Midterm-Suphawich-S

October 4, 2021

1 CP - Midterm - 2021

1.1 Instruction

- Modify this file to be Midterm-<Your FirstName-[First Letter of Last Name]>, e.g., Midterm-Chaklam-S.ipynb
- This exam is open-booked; open-internet.
- You ARE NOT allowed to use sklearn or any libraries, unless stated.
- The completed exams in **.ipynb AND .pdf format** shall be submitted at the Google Classroom
- Any instructions not given, feel free to make any assumptions

1.2 Examination Rules:

- Turn on your webcam during the entire period of the exam time
- All work should belong to you. A student should NOT engage in the following activities which proctors reserve the right to interpret any of such act as academic dishonesty without questioning:
- Chatting with any human beings physically or via online methods
- Plagiarism of any sort, i.e., copying from friends. Both copee and copier shall be given a minimum penalty of zero mark for that particular question or the whole exam.
- No make-up exams are allowed. Special considerations may be given upon a valid reason on unpredictable events such as accidents or serious sickness.

1.3 Question 1 (100 points): Basic Python Skills

By now you should be comfortable with creating classes. Please complete the followings:

1) First, please create a Candy() class

- Please create a Candy() class that requires an input of 'color' attribute assigned in its ___ init ___ function when you create an object.(4 point)
- the second attribute in the ___ init ___ function is 'price' which should automatically be assigned according to colors:
  - red costs 2 baht, blue costs 4 baht and others cost 6 baht (12 points)
- Please add a ___ str ___ method that will allow you to print out the color and price of the candy (8 points)

2) We will need another class called CandyBasket() which we will use to keep our candies in. This class should have the following methods (or functions) :

- It should automatically create an empty basket when you create an object from this class. Please create 2 objects called Basket1 and Basket2 from the CandyBasket() class (8 points)
- Create a function for adding candy into the basket. Please add 3 Red candies and 3 Yellow candies into Basket1 and add 3 Blue candies into Basket2. (8 points)
- Create a ___ len ___ function that will allow you to print out the number of candies in the basket. Please print out the number of candies in Basket1 and Basket2 (8 points)
- Create a ___ str ___ function that will allow you to print out the content of the basket. Please print out the content of Basket1 and Basket2 (4 point)
- Also create a function that can plot a bar plot to show the number of candies of each color. Please plot out the content of Basket1 and Basket2 (8 points)
- Create a function for removing a candy from the basket you should be able to choose which candy to remove !) Please remove 2 Yellow candies from Basket1 (12 points)
- Create a function that will allow you to transfer a specific color of candy from one basket to another. (It should also check and **raise and error** if you don't have the specific color of candy that you want to transfer) Please transfer a Purple candy from Basket2 to Basket1 (it should give you an error). and transfer 2 Red candies from Basket1 to Basket2. (20 points)
- Create one last function that will calculate the total price of all candies in your basket. Please calculate and print out the total price of candies in Basket1 and Basket2 (8 points)

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

[2]:
```
class Candy:
    def __init__(self, color):
        self.color = color
        if color == 'red':
            self.price = 2
        elif color == 'blue':
            self.price = 4
        else:
            self.price = 6
    def __str__(self):
        return f"Color: {self.color}, price: {self.price}"

class CandyBasket:
    def __init__(self):
        self.basket = []

    def add(self, candy):
        self.basket.append(candy)

    def remove(self, color, max_color=1):
        new_arr = []
        for b in self.basket:
```

```
            if b.color != color or max_color == 0:
                new_arr.append(b)
            else:
                max_color -= 1
        self.basket = new_arr

    def tranfer(self, color, target, max_color=None):
        objs = []
        if max_color is None:
            max_color = len(self.basket)

        for b in self.basket:
            if len(objs) == max_color:
                break
            if b.color == color:
                objs.append(b)
                self.basket.remove(b)
        if len(objs) == 0:
            raise ValueError(f"{color} does not exist.")
        for b in objs:
            target.add(b)

    def __len__(self):
        return len(self.basket)

    def __str__(self):
        s = ""
        for b in self.basket:
            s += f"{b}\n"
        return s

    def calculate(self):
        total = 0
        for b in self.basket:
            total += b.price
        return total

basket1 = CandyBasket()
basket2 = CandyBasket()

# add
basket1.add(Candy('red'))
basket1.add(Candy('red'))
basket1.add(Candy('red'))
basket1.add(Candy('yellow'))
basket1.add(Candy('yellow'))
basket1.add(Candy('yellow'))
```

```
basket2.add(Candy('blue'))
basket2.add(Candy('blue'))
basket2.add(Candy('blue'))

# print out
print("Basket1 size:", len(basket1))
print("Basket2 size:", len(basket2))


print(f"Basket1 content:")
print(basket1)
print(f"Basket2 content:")
print(basket2)

# remove yellow
basket1.remove('yellow', max_color=2)
print(f"After remove 'yellow'")
print("Basket1 size:", len(basket1))
print("Basket2 size:", len(basket2))
print(f"Basket1 content:")
print(basket1)
print(f"Basket2 content:")
print(basket2)

# test transfer
basket1.tranfer('red', basket2, max_color=2)
print(f"After transfer 'red' from basket2 to basket1.")
print("Basket1 size:", len(basket1))
print("Basket2 size:", len(basket2))
print(f"Basket1 content:")
print(basket1)
print(f"Basket2 content:")
print(basket2)

# price
print(f"Total price of basket1: ", basket1.calculate())
print(f"Total price of basket2: ", basket2.calculate())
```

```
Basket1 size: 6
Basket2 size: 3
Basket1 content:
Color: red, price: 2
Color: red, price: 2
Color: red, price: 2
Color: yellow, price: 6
Color: yellow, price: 6
Color: yellow, price: 6
```

```
Basket2 content:
Color: blue, price: 4
Color: blue, price: 4
Color: blue, price: 4

After remove 'yellow'
Basket1 size: 4
Basket2 size: 3
Basket1 content:
Color: red, price: 2
Color: red, price: 2
Color: red, price: 2
Color: yellow, price: 6

Basket2 content:
Color: blue, price: 4
Color: blue, price: 4
Color: blue, price: 4

After transfer 'red' from basket2 to basket1.
Basket1 size: 2
Basket2 size: 5
Basket1 content:
Color: red, price: 2
Color: yellow, price: 6

Basket2 content:
Color: blue, price: 4
Color: blue, price: 4
Color: blue, price: 4
Color: red, price: 2
Color: red, price: 2

Total price of basket1:  8
Total price of basket2:  16
```

