Overview

1. Abstract

Instead of using the given codes, I wrote most of the codes by myself to gain hands on experience.

Keywords: k-means, unbalanced dataset, metric trap, stratified sampling, resampling, epoch, batch, early_stopping

Below are the main work I have done in this project:

- Implemented the k-means clustering instead of using KMeans from sklearn.
- Computed Σ based on clusters instead of on the whole training dataset.
- Adopted the concepts of epochs, batches and early_stopping while training the model. Use batches during training procedure to speed up the convergence, and use early_stopping to reduce the unnecessary running time.
- Analyzed the performance of the model, and explored the possible root causes, such as highly unbalanced dataset, the metric trap, etc. Detailed information can be found in Scetion 6. Evaluation and Analysis.
- Several trials are conducted in Section 7. Improvements, aiming to bring my work one step further, mainly discussed adding progress bar to show the runtime for time consuming programs, and using stratified sampling to improve the quality of train/set split.

2. Data Preparation

Data visualization

Before preprocessing the dataset, I would like to take a visualized look at it first, so that I can have a general impression on each feature, and later on this would give me great guidance on training the model.

P.S. Most plots in this report are gained from the Jupyter since it is more friendly (also prettier) to plot, so they won't show up after running the main.py, if you have any question, feel free to contact me.

