CIS 3715 Final Project Report

Leomar Durán

April 2022

Part I

Final Report

Project title and student names

- **Project title:** Using satellite imagery to train a model for identifying the type of landmarks.
- **Student names** (1)
  - Leomar Durán

1 Introduction section

1.1 Motivation

The sciences of geomatics and land surveying interest me as hobbies. I really enjoy the idea of collecting data about the terrain, whether it be rural or urban, and working with that data to find solutions to problems or even just for fun.

1.2 The Problem

Overview

An image of a terrain is given to a computer which will make a decision on the fly based on the type of terrain. We will train a model that will be used by the computer to identify this terrain.

This sort of decision might be involved in deciding if the terrain would be appropriate for developing a building thereon. A preliminary sweep by a machine may save on costs of having an engineer waste time looking for land to develop. Another example of making this decision may be helpful for automatic landing software that will be used to safely land aircraft on stable terrain. A third example is that combined with time series data, we can predict different types of weather-related and other natural phenomena, such as draughts, floods and earthquakes. The rain shadow affect is an important scenario where terrain plays a major role in weather.

Data science fundamentals

This problem involves multi-class classification using the linear and ridge regressions specifically. We will classify the terrain according to features such as whether the area is barren land, forested land (trees), grassland, and land with no special features for 4 disjoint classes.

We will evaluate the results using accuracy, recall, precision and the F1 score.

Project objective and constraints

For this project, we hope to train a model to learn different 4 disjoint classes of terrain, and then classify a test sample.

The algorithm that we pick has to deal well with the curse of dimensionality, as there will be $(28 \times 28)\mathrm{px/examples} \times 4\,\mathrm{channels/px} = 3136\,\mathrm{channels/examples}$.

Ideally the solution would also perform well for multiple clases, but this is less of an issue than dimensionality.

1.3 Related works

One of the historical approaches to this problem is that by Basu, Ganguly, Mukhopadhyay, *et al.* [1] themselves, who used a combination of computer vision and neural networks.

Computer vision: Use machine learning and neural networks to teach computers to see [2] compares computer vision with human sight as well as artificial intelligence, making the analogy that computer vision is to seeing as artificial intelligence is to thinking.

However, *Computer vision: Use machine learning and neural networks to teach computers to see* [2] also clarifies the disadvantages of computer vision to traditional machine learning models. Specifically, "[c]omputer vision needs lots of data. It runs analyses of data over and over until it discerns distinctions and ultimately recognize images." That is to say that computer vision has high time and spatial complexity. Because of the amounts of data required, it will require much storage, and the same compounded by the number of iterations that must be performed, training a computer vision model will require much more time. IBM explains that the scale needed for time and storage is such that few organizations have the necessary infrastructure, and as a result, many use a service such as IBM's to perform computer vision[2].

When it comes to neural networks, common issues include overfitting and underfitting[3][4]. Overfitting is when a model is too specific to the training data. As a result when the model is tested against the testing data, small differences can create large errors compared to the expected output[5]. Underfitting results from a model that is too simple and results in high errors in comparison to the expected results for both testing and training data[3]. In order to avoid both, Lawrence and Giles [4] suggests the more complex technique of backpropagation.

Rolf, Proctor, Carleton, *et al.* [6] provides another method of training a model on satellite data. Specifically, they used a hybrid system. First, there is a 18-layer convolutional neural network[7]. However, rather than producing a single output, it produces 2^9 features. There is a second convolutional neural network with a ReLu activation function, which produces 2^{13} features[6]. The values are then concatenated and placed in a linear regression with a ridge regularization function[6].

Sharma [8] explains the two main issues with the convolutional neural network, two of which form the basis of this model. One issue named are that the convolutional neural network is sensitive to variation in images, such as different in phase of the object being imaged, differences in lighting, and differences in positioning. The other issue is that convolutional neural networks are sensitive to even low levels of additive Gaussian white noise.

To satisfy what we have learned from past systems, we will work from the bottom up using a simpler system that will not overfit, iterating until we find the lowest system that will have good performance. This will also solve the issue of noise because this problem is complicated by overfitting. As for position and phase of the image, this is a much different problem to tackle and outside of the scope of this project. However, if the image were more more regular and you could expect a guideline, it could be used to properly orient the image using a rotation matrix.