1.4 Question 2 (195 points): Data Science

Complete the following data science pipeline:

1) **Loading**: Load boston data from sklearn.datasets (5 points)

2) **Feature Selection/Engineering**

- Perform some basic exploratory data analysis as appropriate to determine what are the top five features potentially useful for the prediction (15 points)
- Create X using the top five features, and the corresponding y (10 points)

- Do train test split the data **FROM SCRATCH** (5 points)
- Scale your data **FROM SCRATCH** using **Box-Cox transform** with λ = -1 (15 points) : (6.3.2.2. Mapping to a Gaussian distribution **Box-Cox transform**) https://scikit-learn.org/stable/modules/preprocessing.html#mapping-to-a-gaussian-distribution

$$x_i^{(\lambda)} = \begin{cases} \dfrac{x_i^{(\lambda)} - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \ln(x_i) & \text{if } \lambda = 0 \end{cases}$$

3) **Model Implementation**: Implement Linear Regression class **FROM SCRATCH** with the following features

- choice of **weight initialization** (normal distribution with mean=0 and std=1, random between 0-1, all zero) (15 points)
- choice of **sampling method** (batch, mini batch, stochastic) (15 points)
- **raise ValueError** where appropriate (5 points)
- use **without replacement** for stochastic (5 points)
- choice of **loss function** (normal, ridge) (15 points)
  - normal loss function

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$\frac{\partial J}{\partial \theta_j} = \sum_{i=1}^{m}(h^{(i)} - y^{(i)})x_j$$

  - ridge loss function

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{i=1}^{n}\theta_i^2$$

$$\frac{\partial J}{\partial \theta_j} = \sum_{i=1}^{m}(h^{(i)} - y^{(i)})x_j + \lambda\theta_j$$

- Implement a feature importance function **FROM SCRATCH** which will output a bar plot, plotting the importance scores of each features in sorted order (15 points).
  - If we have scaled our data in our beginning (e.g., standard scaler), feature importance scores are simply the weights. Larger weights imply stronger importance. Anyhow, plotting them can be nasty since some weights can be very big, thus you may want to log-transform them.

4) **Training/Experiment**: Implement a cross validation experiment **FROM SCRATCH** comparing their cross-validation accuracy, and output their feature importance scores (15 points)

- mini-batch sampling, normal loss
- mini-batch sampling, ridge loss with lambda of 0.05
- stochastic sampling, normal loss
- stochastic sampling, ridge loss with lambda of 0.05

5) **Training/Testing**: Select the best parameters you found in 4), train and verify the best model with the testing set. Plot training and testing losses and accuracies as number of iters increases (10 points)

6) **Communication**: Interpret your results (e.g., overfitting, weights, accuracy, etc.). Critical analyses will be used as main criteria for scoring (50 points)