- information of features

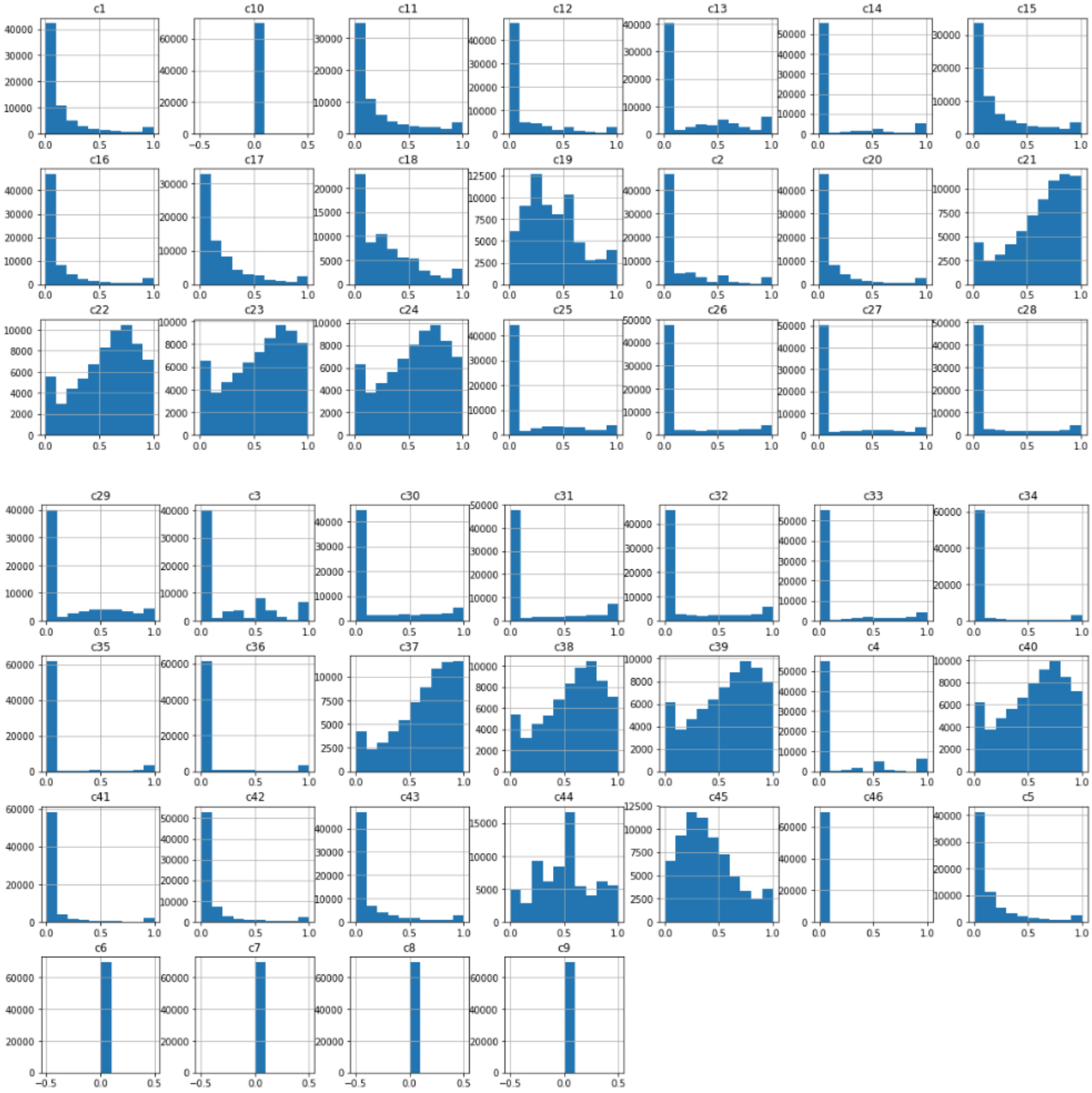
```
data.describe()
```

	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	...
count	69623.000000	69623.000000	69623.000000	69623.000000	69623.000000	69623.0	69623.0	69623.0	69623.0	69623.0	...
mean	0.161688	0.142533	0.252796	0.146082	0.165254	0.0	0.0	0.0	0.0	0.0	...
std	0.234702	0.255634	0.341220	0.311804	0.234517	0.0	0.0	0.0	0.0	0.0	...
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	...
25%	0.019157	0.000000	0.000000	0.000000	0.021930	0.0	0.0	0.0	0.0	0.0	...
50%	0.063291	0.000000	0.000000	0.000000	0.067818	0.0	0.0	0.0	0.0	0.0	...
75%	0.190455	0.200000	0.500000	0.000000	0.196005	0.0	0.0	0.0	0.0	0.0	...
max	1.000000	1.000000	1.000000	1.000000	1.000000	0.0	0.0	0.0	0.0	0.0	...

8 rows × 46 columns

The **std** in above table indicates that: from column 6 to column 10, their stds are all zeros, this would lead to a sigular matrix for Σ, since sigular matrix cannot be converted, these columns should be removed during data preprocessing.

- histograms of 46 features



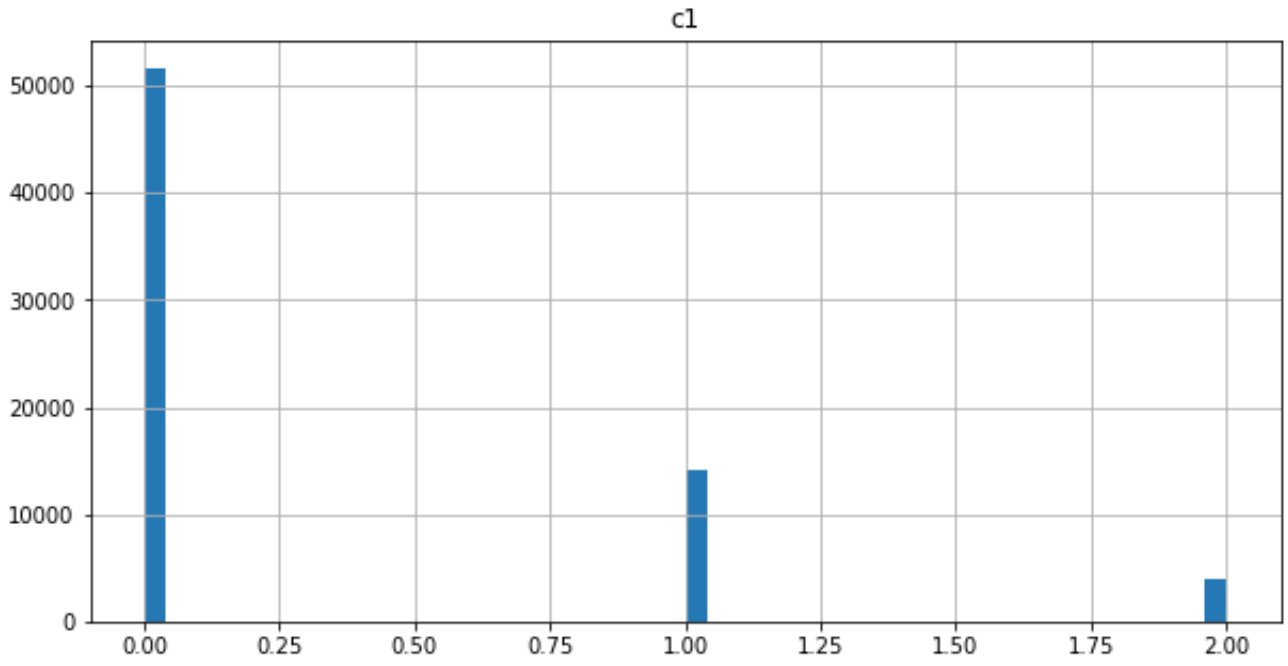
From the histograms of the 46 features we can see that: about 2/3 features mainly locate around 0. The distributions of column(feature) 18~19, 21~24, 37~39, 40, 44~45 are relatively more averagely located in [0,1], maybe these are more important features to train the ranking model.