2 Approach

2.1 Idea

Our primary idea is that we want to avoid overfitting or complex models. As explained in 1.3, overfitting is a very common issue with neural networks, which is usually resolved through back-propagation. However, we want to avoid complexity where possible. This will be our primary motivator in starting from the ground up.

Our progress is also available at the project repository.

2.2 Proposed work

We are given The SAT-4 airborne dataset[1]. This data is hosted by the Louisiana State University's Division of Computer Science and Engineering[1] and can be downloaded directly from the Google Drive[2] along with the SAT-6 airborne dataset, or by itself from Kaggle[3].

The dataset consists of 400,000 example tiles taken from satellite imagery originally from the National Agriculture Imagery Program (NAIP) dataset. Each example has features representing the pixels of a (28×28)px image multiplied by the channels for red, green, blue and near infrared (NIR). According to Basu, Ganguly, Mukhopadhyay, *et al.* [1], these tiles represent "different landscapes like rural areas, urban areas, densely forested, mountainous terrain, small to large water bodies", so these as a disjoint set of landscapes would make for appropriate labels.

Our proposed solution is a multi-class logistic regression. However, we intended to work from the ground up starting with linear regression and testing until we found something that worked good enough without overfitting as was a worry in past works. However, we had an issue with logistic regression.

This issue is that since the dataset has multiple classes, the specific encoding that was used in the data for those classes was one-hot encoding. Thus the labels were vectors. However, the `scikit-learn` package's logistic regression is designed to work specifically with scalar labels. Because of this the ridge regression seemed like the last more natural choice for this dataset.

Design and implementation challenges

A challenge to this solution is the size of the dataset. Because of its size (about 3 Gbyte), we expect long processing times. One possible solution to this challenge may be to reduce the datasize from 400,000 to a more managable number such as 20,000.

Another issue that we will run into is deciding the best way to split the classes for the multi-class classification.

Anticipated project outcomes and impacts

An anticipated outcome is a model that can identify the types of terrains accurately from the given dataset.

[1] <http://csc.lsu.edu/~saikat/deepsat/>

[2] <https://drive.google.com/u/0/uc?export=download&confirm=sWVM&id=0B0Fef71_vt3PUkZ4YVZ5WWNvZWs>

