CSE144 HW1

Jongbin Baek

Summary

In this assignment, the goal is to build a linear regression model to predict the song popularity based on energy, acoustics, instumentalness, liveness, dancibility, etc.

In the first section, you'll perform standard data preprocessing using techniques we covered in class. In the second section, you'll train a simple linear regression model.

```
[66] !python --version

     Python 3.7.13
```

```
[67] #Tuple: you can change
     from typing import List, Tuple
     #marplotlib is the library for plotting
     import matplotlib.pyplot as plt
     #one of the fundamental packages for scientific computing contains the ndarray
     import numpy as np
     #pandas is the python library for data wrangling and analysis DataFrame
     import pandas as pd
     #X is the data that is a two dimensional array(matrix), y is the data that is one dimensional arrat(a vector)
     from sklearn.model_selection import train_test_split
```

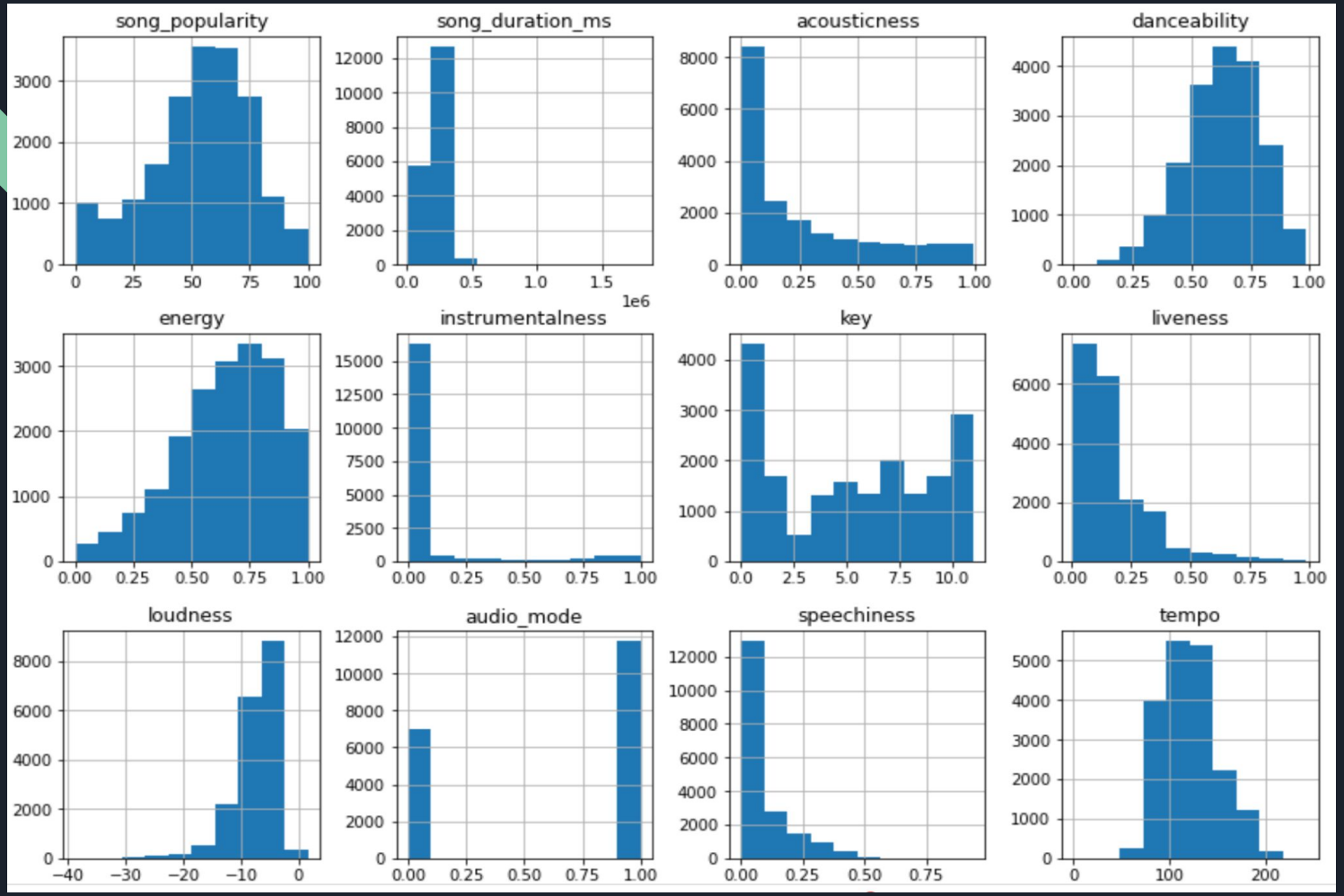
- Import NumPy, Pandas, and the train_test_split() function from scikit-learn to use for this assignment. We also imported Matplotlib to visualize the training and validation loss.

```
[68]  from google.colab import files
      uploaded = files.upload()
```