- information of label

```
label.describe()
```

	c1
count	69623.000000
mean	0.313891
std	0.571259
min	0.000000
25%	0.000000
50%	0.000000
75%	1.000000
max	2.000000

- histograms of label



The label has three values 0, 1 and 2. While there are more than 50,000 '0's, there are only about 5,000 '2's, which means the dataset is very unbalanced, this would increase the difficulty for the model to capture the label of 2.

Data preprocessing

- Read data

The 'data.csv' and 'label.csv' above are used to visualize the dataset, below use 'Querylevelnorm*t.csv' and 'Querylevelnorm*X.csv' to get numpy arrays.

```
y = GetTargetVector(r'Querylevelnorm_t.csv')
x = GenerateRawData(r'Querylevelnorm_X.csv')
```

- Delete all zero columns.

To avoid the sigular matrix which is not reversible, delete all zero columns, I also delete the 45th column since it is all 0 except only one '1'.

```
x = np.delete(x, [5,6,7,8,9,45], axis=1)
```

- train / test split

train dataset: the first 80%, validation dataset: another 10%, test dataset: the last 10%

3. K-means Clustering

Though sklearn has KMeans, since TA explained how k-means works, I would like to try writing it myself.

- Initialize Centroids

Randomly select k different data points from X as the initial centroids for K-means clustering.

```
def init_k_means_centroids(X, k):
    init_centroids = np.zeros([k, X.shape[1]])

    np.random.seed(42) # make results repeatable to be able to debug
    while(True):
        X_permute = np.random.permutation(X) # shuffle x
        init_centroids = X_permute[0:k, :]
        unique_centroids = (np.unique(init_centroids, axis = 0)).shape[0]
        if(unique_centroids== k): # The initial centroids should be unique.
            break

    return init_centroids
```

- Find Memberships

Group data based on the minimum distance to the centroids.

```
# memberships holds the cluster number that each x belongs to

def find_memberships(X, centroids):
    k = centroids.shape[0]
    m = X.shape[0]

    distances = np.zeros([m,k])
    for i in range(k):
        diff = centroids[i] - X
        diff_sqr = diff ** 2
        diff_sum = np.sum(diff_sqr, axis = 1)
        distances[:,i] = diff_sum

    memberships = np.argmin(distances, axis = 1)
    return memberships
```

- Compute New Centroids

Compute the centroids of the clusters in each iteration, also save the final distribution of each cluster in *cluster_distribution.csv*.

```
def compute_centroids(X, M, init_centroids, max_iters):

    centroids = init_centroids

    for loop in range(max_iters):
        prev_centroids = centroids.copy()

        memberships = find_memberships(x_train, centroids)
        index = []
        for i in range(M):
            index_one_cluster = np.where(memberships == i)
            index.append(index_one_cluster)

        for i in range(0, M):
            if(X[index[i]].size > 0): centroids[i,:] = np.mean(X[index[i]], axis = 0)

        if(np.array_equal(centroids,prev_centroids)):
            print("centroids converge after iterations: ", loop)
            cluster_sizes_csv(memberships, M, "cluster_distribution.csv")
            break

    return centroids, memberships
```

- Do Iterations

Do K-Means clustering, make sure each centroid is unique.

```
def k_means(X, M, max_iters):
    has_nan = True

    # check if centroids has nan, if does, do k-means again
    while(has_nan):
        init_centroids = init_k_means_centroids(X, M)
        centroids, memberships = compute_centroids(X, M, init_centroids, max_iters)
        has_nan = np.isnan(centroids).any()
    return centroids, memberships
```

- Take a look at the distribution of k-means clusters.