[3] <https://www.kaggle.com/datasets/crawford/deepsat-sat4>

3 Results

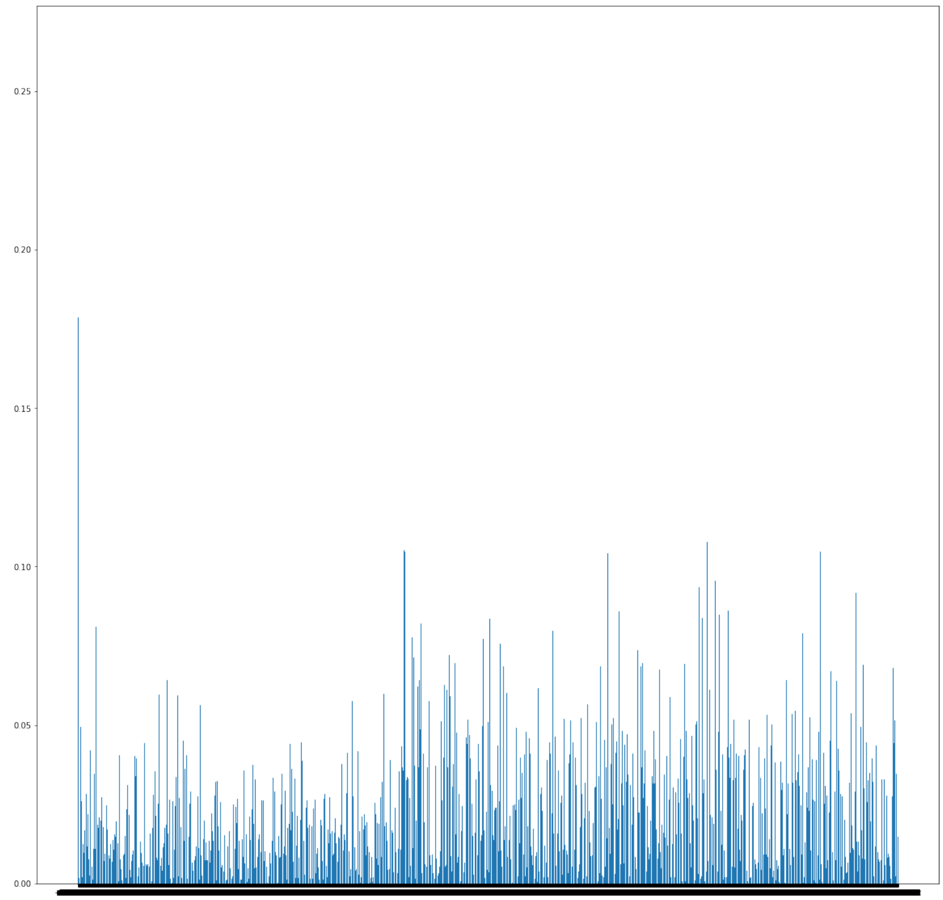


Figure 1: The weights including bias learned.

Fig. 1 shows the weights that were learned. The bias is most important in this model. Most of the weights are equally important, but there are a few that are more important, and many more that are not very important and could be reduced. I did not perform any feature reduction however.

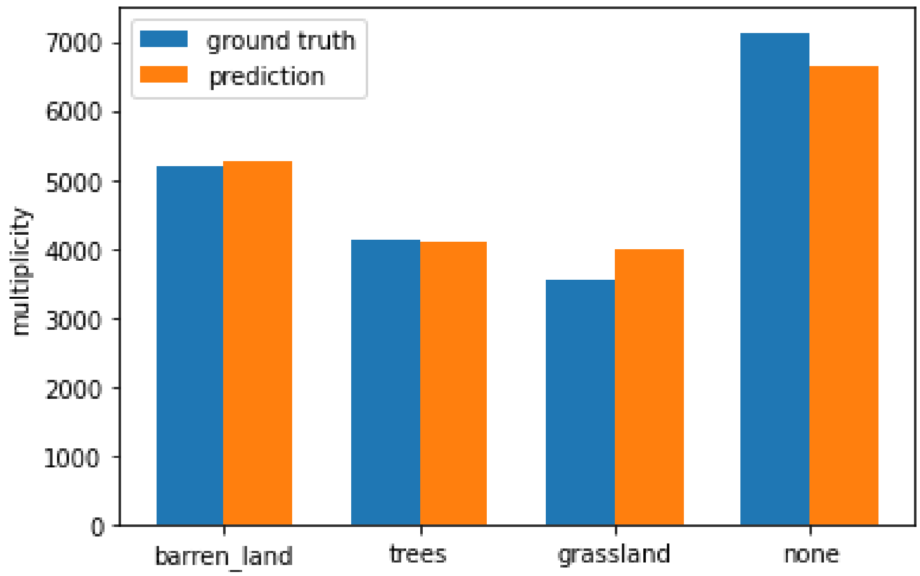


Figure 2: Bar plot of the multiplicities of the classes in ground truth and prediction.

In Fig. fig:multiplicity bar plot, we see a visual representation of the multiplicities of the terrain type classes. It seems like a close fit. However, all this is, is a visual representation of a count of each of the classes.

dividend	divisor	quotient
MAE train	MAE test	0.8367
RMSE train	RMSE test	0.8431
MAE test	MAD train	0.7619
RMSE test	σ train	0.7358

Table 1: Error ratios.

Table 1 shows us the results of the linear regression in numbers. Specifically, we found that the linear regression was slightly overfit by the two error ratios of 0.8367 and 0.8431. These represent an overfit because the errors of the training are less than the errors against the test data. Thus the ratio is less than 1.

This table also shows us that the mean absolute error is at about 0.76 mean absolute deviations around the median and that the root mean square error is at about 0.74 standard deviations, which mean that the model has a moderate predictive power since it is close to 1.

In their normal forms, mean absolute error and root mean square error give us a measure of the predictive power of the model[9]. Then we can use these last two normalizations because the more familiar standard deviation is also known as the root mean square. In fact, the root mean square error is the same formula as the root mean square, but using the corresponding predicted value for each label instead of the mean. Likewise we use the mean absolute deviation about the median to normalize the mean absolute error.

