Mathematical Foundations of Machine Learning. Homework 3

Teresa Morales

October, 28th 2019

Homework 3 Teresa Morales

1\. a) $\hat{r} = y - X\hat{\omega}$, $r = y - X\omega$ } $y = \hat{r} + X\hat{\omega}$, $r = (\hat{r} + X\hat{\omega}) - X\omega = \boxed{\hat{r} + X(\hat{\omega} - \omega)}$

b) $||r||^2 = (\hat{r} + X(\hat{\omega}-\omega))^T(\hat{r} + X(\hat{\omega}-\omega))$

$= \hat{r}^T\hat{r} + \hat{r}^T X(\hat{\omega}-\omega) + (\hat{\omega}-\omega)^T X^T \hat{r} + (\hat{\omega}-\omega)^T X^T X(\hat{\omega}-\omega)$

c) $\hat{r}$ is orthogonal to the columns of X which implies that

$X^T\hat{r} = \hat{r}^T X = \underline{0} \rightarrow ||r||^2 = \hat{r}^T\hat{r} + (\hat{\omega}-\omega)X^T X(\hat{\omega}-\omega)$

If X^TX is positive definite, then $(\hat{\omega}-\omega)X^TX(\hat{\omega}-\omega) > 0 \ \forall (\hat{\omega}-\omega)$

This would therefore imply that $||r||^2 > ||\hat{r}||^2 \Leftrightarrow ||r|| > ||\hat{r}||$

and $\hat{\omega}$ is the least squares solution.

2\. a) We can use Gram-Schmidt orthogonalization:

$X_1 = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix} \quad X_2 = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$

(1º) We initialize: $u_1 = X_1 / ||X_1||_2 = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix} / 3 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

(2º) We write X_2 as a weighted sum of u_1 plus a residual

$X_2 = w \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \text{resid}$

$w = \underset{w}{\text{argmin}} \, ||X_2 - u_1 w||_2^2$

$w = \underbrace{(u_1^T u_1)}_{1}^{-1} u_1^T X_2 = u_1^T X_2$

$\text{resid} = \underline{X_2} - \underline{u_1}(u_1^T X_2)$

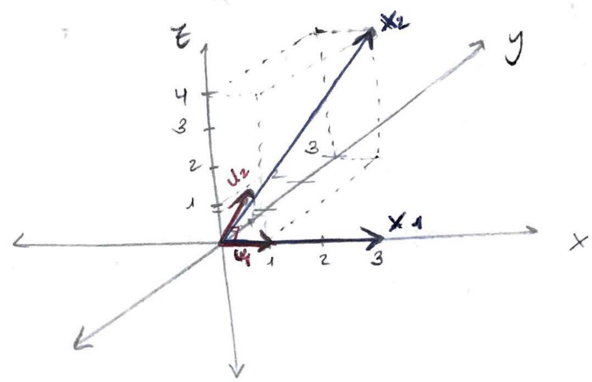
$= \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} [1 \ 0 \ 0] \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$

$= \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 3 \\ 4 \end{bmatrix}$

(8°) $u_2 = \frac{\text{resid}}{\|\text{resid}\|_2} = \begin{bmatrix}0\\3\\4\end{bmatrix} / \sqrt{25} = \begin{bmatrix}0\\3/5\\4/5\end{bmatrix}$

$\Rightarrow$ the orthonormal vectors are $\begin{bmatrix}1\\0\\0\end{bmatrix}$ and $\begin{bmatrix}0\\3/5\\4/5\end{bmatrix}$

b)



c) Compute $\hat{y} = X(X^T X)^{-1} X^T y$

I can express X as a weighted sum of columns of U, which means that $\hat{y} = Xw = U\tilde{w}$, where

$$\tilde{w} = \underset{w}{\operatorname{argmin}} \|y - U\tilde{w}\|_2^2 = (\underbrace{U^T U}_{I})^{-1} U^T y = U^T y$$

$$\leftrightarrow \hat{y} = UU^T y = \begin{bmatrix}1 & 0\\0 & 3/5\\0 & 4/5\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\0 & 3/5 & 4/5\end{bmatrix}\begin{bmatrix}1\\3\\1\end{bmatrix}$$

$$= \begin{bmatrix}1 & 0 & 0\\0 & 9/25 & 12/25\\0 & 12/25 & 16/25\end{bmatrix}\begin{bmatrix}1\\3\\1\end{bmatrix} = \begin{bmatrix}1\\1.56\\2.08\end{bmatrix}$$