After running k_means function, a **cluster_distribution.csv** is generated, showing how many samples in each cluster.

1	3933
2	9137
3	3246
4	4968
5	3836
6	2623
7	5537
8	12116
9	5925
10	4377

- Take a look at one centroids μ.

```
Mu[0]
```

```
array([0.10597079, 0.49796196, 0.11769488, 0.09366747, 0.11484697,
       0.20614201, 0.5056163 , 0.11487162, 0.08174747, 0.22047328,
       0.06933056, 0.38368481, 0.24171279, 0.40377657, 0.07060718,
       0.69719216, 0.59719787, 0.57035324, 0.57213089, 0.6214896 ,
       0.66613744, 0.62018146, 0.64817435, 0.14753779, 0.06664612,
       0.05293495, 0.05669102, 0.08717847, 0.02036448, 0.01975191,
       0.01666785, 0.72604961, 0.6231849 , 0.59717287, 0.59826849,
       0.13948572, 0.18094403, 0.17575636, 0.49296355, 0.39610435])
```

4. Tune Hyper-parameters

Number of Basis Functions M

If M is too large, the model would be very time consuming, and if M is too small, the model might not perform well. After many experiments on M's value, I decided to select 10 as the value of M.

```
M = 10
```

Regularization Term λ

Without regularization, the model is prone to overfitting if the higher order terms are assigned large weights to caputure noise in the data. After many experiments on η's value, I decided to select 0.9.

```
Lambda = 0.9
```

Learing Rate η

If learning rate is too big, the model might bounce back and forth, fail to reach the valley. And if the learning rate is too small, the model will spend too much time in training. After many experiments on η's value, I decided to select 0.01.

```
learning_rate = 0.01
```

Big Sigma Σ

Compute Σ on each cluster, leave out co-variances, only store its variances (diagonal values) according to the assignment.

Store big sigma in a 3-D matrix $\Sigma[M][N][N]$, M is the number of clusters, which is 10, and N is the number of features, which is 40 in my project.

- take a look at one cluster's sigma

```
np.diagonal(sigma[0])
```

```
array([0.22466707, 0.90380705, 0.37091698, 0.48483796, 0.24330074,
       0.64197072, 0.85209047, 0.35558608, 0.38491785, 0.66541518,
       0.14058702, 0.93163979, 0.53438586, 0.60835602, 0.14290848,
       0.49972936, 0.580085  , 0.66706982, 0.65161853, 0.56518043,
       0.6581744 , 0.81614118, 0.75459055, 0.56435087, 0.17738258,
       0.20362324, 0.14036143, 0.41530594, 0.060343  , 0.09900474,
       0.05313156, 0.37175971, 0.48276163, 0.56009252, 0.57853349,
       0.72769866, 0.78750904, 0.7339141 , 0.6440483 , 0.56219469])
```

The original sigma is relatively small, in order to contain more possible data, multiply it by sigma_factor to improve the model performance.

```
sigma_factor = 10
sigma = sigma * sigma_factor
```

Basis Function $\phi_j(x)$

Every continuous function in the function space can be represented as a linear combination of basis functions, just as every vector in a vector space can be represented as a linear combination of basis vectors. In this project, we use Gaussian Radial Basis Function.

Design Matrix $\Phi(x)$

Compute $\phi(x)$ for each x based on the μ and σ it belongs to.

```
phi_matrix[i][j] = gauss_activation(X[i], Mu[j], Sigma[j])
```

5. Train Model

Closed-Form Solution

Just calculate the weights according to the closed-form formula.

```
def w_closed_form(y, phi, Lambda):
    phi_t = phi.T
    I = np.identity(phi.shape[1])
    I[0,0] = 0
    w = np.dot(np.linalg.inv(np.add(np.dot(phi_t,phi),Lambda*I)),np.dot(phi_t,y))
    return w
```

SGD Solution

Adopt batch and early_stopping to speed up the convergence of model training in SGD solution.

early_stopping

- When the variance of last 100 epochs' losses is less than early_stopping, stop training.

```
# compute the variance of the last 100 epochs' losses
var_latest_epochs = lossHistory[-100:]

if np.var(var_latest_epochs) < early_stopping:
    num = epoch + 1
    print("Early stop at epoch: ", num)
    break
```

batch_size

Below are the advantages and disadvantages using batches.

- advantages
    - It requires less memory.
    - Typically, networks trains faster with mini-batches. That's because we update weights after each propagation. If we used all samples during propagation we would make only 1 update for the network's parameter.

- disadvantages
    - The smaller the batch, the less accurate estimate of the gradient.

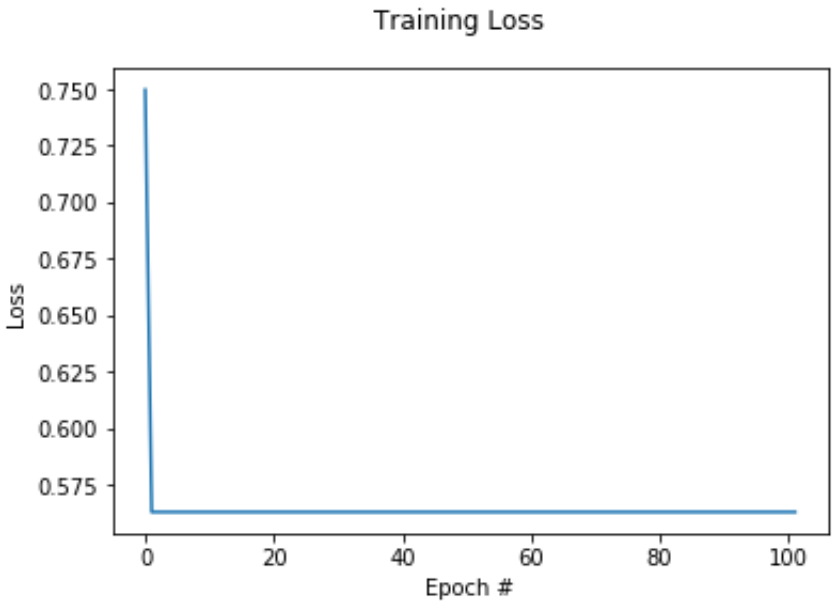
After many experiments on the size of batches, I decided to choose batch_size = 120 where it performs the best.

```
M              = 10
Lambda         = 0.9
learning_rate  = 0.01
sigma_factor   = 10
batch_size     = 120
epochs         = 500
early_stopping = 1e-05
```

Inpired by the project 1.1 which introduces epochs and batches, for the first time I have got to know the power of batches, so I wrote a simpler version of epochs and batches in this project.

```
Epoch 0/500
120/120[==========================] - loss: 0.7497255250459456
Epoch 1/500
120/120[==========================] - loss: 0.5627289108687523
Epoch 2/500
120/120[==========================] - loss: 0.5627265089001449
Epoch 3/500
120/120[==========================] - loss: 0.5627265223204907
Epoch 4/500
120/120[==========================] - loss: 0.5627265224024676
Epoch 5/500
120/120[==========================] - loss: 0.5627265224028173
Epoch 6/500
120/120[==========================] - loss: 0.5627265224028186
Epoch 7/500
120/120[==========================] - loss: 0.5627265224028186
Epoch 8/500
120/120[==========================] - loss: 0.5627265224028186
Epoch 9/500
120/120[==========================] - loss: 0.5627265224028186
Epoch 10/500
120/120[==========================] - loss: 0.5627265224028186
Epoch 11/500
120/120[==========================] - loss: 0.5627265224028186
Epoch 12/500
```

Also plot the training loss vs epochs to see how training loss decreases with epochs.



The plot indicates a **fast convergence**, running several epochs like 10 epochs is enough, and that would be super quick like in one second. **Batch really does a good job in speeding up the training procedure.**

6. Evaluation and Analysis

Evaluation on Closed-Form

```
------ Closed Form with Radial Basis Function ------
>>> Evaluating Erms (closed-form) on training dataset...
>>> Evaluating Erms (closed-form) on validation dataset...
>>> Evaluating Erms (closed-form) on testing dataset...


    E_rms Training      =  0.5739822321967568
    E_rms validation    =  0.5650963403815724
    E_rms Testing       =  0.6519620797148193
```

Evaluation on SGD

```
    E_rms Training      =  0.5851084019164987
    E_rms validation    =  0.5741864744680711
    E_rms Testing       =  0.6737940500642073
```

Analysis 1

Normally the model should predict the training dataset at best, but here the E_rms validation is better than the E_rms training, why?

```
threshold = 0.5
print(np.count_nonzero(y_sgd_train <= threshold) / y_sgd_train.shape[0])
print(np.count_nonzero(y_sgd_val <= threshold) / y_sgd_val.shape[0])
print(np.count_nonzero(y_sgd_test <= threshold) / y_sgd_test.shape[0])
```

```
0.9996588746454091
0.9998563631140477
1.0
```

```
print(np.count_nonzero(y_train == 0) / y_train.shape[0])
print(np.count_nonzero(y_val == 0) / y_val.shape[0])
print(np.count_nonzero(y_test == 0) / y_test.shape[0])
```

```
0.7452152680527129
0.7519390979603562
0.7022834984920293
```

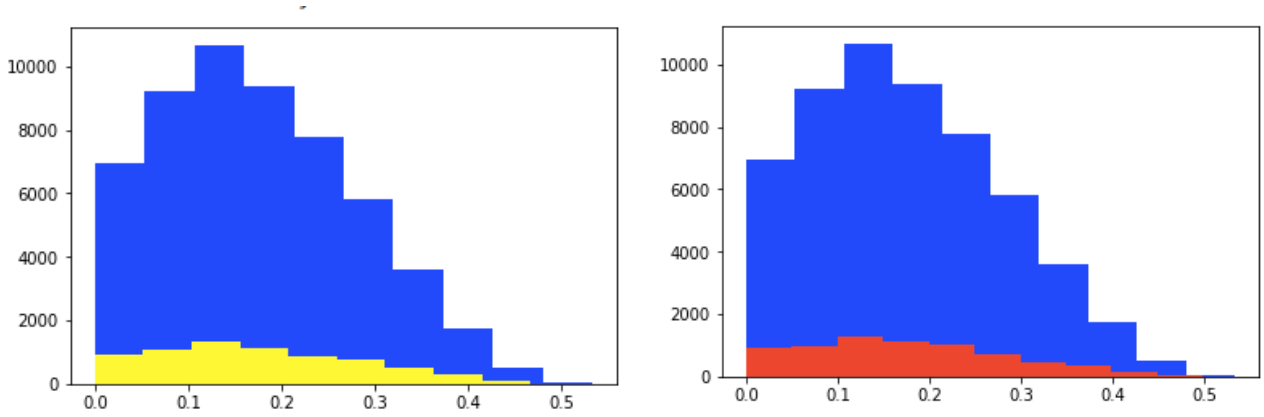
The results above shows that the predictions on training, validation, testing datasets are mainly located near zero, about 99.9% of the predictions are less than 0.5, this means the more zeros in the whole dataset, the less E_rms will be. Since the validation dataset has 0.6% more zeros than training dataset, its E_rms of validation would be a little lower than E_rms of training dataset.

One way to solve this problem is adopting stratified sampling, details can be found in section 7. Improvements.

Analysis 2

Is the model good? If not, why?

First take a look at predictions on training (blue), validation (yellow) and testing (red) datasets.



The predictions on validation and testing datasets share the similar distribution, in other words, they all predict lots of numbers in the range of [0, 0.5]. From this model, we are always told that there are no relevance or very poor relevance, in other words, we can't obtain the good relevance ranking, the one we truly want to focus on

when it comes to ranking problems, so, to be frankly, the model I get is useless.

I tried dozens of diffrent combinations of hyper-parameters, but none of them gives me a surprise, the E_rms remains high, the model seems like a stubborn person, always treats '0' as precious and refuses to learn some '1' and '2'.

Maybe it's time to think out of the box, instead of relying on tuning hyper-parameters, I decided to focus on the original datset.

The Metric Trap

The data we have is unbalanced, we want to predict the relevance, but nearly 75 percent of the dataset gives zero relevance, which **makes it hard for the model to learn minorities ('1' and '2') from the dataset**. Besides, the labels are discrete, while our model is linear regression, predicts decimals, **again makes it harder for the model to learn well**.

When dealing with unbalanced datasets, simpler metrics like accuracy **can be misleading**. In a dataset with highly unbalanced classes, if the classifier always "predicts" the most common class without performing any analysis of the features, it will still have a high accuracy rate, obviously illusory. For example, if the model is so aweful that it predicts all labels as zeros , then it still gets about 0.75 accuracy and around 0.6 E_rms.