1.4.1 Loading

[3]:
```
from sklearn.datasets import load_boston
boston = load_boston()
```

1.4.2 Feature Selection/Engineering

[4]:
```
feature_names = boston.feature_names
print(boston.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value
(attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000
sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0
otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by
town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's
```

```
    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie
Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
…', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that
address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential
Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In
Proceedings on the Tenth International Conference of Machine Learning, 236-243,
University of Massachusetts, Amherst. Morgan Kaufmann.
```

[5]:
```python
def train_test_split(X, y, percent_train=.7):
    idx = np.arange(0,len(X),1)
    np.random.shuffle(idx)
    idx_train = idx[0:int(percent_train*len(X))]
    idx_test = idx[len(idx_train):len(idx)]
    X_train = X[idx_train]
    X_test = X[idx_test]
    y_train = y[idx_train]
    y_test = y[idx_test]
    return X_train, X_test, y_train, y_test

def boxcox(X, lam=0):
    if (lam==0):
        return np.log(X)
    top = (X ** lam) - 1
    return top / lam
```

I decided to choose last 5 features of the dataset to be top feature for training this model because

it can be impact directly to the house prices.

- DIS weighted distances to five Boston employment centres
- RAD index of accessibility to radial highways
- TAX full-value property-tax rate per $10,000
- B 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT % lower status of the population

[6]:
```
X = boston.data[:, [7, 8, 9, 11, 12]]
y = boston.target
X = boxcox(X, -1)
```

[7]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, percent_train = 0.7)

assert X_train.shape[0] == y_train.shape[0]
assert X_test.shape[0] == y_test.shape[0]

# add intercept to our X
intercept = np.ones((X_train.shape[0], 1))
X_train   = np.concatenate((intercept, X_train), axis=1)  #add intercept
intercept = np.ones((X_test.shape[0], 1))
X_test    = np.concatenate((intercept, X_test), axis=1)  #add intercept

print("X_train:", X_train.shape)
print("X_test:", X_test.shape)
print("y_train:", y_train.shape)
print("y_test:", y_test.shape)
```

```
X_train: (354, 6)
X_test: (152, 6)
y_train: (354,)
y_test: (152,)
```

1.4.3 Model Implementation

[8]:
```
class LinearRegression:
    def __init__(self, alpha=0.001, max_iter=10000,
                 weight_method="zero", fit_method="batch", loss_method="normal",
                 without_replacement=True, ridge_lamda=0.05):
        self.alpha = alpha
        self.max_iter = max_iter
        self.losses = []
        self.weight_method = weight_method # normal, random, zero
        self.fit_method = fit_method # batch, sto, minibatch
        self.loss_method = loss_method # normal, ridge
        self.ridge_lamda = ridge_lamda
        self.without_replacement = without_replacement
```

```
    def init_weight(self, n):
        method = self.weight_method
        if method == "normal":
            self.theta = np.zeros(n)
        elif method == "random":
            self.theta = np.random.rand(n,)
        elif method == 'zero':
            self.theta = np.zeros(n)
        else:
            raise ValueError(f"Weight method is invalid.")

    def fit(self, X, y):
        self.init_weight(X.shape[1])
        iter_stop = 0
        list_of_used_ix = [] #<===without replacement

        for i in range(self.max_iter):
            if self.fit_method == "sto":
                i = np.random.randint(X.shape[0])
                if self.without_replacement:
                    while i in list_of_used_ix:
                        i = np.random.randint(X.shape[0])
                X_train = X[i, :].reshape(1, -1)
                y_train = y[i]
                list_of_used_ix.append(i)
                if len(list_of_used_ix) == X.shape[0]:
                    list_of_used_ix = []
            elif self.fit_method == 'minibatch':
                batch_size = int(0.3 * X.shape[0])
                ix = np.random.randint(0, X.shape[0]) #<----with replacement
                X_train = X[ix:ix+batch_size]
                y_train = y[ix:ix+batch_size]
            elif self.fit_method == 'batch':
                X_train = X
                y_train = y
            else:
                raise ValueError(f"Fit method is incalid.")

            loss, grad = self.gradient(X_train, y_train)

            # update theta
            self.theta = self.theta - self.alpha * grad
            self.losses.append(loss)

    def predict(self, X): # <--- X_test
        return self.h_theta(X)
```

```
    # can name it predict for easy understanding
    def h_theta(self, X):
        return X @ self.theta

    def mse(self, yhat, y):
        return ((yhat - y)**2 / yhat.shape[0]).sum()

    def normal_loss(self, yhat, y, X):
        return np.dot((yhat - y), X).sum()

    def ridge_loss(self, yhat, y, X):
        return self.normal_loss(yhat, y, X) + np.dot(self.ridge_lamda, self.
↪theta).sum()