- It allowed to upload the data file called “song_data.csv”

```
# Load data
df = pd.read_csv("./song_data.csv", index_col=0).drop(["song_name"], axis = 1) #axis = 1 all the row line(가로) add u
# Visualize data
df.info()
df.hist(figsize=(14,14))
```

- This will visualize the data of the “song_data.csv”
- Next silde


song_popularity
song_duration_ms
1e6
acousticness
danceability
energy
instrumentalness
key
liveness
loudness
audio_mode
speechiness
tempo

Data Cleaning

`data.info()` shows that there are some missing values in the dataset. Also, we can see from the histogram that outliers exist for some features. Moreover, the range of different features has a huge gap: most in (0,1), but some are on the order of 1e2, or even 1e5. In the following you need to perform:

1. Drop rows that contain NULL values.
2. Remove outliers for features in ['song_duration_ms', 'instrumentalness'] based on interquantile range.
3. Scale ranges of independent variables using Z-score method, and normalize the range of dependent feature ['song_popularity'] from [0,100] to [0,1]

- As you can see above, we need to remove all the null value and outliers for song_duration_ms and instrumentalness.
- For the song popularity, we are going to normalize it into [0,1] by using the z-score method

```
# drop rows that have NULL value
# ========== YOUR CODE STARTS HERE ==========
#Drop all the rows with any NULL/NAT/NAN values
#data = df.dropna(axis=0, how = 'all', subset=None, inplace=False)
data = df.dropna()
data = data.reset_index(drop=True)

# ========== YOUR CODE ENDS HERE ============
'''
```