regularization λ		accuracy	recall	precision	F_1 score
[dB]	<1>				
none		0.586 65	0.449 40	0.606 50	0.503 64
−10	0.3162	0.587 15	0.449 22	0.607 58	0.503 74
−9	0.3548	0.587 20	0.449 29	0.607 78	0.503 81
−8	0.3981	0.587 30	0.449 43	0.607 84	0.503 92
−7	0.4467	0.587 35	0.449 35	0.607 94	0.503 88
−6	0.5012	0.587 35	0.449 27	0.607 94	0.503 82
−5	0.5623	0.587 55	0.449 08	0.608 11	0.503 65
−4	0.631	0.587 60	0.449 08	0.608 37	0.503 70
−3	0.7079	0.587 35	0.448 64	0.607 88	0.503 16
−2	0.7943	0.587 45	0.448 85	0.607 86	0.503 30
−1	0.8913	0.587 55	0.448 68	0.608 03	0.503 18
0	1.000	0.587 75	0.448 74	0.608 64	0.503 34
1	1.122	0.587 75	0.448 56	0.608 62	0.503 21
2	1.259	0.588 00	0.448 87	0.609 07	0.503 55
3	1.413	0.588 10	0.448 70	0.609 57	0.503 51
4	1.585	0.588 50	0.448 78	0.610 41	0.503 78
5	1.778	0.588 50	0.448 52	0.610 78	0.503 60
6	1.995	0.588 45	0.448 17	0.610 71	0.503 26
7	2.239	0.588 55	0.447 99	0.610 74	0.503 09
8	2.512	0.588 75	0.447 99	0.611 39	0.503 28
9	2.818	0.589 10	0.447 90	0.612 28	0.503 43
10	3.162	0.589 25	0.447 91	0.612 35	0.503 36

Table 2: Scores with different values of the regularization hyperparameter λ.

Table 2 shows us the scores for the different regularization hyperparameters λ of a ridge regression with the first being the original linear regression with no regularization. The models have somewhat bad recall with moderately fair precision. A visiual representation is provided by Fig.3 below.

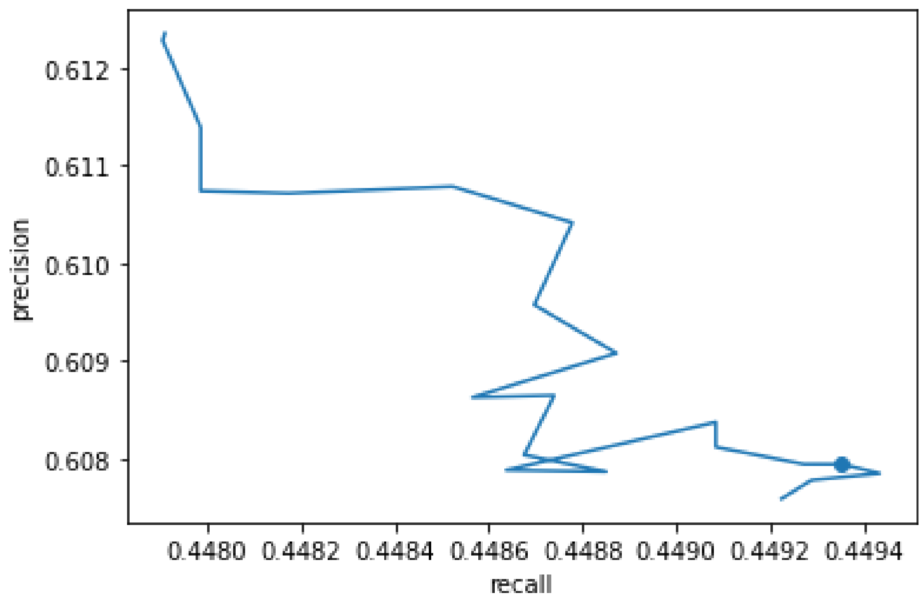


Figure 3: Precision versus recall by regularization hyperparameter λ with linear regression as a disc.

Hyperparameter $\lambda = -8\,\mathrm{dB} = 0.3981$ the F_1 score, representing the harmonic mean of recall and precision, maximizes at $F_1 = 0.503\,92$.

4 Conclusion

Overall, the project was not as successful as I had planned. The ridge regressions gave very low recall and a somewhat low precision.

I believe that the two issues that gave me the most difficulty were the large number of examples (400,000) and the labels which were each a 4-vector one-hot encoded.

The sampling program took a while to write because of the memory issues, and because I was having a hard time keeping up with other classes, which I neglected the project as a result.

As for the one-hot encoding, I was not sure how best to handle it. I considered after the project that label or ordinal encoding may be options. However that carries with it the connotation of ranking, which I'm not sure how the terrain type would rank, and it may depend on application, which is outside the scope of this project.

It may have also been interesting to attempt clustering to see if the examples fit the labels provided, or even if 4 labels was the optimal number.