    def gradient(self, X, y):
        yhat = self.h_theta(X)
        if self.loss_method == 'normal':
            loss = self.normal_loss(yhat, y, X)
        elif self.loss_method == 'ridge':
            loss = self.ridge_loss(yhat, y, X)
        else:
            raise ValueError(f"Loss method is invalid.")
        grad = X.T @ (yhat - y)
        return loss, grad
```

1.4.4 Training/Experiment

[9]:
```
def cross_validation(X_train, X_test, y_train, y_test, arr_params):
    for i, params in enumerate(arr_params):
        model = LinearRegression(**params)
        model.fit(X_train, y_train)
        yhat = model.predict(X_test)
        mse = model.mse(yhat, y_test)
        print(f"Cross: {i+1}, MSE: {mse}")
```

[10]:
```
arr_params = [
    {'fit_method': 'minibatch', 'loss_method': 'normal'},
    {'fit_method': 'minibatch', 'loss_method': 'ridge', 'ridge_lamda': 0.05},
    {'fit_method': 'sto', 'loss_method': 'normal'},
    {'fit_method': 'sto', 'loss_method': 'ridge', 'ridge_lamda': 0.05},
]

cross_validation(X_train, X_test, y_train, y_test, arr_params)
```

```
Cross: 1, MSE: 21.726659566173538
Cross: 2, MSE: 21.788568599813225
Cross: 3, MSE: 77.60200825282163
```

```
Cross: 4, MSE: 76.74363584346028
```

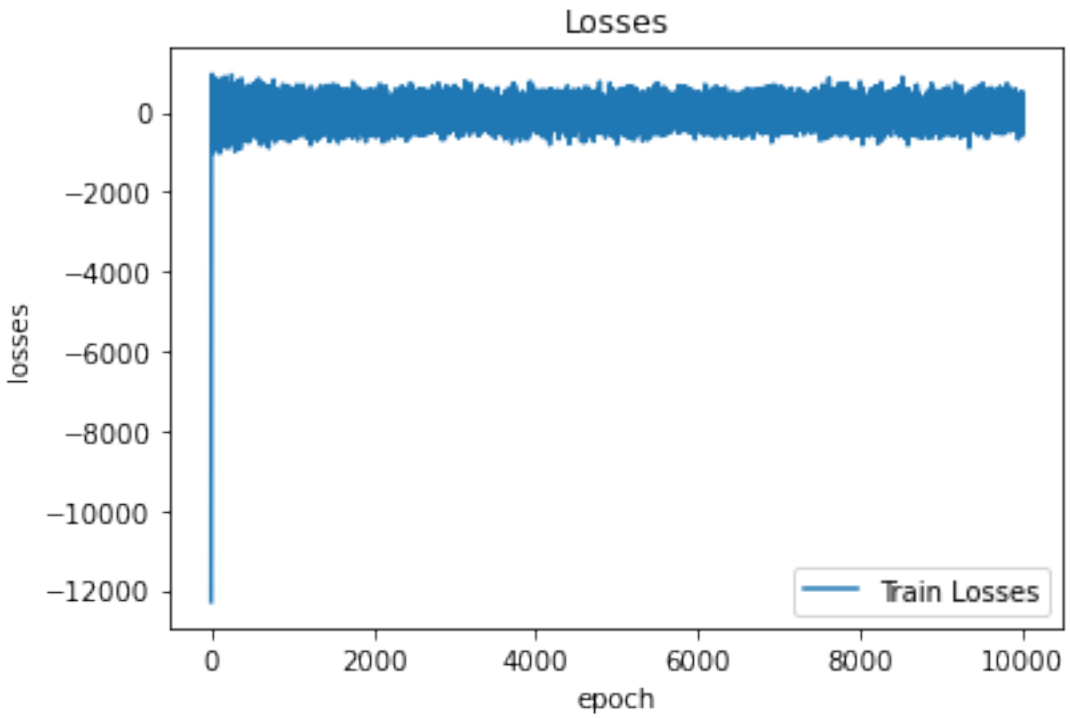
1.4.5 Training/Testing

[11]:
```
model = LinearRegression(**arr_params[0])
model.fit(X_train, y_train)
yhat = model.predict(X_test)
mse = model.mse(yhat, y_test)

print(f"MSE: {mse}")
print(f"Theta: {model.theta}")
```

```
MSE: 21.739822258867786
Theta: [ 49.21443085  -1.55767283  -1.10069228  47.44446912   2.30196113
 -83.8856549 ]
```

[12]:
```
plt.plot(model.losses, label = "Train Losses")
plt.title("Losses")
plt.xlabel("epoch")
plt.ylabel("losses")
plt.legend()
plt.show()
```



1.4.6 Communication

Interpret your result:

1. Picking feature: from cross validation, the MSE is quite more lower than picking other features, also work well in minibatch.
2. Loss (error): from the plot, the loss is quite good because it is decreasing signicifantly and near around zero, although the beginning is negative value.
3. fluctuating: The result is fluctuate a bit, I think because of using minibatch in cross validation.

[ ]: