ols['MSE'].mi


# Print the results
print(f"The OLS MSE is {round(ridge_vs_ols['MSE'][0], 3)}. This is where lambda equ
print(f"The Ridge MSE is {round(ridge_vs_ols['MSE'].min(), 3)}. This is where lambd
print('This lambda is chosen by picking the smallest MSE value.')
```

```
The OLS MSE is 0.185. This is where lambda equals to 0 (No penalty is introduced).
The Ridge MSE is 0.161. This is where lambda is 3.158.
This lambda is chosen by picking the smallest MSE value.
```

As seen above, the Ridge gives a lower prediction error, making it a better prediction model. It achieves this by introducing a penalty term, that shrink the coefficients towards zero, which reduces variance compared to the OLS and even though it increses the bias, that is offset by the reduction is variance.

2. ISLR Exercise 3 in Section 6.8 (p. 260). Please use the version of the textbook posted online (7th printing).

3. Suppose we estimate the regression coefficients in a linear regression model by minimizing

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2 \quad \text{subject to} \quad \sum_{j=1}^{p}|\beta_j|\le s$$

for a particular value of s. For parts (a) through (e), indicate which of i. through v. is correct. Justify your answer.

(a) As we increase s from 0, the training RSS will:
   i. Increase initially, and then eventually start decreasing in an inverted U shape.
   ii. Decrease initially, and then eventually start increasing in a U shape.
   iii. Steadily increase.
   iv. Steadily decrease.
   v. Remain constant.

(b) Repeat (a) for test RSS.

(c) Repeat (a) for variance.

(d) Repeat (a) for (squared) bias.

(e) Repeat (a) for the irreducible error.

a) (iv) Steadily decrease.

As we increase s, the coefficients also increase, eventually reaching their values from the OLS model. This leads to a steadily decrease in the training error, as the model becomes more complex it fits the training data better.

b) (ii) Decrease initially, and then eventually start increasing in a U shape.

At s = 0, our model is very simple, with all coefficients set to zero. This simplicity leads to high test error. As we increase s, the model starts to fit the test better and better. Eventually it starts to increase again as we overfit the model on the training data and it is not able to capture the patterns on the test sample.

c) (iii) Steadily increase.

When there's no penalty, our model predicts a constant value and has very little variance. As we introduce more penalty, the model has higher coefficients, leading to higher variability in the predictions.

d) (iv) Steadily decrease.

With no penalty, the model predicts a constant which is far form the actual value in the data. This is because the model is too

simple. However, as we increase s, coefficients become non-zero and the model can fit the complexity of our data better and better, thus leading to a decrease in bias.

e) (v) Remain constant

The irreducible error represent the randomness or noise in the data that cannot be captured by any model, regardless of how complex or simple it is. It is not going to be dependant on the s, so it is constant.

3. Consider the 'dense' regression model discussed in the last lecture:

$$Y_i = \beta_0 + \beta_1 X_{1i} + \beta_2 X_{2i} + \ldots + \beta_{50} X_{50,i} + \epsilon_i,$$

where $X_1, \ldots, X_{50}$ are correlated jointly normal random variables, $\epsilon \sim N(0, 2^2)$, and the regression coefficients are arbitrarily chosen numbers between 0 and 1 (not shown).

The file PCA_data.csv contains a training sample of size $n = 500$ and a test sample of size $m = 500$ generated from this model. The exercise below asks you to do the exercise that produces the last column of the 'Dense DGP' table on slide 22, Lecture 3. (I posted it on Moodle.)

a) Load the data set called `PCA_data.csv` (posted). Designate the first $N_{tr} = 500$ observations as the training sample and the last $N_{te} = 500$ as the test sample.

In [9]:
```
# import everything necessary for this exercise
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

data = pd.read_csv('data/PCA_data.csv')
data.shape
```

Out[9]:
```
(1000, 51)
```

In [10]:
```
data
```

Out[10]:

	Y	X1	X2	X3	X4	X5	X6	X7	X
0	-6.539179	0.948014	1.259177	0.763472	0.128735	0.410222	0.420989	-0.101123	-1.2425
1	2.036508	0.019661	-1.951131	-1.097787	0.919061	-0.069719	0.405042	1.808955	-0.3430
2	7.295433	0.283019	1.127620	1.458154	0.227844	-2.003859	-1.775359	-2.480777	0.5066
3	6.825405	0.845641	-0.921489	-1.368601	2.085195	-0.294420	1.029018	0.098418	-0.6775
4	-0.176432	0.566670	0.667277	-0.248455	1.361258	0.041326	0.149155	-0.271543	0.2291
...	...	...	...	...	...	...	...	...	
995	-3.298207	1.152532	0.610173	2.956214	-1.432405	-2.397398	-0.945650	-0.952754	-0.8250
996	2.334896	-0.434171	-0.293504	-1.437907	-0.106369	1.617408	1.136066	0.214231	0.0321
997	-1.049071	-1.474982	-0.437761	-0.264495	-1.270952	0.129644	-0.668138	-0.298138	1.4660
998	-6.145876	0.349684	-0.670108	0.161680	-0.250096	-0.658828	-0.076775	-0.338317	-1.3905
999	-1.320450	1.429819	1.601074	0.964628	0.272910	0.848267	1.272262	1.013718	-1.2562

1000 rows × 51 columns

In [11]:
```
train = data.iloc[:500]
test = data.iloc[500:]