As for the outcomes, I believe that we were able to prove that a somewhat reliable model is possible without resorting to computer vision or a neural network (convolutional or otherwise). I also got to learn many new techniques, to teach myself features of `pandas` and `sklearn` with which I was not familiar, and to explore the limits of those with which that I am familiar.

5 Acknowledgements

[1] S. Basu, S. Ganguly, S. Mukhopadhyay, R. Dibiano, M. Karki, and R. Nemani. "Deepsat - a learning framework for satellite imagery, acm sigspatial 2015." (2015).

[2] "Computer vision: Use machine learning and neural networks to teach computers to see." (2020), [Online]. Available: https://www.ibm.com/topics/computer-vision.

[3] S. Vignesh, "Overfitting in deep neural networks & how to prevent it," *Medium, Analytics Vidhya*, 2020. [Online]. Available: https://medium.com/analytics-vidhya/the-perfect-fit-for-a-dnn-596954c9ea39.

[4] S. Lawrence and C. L. Giles, "Overfitting and neural networks: Conjugate gradient and backpropagation," *IEEE Xplore*, 2005. [Online]. Available: https://ieeexplore.ieee.org/document/857823.

[5] H. Gao, "Lecture 7: Linear regression," *Temple University, CIS 3715/Principles of Data Science*, 2022.

[6] E. Rolf, J. Proctor, T. Carleton, *et al.*, "A generalizable and accessible approach to machine learning with global satellite imagery," *Nature Communications*, 2021. [Online]. Available: https://www.nature.com/articles/s41467-021-24638-z.pdf.

[7] "Resnet18: Resnet-18 convolutional neural network." (2018), [Online]. Available: https://www.mathworks.com/help/deeplearning/ref/resnet18.html.

[8] P. Sharma, "Disadvantages of cnn models," *OpenGenus IQ*, 2020. [Online]. Available: https://iq.opengenus.org/disadvantages-of-cnn/.

[9] "Root mean square," 2016. [Online]. Available: https://byjus.com/maths/root-mean-square/.

CIS 3715 Final Project Report Appendix

Leomar Durán

April 2022

Part II

Jupyter notebook of the project

1 Using satellite imagery to train a model for identifying the type of land-marks

1.1 Preprocess data

Now we may work with the data.

Start by importing necessary modules and setting up important constants.

[52]:
```
import pandas as pd                                           # for the dataframes
from math import *                                            # for sqrt
from statistics import *                                      # for mean
import numpy as np                                            # for linear algebra
import matplotlib.pyplot as plt                               # for various plots
from sklearn.linear_model import LinearRegression, Ridge      # for the learning␣
 ↪models
from sklearn.metrics import \
    mean_absolute_error, mean_squared_error                   # for evaluating␣
 ↪models
from sklearn.metrics import \
    accuracy_score, f1_score, recall_score, precision_score # for further␣
 ↪evaluating models
```

[78]:
```
# constants
X_TRAIN_FILENAME = r'dataset/X_train_sat4_samp20000.csv'     # filename of the␣
 ↪training dataset input
Y_TRAIN_FILENAME = r'dataset/y_train_sat4_samp20000.csv'     # filename of the␣
 ↪training dataset output
X_TEST_FILENAME = r'dataset/X_test_sat4_samp20000.csv'       # filename of the␣
 ↪testing dataset input
Y_TEST_FILENAME = r'dataset/y_test_sat4_samp20000.csv'       # filename of the␣
 ↪testing dataset output

R2_TOLERANCE = 0.81                                           # use 0.81 for r^2␣
 ↪for strong correlations
ACCURACY_TOLERANCE = 0.05                                     # maximum allowed␣
 ↪error for accuracy
```

Read in the files and do a high level inspection.