- I used the dropna() to remove all the null values

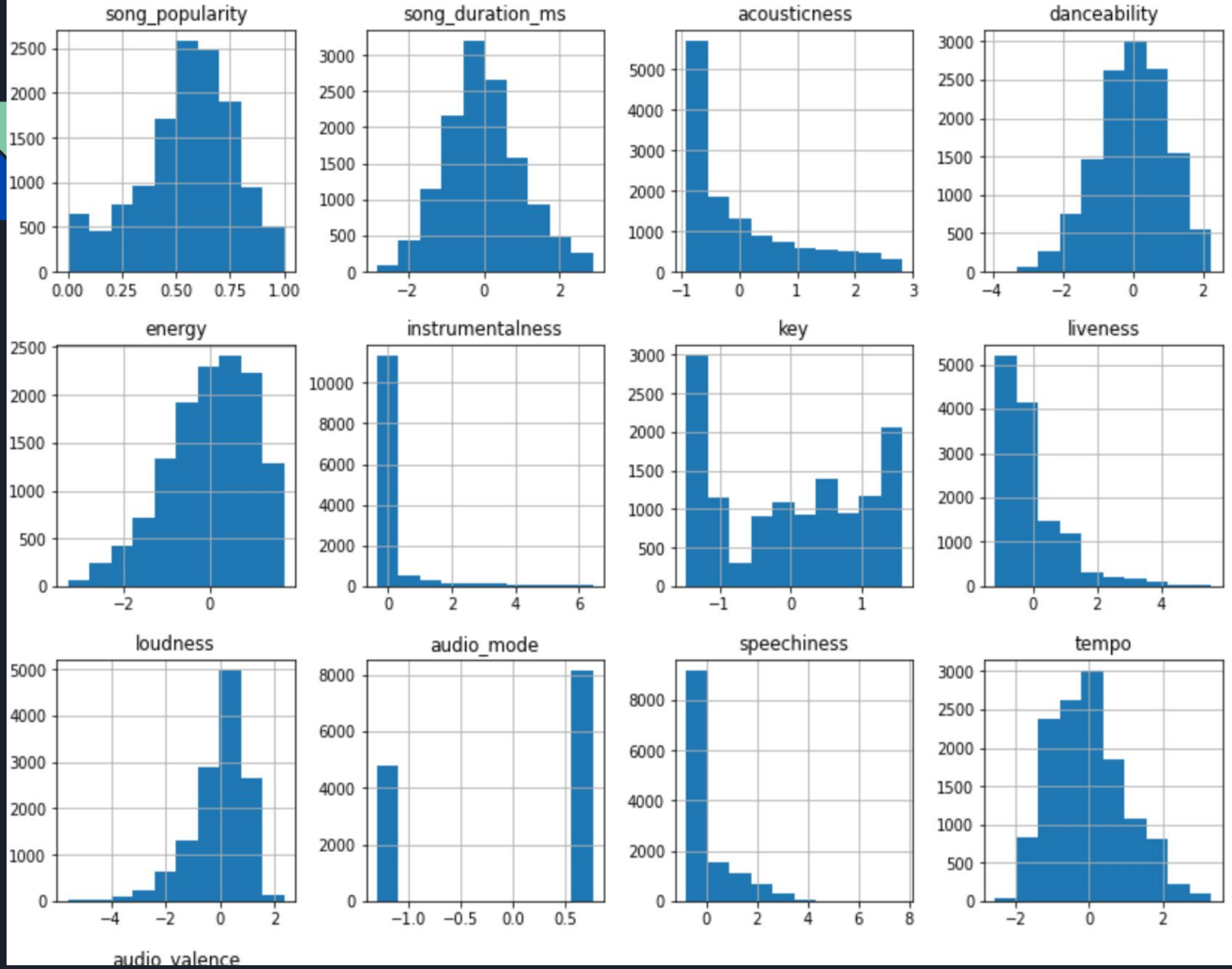
```
'''
Remove outliers for features in ['song_duration_ms', 'instrumentalness']
based on interquantile range. Here for each feature, we first sort data in an ascending order.
Let q1 and q3 be the data that ranks 25% and 75% respectively. We then let iqr = q3 - q1, and
compute
                a = q1 - iqr x 1.5,
                b = q3 + iqr x 1.5,
and remove the data out of the range [a, b].
Note: this can be realized with function quantile().
'''
for feature in ['song_duration_ms', 'instrumentalness']:

    # ========== YOUR CODE STARTS HERE ==========
    data = data.sort_values(feature, ascending = True)
    q1 = data[feature].quantile(0.25)
    q3 = data[feature].quantile(0.75)
    iqr = q3 - q1
    a = q1 - 1.5*iqr #lowerbound
    b = q3 + 1.5*iqr #upperbound
    data = data[(data[feature] > a) & (data[feature] < b)]

    # ========== YOUR CODE ENDS HERE ============
```

- In this picture, I used the formula that is given and apply it to remove the outlier in the data of “song_duration_ms and instrumentalness..

```
# ========== YOUR CODE STARTS HERE ==========
#scale all independent feature using z score normalization, and normalize dependent feature(song_popularity) range to [0,1]
for feature in data.columns:
    #normalize all the feature that is not song_popularity
    if feature != 'song_popularity':
        # (original value - mean of data)/standard deviation of data
        data[feature] = (data[feature] - data[feature].mean()) / data[feature].std()
#normalize dependent faeture(song_popularity) to [0,1]
data['song_popularity'] = (data['song_popularity'] - min(data['song_popularity'])) / (max(data['song_popularity']) - min(data['song_popularity']))
# ========== YOUR CODE ENDS HERE ============
```