X_train = train.drop(columns=['Y'])
y_train = train['Y']
X_test = test.drop(columns=['Y'])
y_test = test['Y']
```

b) Compute the first 10 principal component vectors and the corresponding scores $Z_1^*, \ldots, Z_{10}^*$ for $(X_1, X_2, \ldots, X_{50})$. For simplicity, you can use the whole data set for this (both the training sample as well as the test sample).

In [12]:
```
pca = PCA(n_components=10)
X_pca_train = pca.fit_transform(X_train)
X_pca_test = pca.transform(X_test)
```

In [13]:
```
X_pca_train.shape[1] # number of Z scores
```

Out[13]:
```
10
```

In [14]:
```
pca.explained_variance_ratio_ # show Z scores
```

Out[14]:
```
array([0.11289815, 0.09771981, 0.0887854 , 0.07683269, 0.06390704,
       0.0571617 , 0.05223682, 0.0470202 , 0.04548858, 0.04133884])
```

c) Estimate an OLS regression of Y on a constant and $X_1, \ldots, X_{50}$ over the training sample. Estimate OLS regressions of Y on a constant and $Z_1^*, \ldots, Z_k^*$ over the training sample for $k = 1, 5, 10$.

d) Use the four models estimated under part c) the obtain predictions for the outcomes Y_i in the test sample. Compute the mean squared prediction error for the four different predictions and report these numbers. You should get results similar to those on slide 22, but there will be some differences because the whole experiment is performed only once. (The slide averages over many experiments.)

In [15]:
```
# c)
# Estimate OLS regression of Y on X1, ..., X50 over the training sample
ols_X_model = LinearRegression().fit(X_train,y_train)

# Estimate OLS regression Y on Z1, ..... Z10 over the training sample
ols_X_pca_model = LinearRegression().fit(X_pca_train,y_train)

# Create Linear Regression models for with Principle Components
k_values = [1,5,10]

ols_X_pca_models = {} # to store models

for k in k_values:
    ols_X_pca_models[k] = LinearRegression().fit(X_pca_train[:,:k],y_train)

# d)
# Initiate lists to store data for summary table
models = []
mspes = []
models.append("OLS All Features")

# Predict on OLS with all features
ols_preds = ols_X_model.predict(X_test)
mspes.append(mean_squared_error(ols_preds,y_test))

# Predict on PCA linear regressions
for k in k_values:
    model = f'PCA (k={k})'
    pred = ols_X_pca_models[k].predict(X_pca_test[:,:k])
    mspe = mean_squared_error(pred,y_test)

    models.append(model)
    mspes.append(mspe)

# Create summary Table
summary_table = pd.DataFrame({
    "Model":models,
    "MSPE":mspes
    })
summary_table
```

Out[15]:

	Model	MSPE
0	OLS All Features	4.334077
1	PCA (k=1)	15.017898
2	PCA (k=5)	12.470383
3	PCA (k=10)	9.035481

e) Consider again the original 'Dense DGP' table on slide 22, Lecture 3. Discuss and explain the MSPE patterns you see in the first column ($N_{tr} = 75$) and the last column ($N_{tr} = 500$).

DENSE DGP	$N_{tr} = 75$ MSPE	$N_{tr} = 150$ MSPE	$N_{tr} = 500$ MSPE
OLS	12.9	6.0	4.5
PCA (k=1)	14.9	14.7	14.6
PCA (k=5)	13.6	13.0	12.7
PCA (k=10)	9.3	8.5	8.0

First of all, the MSPE of OLS improves a lot as we increase the training sample, because a larger sample provides more information about the underlying relationships in the data, allowing the OLS to better capture the true patterns and thus reduce prediction error. However, this model has a very high number of coefficients thus, it is prone to overfitting. This is where PCA is great, as they can simplify our predictors by reducing dimensionality and still capturing most of the variance in our data.

As for our PCA Models, the one with 10 principle components even outperforms the OLS in the small training sample. This suggests that in a smaller training sample, capturing more variance is beneficial for reducing prediction errors. However, in the larger dataset even with the highest number of principle components the PCA models can't outperform the OLS model. Still, increasing k leads to a lower MSPE, showing that additional complexity introduced by higher values for k allows the model to capture more patterns in the data

In conclusion, we can see that if we wish to get the lowest prediction errors we have to be careful with overfitting and we might have to trade some prediction accuracy to get a simpler model.