[3]:
```
# read in the training data
X_train = pd.read_csv(X_TRAIN_FILENAME, header=None, index_col=None)
y_train = pd.read_csv(Y_TRAIN_FILENAME, header=None, index_col=None)
```

[4]:
```
# data shape constants
(N_XAMPS, N_FEATS) = X_train.shape
(_, N_LBLS) = y_train.shape
# colors for grapphing
```

```
CHANNELS = (r'r', r'g', r'b', r'maroon')
N_CHANNELS = len(CHANNELS)
# number of pixels
NF_PIXELS = N_FEATS/N_CHANNELS
F_WIDTH = sqrt(NF_PIXELS)
F_HEIGHT = NF_PIXELS/F_WIDTH
# round number of pixels
NI_PIXELS = int(NF_PIXELS)
I_WIDTH = int(F_WIDTH)
I_HEIGHT = int(F_HEIGHT)

print(r"{} images of ({}x{})px x {} channels".format(N_XAMPS, I_WIDTH, I_HEIGHT,␣
 ↪N_CHANNELS))
```

```
20000 images of (28x28)px x 4 channels
```

[5]:
```
# combine training data features, labels
df_train = pd.concat([X_train, y_train], axis=1)
```

[6]:
```
# print shapes of X, y
print("X_train shape\t{}".format(X_train.shape))
print("y_train shape\t{}".format(y_train.shape))
print("combined shape\t{}".format(df_train.shape))
```

```
X_train shape   (20000, 3136)
y_train shape   (20000, 4)
combined shape  (20000, 3140)
```

[7]:
```
# print some basic information about the dataset
print('\n===data frame information===')
df_train.info()

# print its parameters
print('\n===data frame parameters===')
df_train.describe()
```

```

===data frame information===
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20000 entries, 0 to 19999
Columns: 3140 entries, 0 to 3
dtypes: int64(3140)
memory usage: 479.1 MB

===data frame parameters===
```

[7]:
```
                 0             1             2             3             4      \
count  20000.000000  20000.000000  20000.000000  20000.000000  20000.000000
mean     127.778700    123.954900    110.979800    158.810900    127.608900
```

```
std      42.825413    37.944932    35.705831    37.819527    42.946379
min       0.000000     3.000000     1.000000     4.000000     0.000000
25%      98.000000    99.000000    89.000000   140.000000    98.000000
50%     124.000000   122.000000   110.000000   166.000000   123.000000
75%     159.000000   148.000000   132.000000   185.000000   159.000000
max     255.000000   255.000000   255.000000   253.000000   244.000000

                5             6             7             8             9  \
count  20000.00000  20000.000000  20000.000000  20000.000000  20000.000000
mean     123.83680    110.863550    158.690150    127.607600    123.907450
std       38.08994     35.786297     37.861817     42.864149     37.943326
min        2.00000      0.000000      0.000000      0.000000      1.000000
25%       99.00000     89.000000    140.000000     98.000000     99.000000
50%      122.00000    109.000000    166.000000    123.000000    122.000000
75%      148.00000    132.000000    185.000000    158.000000    148.000000
max      255.00000    255.000000    252.000000    246.000000    255.000000

       ...          3130          3131          3132          3133  \
count  ...  20000.000000  20000.000000  20000.000000  20000.000000
mean   ...    111.151800    158.804500    128.173150    124.216950
std    ...     35.980267     37.691749     42.905264     37.907691
min    ...      0.000000      0.000000      0.000000      2.000000
25%    ...     89.000000    140.000000     99.000000    100.000000
50%    ...    110.000000    166.000000    124.000000    122.000000
75%    ...    133.000000    185.000000    159.000000    148.000000
max    ...    255.000000    254.000000    248.000000    251.000000

              3134         3135             0             1             2  \
count  20000.00000  20000.00000  20000.000000  20000.000000  20000.000000
mean     111.20020    158.91350      0.263800      0.201600      0.178600
std       35.64033     37.59736      0.440703      0.401205      0.383027
min        0.00000      3.00000      0.000000      0.000000      0.000000
25%       89.00000    140.00000      0.000000      0.000000      0.000000
50%      110.00000    166.00000      0.000000      0.000000      0.000000
75%      133.00000    185.00000      1.000000      0.000000      0.000000
max      255.00000    245.00000      1.000000      1.000000      1.000000

                  3
count  20000.000000
mean       0.356000
std        0.478827
min        0.000000
25%        0.000000
50%        0.000000
75%        1.000000
max        1.000000
```

```
[8 rows x 3140 columns]
```

We can see from these summaries that all 3140 columns, features and labels, are `int64`, and thus numerical features.

```
[8]: # calculate the number of missing values
n_missing = df_train.isnull().sum()

# print the number missing for each column,
# but ignore 0s since there are so many columns
print(r'===# missing values per column===')
print(n_missing[n_missing != 0])
```