- There are two things I need to accomplish.
- I used the formula (original value - mean of data)/standard deviation of data to normalize all the data except the song popularity
- And I used the formula data-min(data)/max(data)-min(data) to normalize the song popularity range to [0, 1]
- There are picture of the graph in the next slide after normalization.

dtype: object



```python
def test_split(data, test_size: float=0.3, seed=seed):
    '''
    train_test_split extracts 75 percents of the rows in the dataset as the training set
    and the remaining 25 percents of the data as the test set
    Use function train_test_split() to split test set.
    '''
    # ========== YOUR CODE STARTS HERE ==========
    #Python iloc() function enables us to select a particular cell of the dataset
    data_X = data.iloc[:,1::] #all the rows and columns except the first(song popularity)
    data_y = data.iloc[:,0:1] #all the rows and labels(song popularity)
    X_train_val, X_test, y_train_val, y_test = train_test_split(data_X, data_y, test_size = test_size, random_state = seed)
    # ========== YOUR CODE ENDS HERE ============
    # print("X_train_val: Wn", X_train_val)
    # print("X_test: Wn", X_test)
    # print("y_train_val: Wn", y_train_val)
    # print("y_test: Wn", y_test)
    return (X_train_val.reset_index(drop=True), #default the index
            y_train_val.reset_index(drop=True),
            X_test.reset_index(drop=True),
            y_test.reset_index(drop=True))
```

- This is the test_split function()
- This function will extracts 75 percents of the row in the dataset as the training set and the remaining 25 percents of the data as the test set.
- I used iloc() to take the certain parts that I want. So data_X will take all the rows and columns except the first(song popularity)
- And data_y will take all the rows and label(song_popularity)

```
def train_val_split(X_train_val, y_train_val, k=5, seed=seed):
    '''
    Use given index sets to generate k train and validation pairs. The return value should be
    a list whose components are tuples:
    [(X_train1, y_train1, X_val1, y_val1), ..., (X_traink, y_traink, X_valk, y_valk)]
    Here suppose data length is 10, the index sets could be like:
                    [[2,5], [1,6], [3,8], [9,4], [0,7]]
    '''
    # shuffle the dataset randomyly
    index_shuffle = list(X_train_val.index)
    random.shuffle(index_shuffle)
    index_split_list = [[index_shuffle[i+j] for i in range(0, len(index_shuffle), 5)  if i+j<len(index_shuffle)] for j in range(5)]
    train_val_pairs = [(None, None, None, None) for _ in range(k)]
```

- This is the function called train_val_split that will go to split the train and validation.
- Before we start, we need to set up things like we need to shuffle the data.
- And index_split_list will split the index into K fold which is going to be 5
- And store in the train cal pairs

```
# ========== YOUR CODE STARTS HERE ==========
for i in range(k):
  train = pd.DataFrame() #list x_train val = independent variable
  train_label = pd.DataFrame() #dependent variable
  val = pd.DataFrame(columns=list(X_train_val))
  val_label = pd.DataFrame()
  for j in range(k):
    indices = index_split_list[j] #  [[2,5], [1,6], [3,8], [9,4], [0,7]] j = 0 ----> [2, 5]
    if i == j:
      for index in indices:
        val = val.append((X_train_val.iloc[index]), ignore_index=True)
        val_label = val_label.append((y_train_val.iloc[index]), ignore_index=True)
    else:
      for index in indices:
        #train = pd.concat([train, X_train_val.iloc[:32]], ignore_index=True)
        train = train.append((X_train_val.iloc[index]), ignore_index=True)
        train_label = train_label.append((y_train_val.iloc[index]), ignore_index=True)
  train_val_pairs[i] = (train, train_label, val, val_label)
# ========== YOUR CODE ENDS HERE ==========
```