Problem 3

The code to perform Gram-Schmidt orthogonalization is the following:

```python
import numpy as np
import math

def eliminate_zeros(X):
    X_nz = X[:, ~np.all(X == 0, axis=0)]
    return X_nz


def initialize(X, rows):  # Calculate Uj for j = 1.
    U_1 = X[:, 0]/math.sqrt(sum([X[i, 0]**2 for i in range(rows)]))
    return U_1


def calc_Uj(X, U, rows, j):  # Calculate Uj for j = 2, 3...p.
    X_j = X[:, j-1] - \
        sum([np.dot(U[:, i], np.dot(np.transpose(U[:, i]), X[:, j-1])) for i in range(j-1)])
    X_jr = np.array([round(X_j[i], 10) for i in range(rows)])
    if np.all(X_jr == 0):
        U_j = X_jr
    else:
        U_j = X_j/math.sqrt(sum([X_j[i]**2 for i in range(rows)]))
    return U_j


def run_GS(X):
    X_np = eliminate_zeros(X)
    n_rows = len(X_np)
    U_1 = initialize(X=X_np, rows=n_rows)
    U = np.transpose(np.array([U_1]))  # creates structure to append Ujs.
    n_cols = len(X_np[0])
    for j in range(2, n_cols+1):  # loop to append Ujs for j = 2...p.
        U_j = calc_Uj(X_np, U, n_rows, j)
        U_c = np.transpose(U)
        U = np.transpose(np.vstack([U_c, U_j]))
    U_final = eliminate_zeros(U)
    print('U is', U_final)
    print('The rank of X is', len(U_final[0]))
```

Here are some examples (X1 and X2 correspond to examples given in class):

```python
X_1 = np.array([[1, 1],
                [1, 3],
                [0, 0]])

X_2 = np.array([[1, -1, 2],
                [-1, 1, 2]])

X_3 = np.array([[0, 1, -1, 2, 0, 8],
                [1, 3, -1, 1.8, 0, 2],
                [4, 6, -8, 0, 0, 2],
```

```
                 [2, 1, 0, 3.5, 0, 2],
                 [1, 3, -2.3, 1, 0, 7]])

run_GS(X_1)
U is [[ 0.70710678 -0.70710678]
 [ 0.70710678  0.70710678]
 [ 0.          0.        ]]
The rank of X is 2

run_GS(X_2)
U is [[ 0.70710678  0.70710678]
 [-0.70710678  0.70710678]]
The rank of X is 2

run_GS(X_3)
U is [[ 0.          0.32522182 -0.06847489  0.92229249 -0.19727777]
 [ 0.21320072  0.50261554  0.62675823 -0.2381579  -0.50237239]
 [ 0.85280287  0.05913124 -0.48948081 -0.08874738 -0.14752358]
 [ 0.42640143 -0.62087802  0.57368199  0.29095535  0.13757275]
 [ 0.21320072  0.50261554  0.18380103  0.01123671  0.81732123]]
The rank of X is 5
```

4. Suppose $A = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ what is its SVD?

By inspection, we can see that A is a diagonal matrix with positive entries and with $a_{11} \geq a_{22} \geq 0$

Then, the simplest SVD would be the following

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$U = I$

$V = I$

$\Sigma = A$

5. $A = \begin{bmatrix} -2 & 0 \\ 0 & -1 \end{bmatrix}$

In this case, A is also a diagonal matrix, but its diagonal entries are ~~the~~ negative values. $a_{ii} \leq 0$. However, absolute value of those entries does satisfy ~~#~~ $|a_{11}| \geq |a_{22}|$

We could therefore represent A as

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

So the SVD would be $U = I$, $V = -I$ and

$\Sigma = -A$

6. $A \in \mathbb{R}^{n \times d}$

a) Let $r = \text{rank}(A)$, we can represent A as

$$A = U \Sigma V^T = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_r \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r \end{bmatrix} \begin{bmatrix} — v_1 — \\ — v_2 — \\ \vdots \\ — v_r — \end{bmatrix}$$

$(n \times r) \quad (r \times r) \quad (r \times d)$

This would represent the economy singular value decomposition. We can now use the outer-product representation and express this as a sum of rank-1 matrices:

$$A = \begin{bmatrix} | \\ u_1 \\ | \end{bmatrix} [\sigma_1] \boxed{[— v_1 —]} + \begin{bmatrix} | \\ u_2 \\ | \end{bmatrix} [\sigma_2] [— v_2 —] + \cdots + \begin{bmatrix} | \\ u_r \\ | \end{bmatrix} [\sigma_r] [— v_r —]$$

$(n \times 1) \quad (1 \times 1) \quad (1 \times d)$

$$A = \begin{bmatrix} u_{11}\sigma_1 v_{11} & u_{11}\sigma_1 v_{12} & \cdots & u_{11}\sigma_1 v_{1d} \\ u_{21}\sigma_1 v_{11} & u_{21}\sigma_1 v_{12} & & u_{21}\sigma_1 v_{1d} \\ \vdots & \vdots & & \vdots \\ u_{n1}\sigma_1 v_{11} & u_{n1}\sigma_1 v_{12} & & u_{n1}\sigma_1 v_{1d} \end{bmatrix} + \begin{bmatrix} u_{12}\sigma_2 v_{21} & u_{12}\sigma_2 v_{22} & \cdots & u_{12}\sigma_2 v_{2d} \\ u_{22}\sigma_2 v_{21} & u_{22}\sigma_2 v_{22} & & u_{22}\sigma_2 v_{2d} \\ \vdots & \vdots & & \vdots \\ u_{n2}\sigma_2 v_{21} & u_{n2}\sigma_2 v_{22} & & u_{n2}\sigma_2 v_{2d} \end{bmatrix}$$

$(n \times d) \quad (n \times d)$

$$+ \cdots + \begin{bmatrix} u_{1r}\sigma_r v_{r1} & u_{1r}\sigma_r v_{r2} & \cdots & u_{1r}\sigma_r v_{rd} \\ u_{2r}\sigma_r v_{r1} & u_{2r}\sigma_r v_{r2} & & u_{2r}\sigma_r v_{rd} \\ \vdots & \vdots & & \vdots \\ u_{nr}\sigma_r v_{r1} & u_{nr}\sigma_r v_{r2} & & u_{nr}\sigma_r v_{rd} \end{bmatrix}$$

$(n \times d)$

b) Recall rank-k approximations A_k, where $k < r$. Express A_k with the sum of rank one matrices

The subspace approximation theorem tells us that if $A \in \mathbb{R}^{n \times d}$ with rank r, and $k < r$, the best rank k approximation to A is:

$$\min_{Z \in \mathbb{R}^{n \times d}} \| A - Z \|_F^2 = U_k \Sigma_k V_k^T$$

Where U_k is formed by the first columns of U, Σ_k is a k by k diagonal matrix formed by the first diagonal elements of Σ and V_k^T is formed by the first k rows of V^T

$$A_k = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_k \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_k \end{bmatrix} \begin{bmatrix} — & v_1 & — \\ — & v_2 & — \\ & \vdots & \\ — & v_k & — \end{bmatrix}$$

$(n \times k) \quad (k \times k) \quad (k \times d)$

We can use the same type of representation and express A as a sum of rank one matrices.

$$A_k = \begin{bmatrix} | \\ u_1 \\ | \end{bmatrix} \sigma_1 [— v_1 —] + \begin{bmatrix} u_2 \end{bmatrix} (\sigma_2) [— v_2 —] + \dots + \begin{bmatrix} u_k \end{bmatrix} [\sigma_k] [— v_k —]$$

$$A_k = \begin{bmatrix} u_{11}\sigma_1 v_{11} & u_{11}\sigma_1 v_{12} & \dots & u_{11}\sigma_1 v_{1d} \\ u_{21}\sigma_1 v_{11} & u_{21}\sigma_1 v_{12} & & u_{21}\sigma_1 v_{1d} \\ \vdots & \vdots & & \vdots \\ u_{n1}\sigma_1 v_{11} & u_{n1}\sigma_1 v_{12} & & u_{n1}\sigma_1 v_{1d} \end{bmatrix} + \dots + \begin{bmatrix} u_{1k}\sigma_k v_{k1} & u_{1k}\sigma_k v_{k2} & \dots & u_{1k}\sigma_k v_{kd} \\ u_{2k}\sigma_k v_{k1} & u_{2k}\sigma_k v_{k2} & & \vdots \\ \vdots & & & \vdots \\ u_{nk}\sigma_k v_{k1} & \dots & \dots & u_{nk}\sigma_k v_{kd} \end{bmatrix}$$

7- a) What we would do is estimate three different models of least squares for each flour species.

For setosa we would have

$y_s \in \{-1, 1\}$ and $\hat{y}_s = X w_s \simeq y_s$

$$\hat{w}_s = \underset{w_s}{\arg\min} \| y_s - X w_s \|_2^2$$

$$\hat{w}_s = (X^T X)^{-1} X^T y_s$$

For versicolor we would have

$y_{vc} \in \{-1, 1\}$ and $\hat{y}_{vc} = X w_{vc} \simeq y_{vc}$

$$\hat{w}_{vc} = \underset{w_{vc}}{\arg\min} \| y_{vc} - X w_{vc} \|_2^2$$

$$\hat{w}_{vc} = (X^T X)^{-1} X^T y_{vc}$$

For virginica we would have

$y_{vir} \in \{-1, 1\}$ and $\hat{y}_{vir} = X w_{vir} \simeq y_{vir}$

$$\hat{w}_{vir} = \underset{w_{vir}}{\arg\min} \| y_{vir} - X w_{vir} \|_2^2$$

Problem 7

The code for sections b and c is the following:

```python
import numpy as np
import math
import random
import pandas as pd
import scipy.io as sio
import matplotlib.pyplot as plt

file_path = '/Users/teresamorales/Documents/Harris/MFML/Homework 3/
    fisheriris.mat'
fisher_data = sio.loadmat(file_path)
Species = fisher_data['species']
X = fisher_data['meas']
len(X[0])  # 4 columns
len(X)  # 150 rows


def create_dummies(name_species, n):
    y = np.array([1 if Species[i, 0] == name_species else -1 for i
    in range(n)])
    return y

def calc_weights(X, y):
    Xt = np.transpose(X)
    XtX = np.matmul(Xt, X)
    Inverse_XtX = np.linalg.inv(XtX)
    w = np.matmul(np.matmul(Inverse_XtX, Xt), y)
    return w

def divide_randomly(sample_size, training_size, seed):
    sample_size_index = list(range(sample_size))
    random.Random(seed).shuffle(sample_size_index)
    return [sample_size_index[0:training_size], sample_size_index[
    training_size:]]


def sign(num):
    return -1 if num < 0 else 1


def calc_error_rate(splited_index, features_matrix, labels_vector):
    X_reserved = features_matrix[splited_index[1]]
    y_reserved = labels_vector[splited_index[1]]
    X_training = features_matrix[splited_index[0]]
    y_training = labels_vector[splited_index[0]]
    w_training = calc_weights(X_training, y_training)
    y_hat = np.matmul(X_reserved, w_training)
    y_tilda = [sign(i) for i in y_hat]
    return np.sum([y_tilda[i] != y_reserved[i] for i in range(len(
    splited_index[1]))])/len(splited_index[1])


#To run the program, experimenting with different training sample
    sizes, we do the following:
```

```

n_repetitions = 50
training_size = [120, 115, 110, 105, 100, 95, 90, 85, 80, 75,
                 70, 65, 60, 55, 50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 3]

# (1  ) Setosa:
y_setosa = create_dummies('setosa', 150)

error_rates_setosa = {i: [] for i in training_size}

for size in training_size:
    for i in range(n_repetitions):
        index = divide_randomly(150, size, seed=i)
        error = calc_error_rate(index, X, y_setosa)
        error_rates_setosa[size].append(error)

mean_error_setosa = {i: np.mean(error_rates_setosa[i]) for i in training_size}

# (2  ) Versicolor:

y_versicolor = create_dummies('versicolor', 150)

error_rates_versicolor = {i: [] for i in training_size}

for size in training_size:
    for i in range(n_repetitions):
        index = divide_randomly(150, size, seed=i)
        error = calc_error_rate(index, X, y_versicolor)
        error_rates_versicolor[size].append(error)

mean_error_versicolor = {i: np.mean(error_rates_versicolor[i]) for i in training_size}

# (3  ) Virginica:

y_virginica = create_dummies('virginica', 150)

error_rates_virginica = {i: [] for i in training_size}

for size in training_size:
    for i in range(n_repetitions):
        index = divide_randomly(150, size, seed=i)
        error = calc_error_rate(index, X, y_virginica)
        error_rates_virginica[size].append(error)

mean_error_virginica = {i: np.mean(error_rates_virginica[i]) for i in training_size}

# Making a graph

setosa = {'training_size': list(mean_error_setosa.keys()),
          'mean_error_setosa': list(mean_error_setosa.values())}
flours_df = pd.DataFrame(data=setosa)

flours_df['mean_error_versicolor'] = list(mean_error_versicolor.
```

```
    values())

flours_df['mean_error_virginica'] = list(mean_error_virginica.
    values())

fig, ax = plt.subplots(figsize=(12, 6))
plt.plot('training_size', 'mean_error_setosa', data=flours_df,
    color='purple')
plt.plot('training_size', 'mean_error_versicolor', data=flours_df,
    color='red')
plt.plot('training_size', 'mean_error_virginica', data=flours_df,
    color='green')
plt.show()
```

b) What is the average test error (number of mistakes divided by 30)? We can print the average test error for our three models, with sample sizes of 120, for our 50 repetitions as follows:

```
>>> flours_df.loc[0,:]

training_size            120.000000
mean_error_setosa          0.000000
mean_error_versicolor      0.288667
mean_error_virginica       0.123333
Name: 0, dtype: float64
```

We observe that the error rate is very different for each type of flour with a much smaller error for setosa, a slightly bigger error for virginica and a bigger error rate for versicolor.

c) Experiment with even smaller sized training sets. Clearly we need at least one training example from each type of flower. Make a plot of average test error as a function of training set size.

We observe that when we use smaller training sets the error tends to get bigger for the three species and the order that we just described is maintained so that the error rate for setosa is always the smallest and for versicolor the largest as we can see in the following graph.

Figure 1:

d) Now design a classifier using only the first three measurements (sepal length, sepal width, and petal length). What is the average test error in this case?

We repeat the process for a different selection of features as follows:

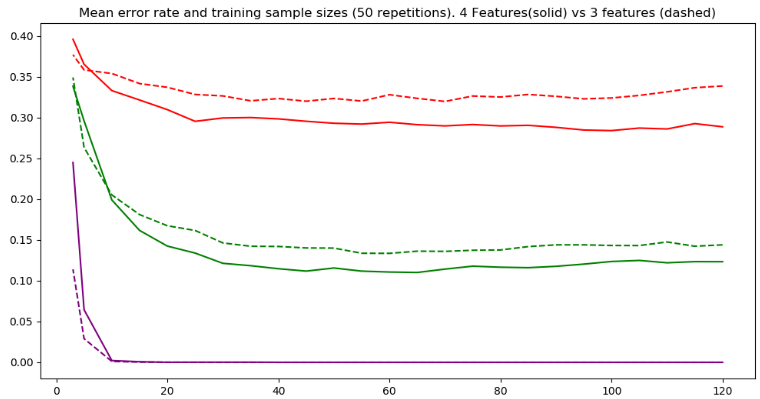
```
X_sel = X[:, 0:3]

# (1  ) Setosa:

error_rates_setosa_sel = {i: [] for i in training_size}

for size in training_size:
    for i in range(n_repetitions):
        index = divide_randomly(150, size, seed=i)
        error = calc_error_rate(index, X_sel, y_setosa)
        error_rates_setosa_sel[size].append(error)

mean_error_setosa_sel = {i: np.mean(error_rates_setosa_sel[i]) for
    i in training_size}

# (2  ) Versicolor:

error_rates_versicolor_sel = {i: [] for i in training_size}

for size in training_size:
    for i in range(n_repetitions):
        index = divide_randomly(150, size, seed=i)
        error = calc_error_rate(index, X_sel, y_versicolor)
        error_rates_versicolor_sel[size].append(error)

mean_error_versicolor_sel = {i: np.mean(error_rates_versicolor_sel[
    i]) for i in training_size}

```

```python
# (3  ) Virginica:

error_rates_virginica_sel = {i: [] for i in training_size}

for size in training_size:
    for i in range(n_repetitions):
        index = divide_randomly(150, size, seed=i)
        error = calc_error_rate(index, X_sel, y_virginica)
        error_rates_virginica_sel[size].append(error)

mean_error_virginica_sel = {i: np.mean(error_rates_virginica_sel[i]) for i in training_size}

# Making a graph

flours_df['mean_error_setosa_sel'] = list(mean_error_setosa_sel.values())
flours_df['mean_error_versicolor_sel'] = list(mean_error_versicolor_sel.values())
flours_df['mean_error_virginica_sel'] = list(mean_error_virginica_sel.values())

fig, ax = plt.subplots(figsize=(12, 6))
plt.plot('training_size', 'mean_error_setosa', data=flours_df, color='purple')
plt.plot('training_size', 'mean_error_versicolor', data=flours_df, color='red')
plt.plot('training_size', 'mean_error_virginica', data=flours_df, color='green')
plt.plot('training_size', 'mean_error_setosa_sel', data=flours_df, color='purple', linestyle='--')
plt.plot('training_size', 'mean_error_versicolor_sel', data=flours_df, color='red', linestyle='--')
plt.plot('training_size', 'mean_error_virginica_sel', data=flours_df, color='green', linestyle='--')
plt.show()
```

We observe that the error rates increase when we use 3 features instead of three. In particular for the flour setosa, with the same training sample size (120) we obtain a mean error rate of 0 again, for virginica we go from 0.12 with 4 features to 0.14 with 3 features and for versicolor we go from 0.29 to 0.34.

As we can see in the graph, this increase happens for all flours except setosa and for most training sample sizes. However, when the training sample sizes get smaller, the error rates tend to get closer and we even observe points where the average error rate with 3 features is smaller than the average error rate with 4 features.

Figure 2:



The following tables show some selected values of this graph. For each species we show the error rates using 4 features (second column) and 3 features (third column):

```
>>> pd.set_option('display.max_columns', None)
>>> flours_df.loc[[0,4,12, 18, 24],['training_size','mean_error_setosa', 'mean_error_setosa_sel']]
    training_size  mean_error_setosa  mean_error_setosa_sel
0             120           0.000000               0.000000
4             100           0.000000               0.000000
12             60           0.000000               0.000000
18             30           0.000167               0.000000
24              3           0.244898               0.114014
>>> flours_df.loc[[0,4,12, 18, 24],['training_size','mean_error_virginica', 'mean_error_virginica_sel']]
    training_size  mean_error_virginica  mean_error_virginica_sel
0             120              0.123333                  0.144000
4             100              0.123600                  0.143200
12             60              0.110667                  0.133556
18             30              0.121333                  0.146333
24              3              0.338367                  0.349116
>>> flours_df.loc[[0,4,12, 18, 24],['training_size','mean_error_versicolor', 'mean_error_versicolor_sel']]
    training_size  mean_error_versicolor  mean_error_versicolor_sel
0             120               0.288667                   0.338667
4             100               0.284000                   0.324000
12             60               0.294222                   0.328000
18             30               0.299500                   0.326500
24              3               0.395918                   0.377007
>>>
```