So the critical task would be how to pre-process the original data so that it can serve much better in training the model. I think of one possible solution, resampling, details can be found in section 8. Future works.

7. Improvements

Here are some improvements on my previous work.

Progress Bar

In this project, some parts of the code (like k-means, computing design matrix) need a bit long time to run,, it is better if using a progress bar to indicate the remaining time. So I tried tqdm below.

```
# pip3 install tqdm
from tqdm import tqdm

# apply tqdm on the long loop
# for example
# in compute_centroids(X, M, init_centroids, max_iters)
# we would like to know how many iterations does it runs
for loop in tqdm(range(max_iters)):
```

Then we can have progress bar while runing some long-time program, the reson why I didn't add this in the main.py is, if someone's pc hasn't installed tqdm, then error would occur.

Stratified Sampling

Stratified sampling offers several advantages over simple random sampling. A stratified sample can provide greater precision than a simple random sample of the same size. In previous train/set split, I just subtract the first 80% samples as training dataset, which corresponds to "Random" column in table below.

```
from sklearn.model_selection import StratifiedShuffleSplit

data = pd.read_csv("input.csv")

sss = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in sss.split(data, data["c47"]):
    train = data.loc[train_index]
    test = data.loc[test_index]

def ratio(data):
    return data["c47"].value_counts() / data.shape[0]

res = pd.DataFrame({
    "Overall": ratio(data),
    "Stratified": ratio(train),
    "Random": ratio(data.iloc[0:55698]),
}).sort_index()

a = pd.DataFrame({"err_Stratified(%)": (res["Stratified"] - res["Overall"]) / res["Overall"] * 100})
b = pd.DataFrame({"err_Random(%)": (res["Random"] - res["Overall"]) / res["Overall"] * 100})
c = pd.concat([res, a, b], axis=1, sort=True)
```

```
c
```

	Overall	Stratified	Random	err_Stratified(%)	err_Random(%)
0	0.741594	0.741589	0.745215	-0.000734	0.488307
1	0.202921	0.202916	0.202000	-0.002821	-0.454056
2	0.055485	0.055496	0.052785	0.020133	-4.866001

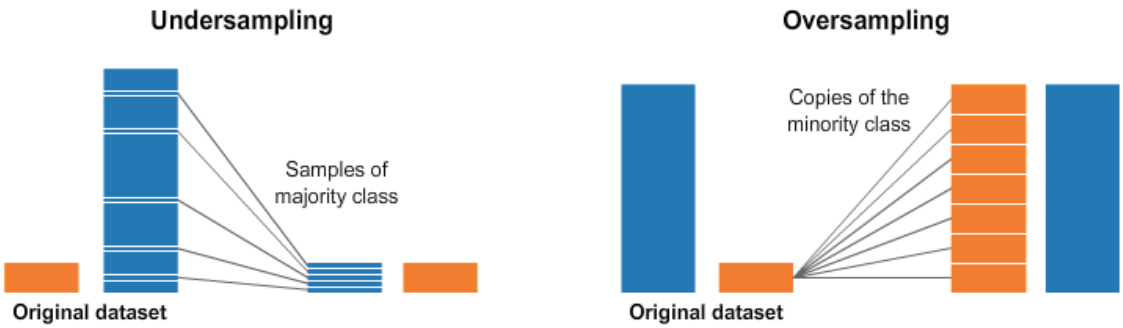
Compare *err_Random* with *err_Stratified*, we can see that *err_Stratified* is much smaller, meaning it is more representative for the original dataset. Especially for the small but important strata '2', there are **5.5496%** of '2' in the stratified training dataset, while there are **5.2785%** of '2' using random split methods, the sampling error is about **4.846% lower** with stratified sampling.

8. Future Works

One possible way to deal with highly unbanlanced dataset is to resample the original data, it can provide more "useful" different sample sets for learning process in some way.

Resampling

Resampling is a widely adopted technique for dealing with highly unbalanced datasets. It consists of removing samples from the majority class (under-sampling) or adding more examples from the minority class (over-sampling).



For the detailed implementation, I would like to put it off as future works for tight time budget.

The END, thank you for your time.