- I and j is going though the list and if i is equal to j, then validation.
- If i and j is not equal, then all the training set.
- And I just store all the values in the train val pairs.

```python
def MSE_loss(pred: np.ndarray, target: np.ndarray):
    # ========== YOUR CODE STARTS HERE ==========
    error = pred - target
    squareE = error ** 2
    mseL = squareE.sum()/(2 * len(pred))
    return mseL
    # ========== YOUR CODE ENDS HERE ============

def gradient(X: np.ndarray, y: np.ndarray, theta: np.ndarray):
    # ========== YOUR CODE STARTS HERE ==========
    grad = (X.T) @ (X @ theta - y) / X.shape[0]  # from the piazza
    return grad
    # ========== YOUR CODE ENDS HERE ============

# Specify epoch and learning rate
# ========== YOUR CODE STARTS HERE ==========
num_epochs = 100
learning_rate = 0.01
```

the students' answer, *where students collectively construct a single answer*

I suspect that your equation for gradients is incorrect. The correct equation is below. It is not identical in form to the one from the professor, but when I derived it myself, I found that I prefer this because it minimizes transposes.

$\text{Gradient} = \frac{1}{m}((X\theta - y)X)$,
where X is an m row, n column matrix ($m \times n$).

I am also initializing θ as a vector of zeroes, so that is fine.

The gradients should start off in the order of 1e-2 and they will quickly become as miniscule as you describe (as soon as the 2nd epoch, depending on your learning rate and initial θ).

- And I also used the gradient formula that was on the piazza.

- I just applied the formula that was on the piazza to the mse _loss function

Notice the formula for MSE loss, which stands for **Mean Squared Loss:**

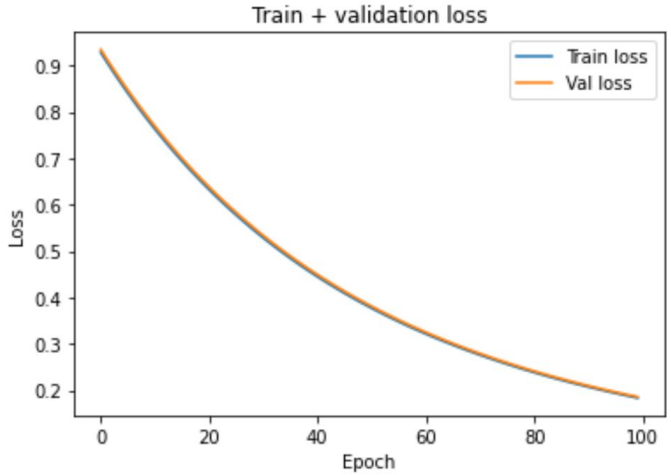
$$MSE = \frac{1}{2N}\sum_{i=1}^{N}(pred - target)^2$$

```
# Translate dataframe to numpy
# ========== YOUR CODE STARTS HERE ==========
X_train, y_train, X_val, y_val = X_train.to_numpy(), y_train.to_numpy(), X_val.to_numpy(), y_val.to_numpy()
X_train = np.c_[np.ones(len(X_train)), X_train]
X_val = np.c_[np.ones(len(X_val)), X_val]
# ========== YOUR CODE ENDS HERE ============
```

- This is to translate the data frame that I have used for most of my code to visualize the data to the numpy
- And I used .to_numpy() for each
- And I used np.c because of the format of the numpy.

```
>>> np.c_[np.array([1,2,3]), np.array([4,5,6])]
array([[1, 4],
       [2, 5],
       [3, 6]])
```

```
Final train loss for the best model: 0.1837660522356303
Final validation loss for the best model: 0.18575221950604417
Parameters for the best model:
 [[ 0.45339326]
 [ 0.08043035]
 [ 0.40101302]
 [ 0.29404897]
 [ 0.28387923]
 [ 0.16610985]
 [ 0.15080198]
 [ 0.11823927]
 [ 0.01710599]
 [ 0.18283374]
 [ 0.28963007]
 [ 0.06313664]
 [-0.12853278]]
```




```
Test loss: song_popularity     0.18089
```

This is what I got result

- Final train loss : 0.183
- Validation loss: 0.185
- Test loss: 0.18
- It takes about 2 mins and 30 sec in the process of the test val split