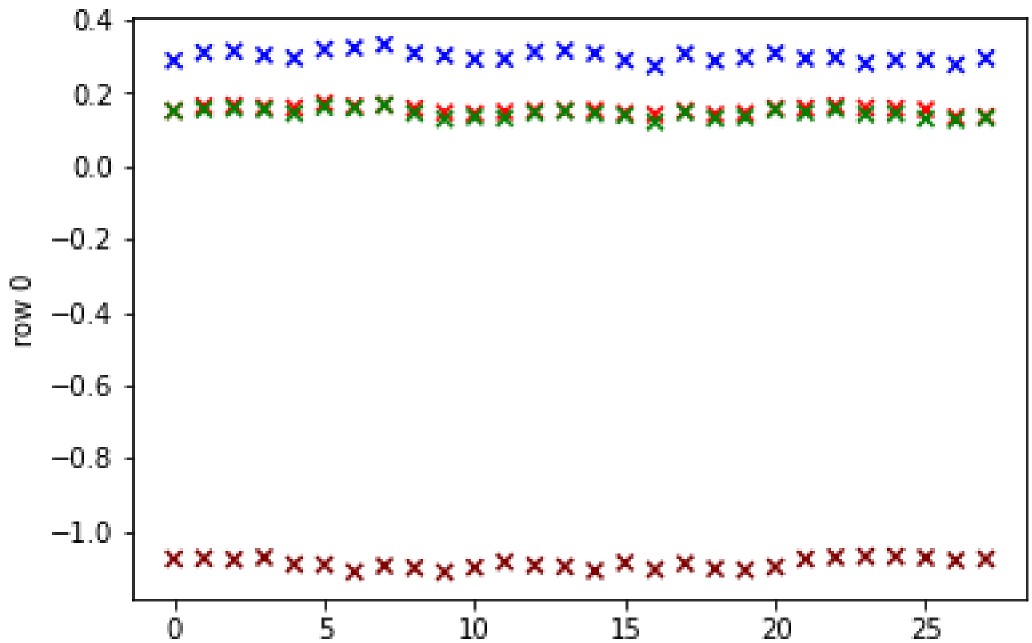
```
===# missing values per column===
Series([], dtype: int64)
```

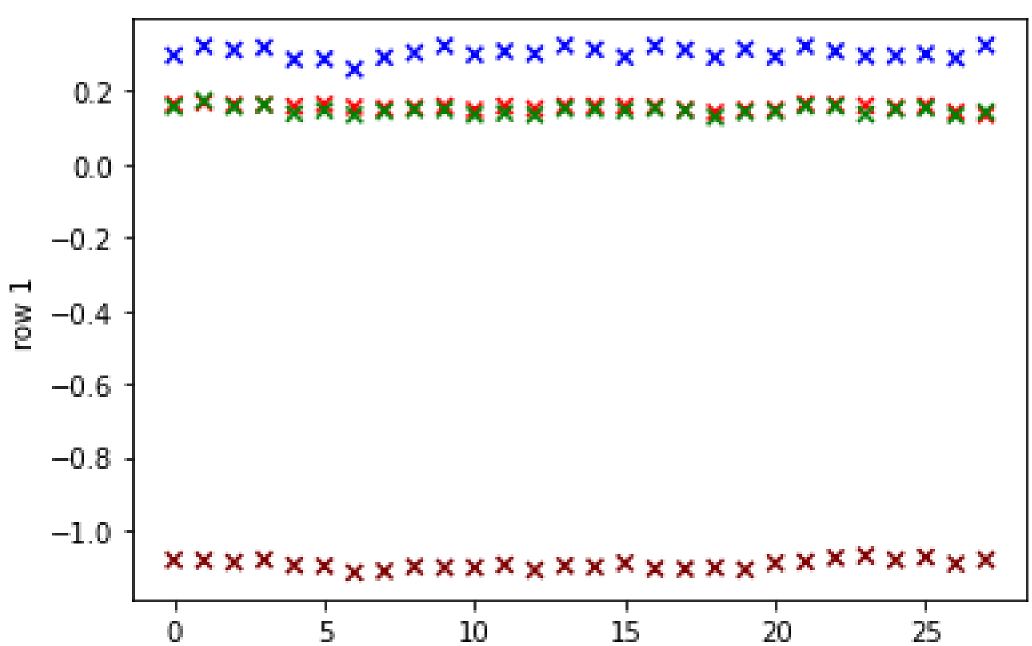
Additionally, we find that there are no columns with missing values. Let's inspect the distributions of each feature and the label.

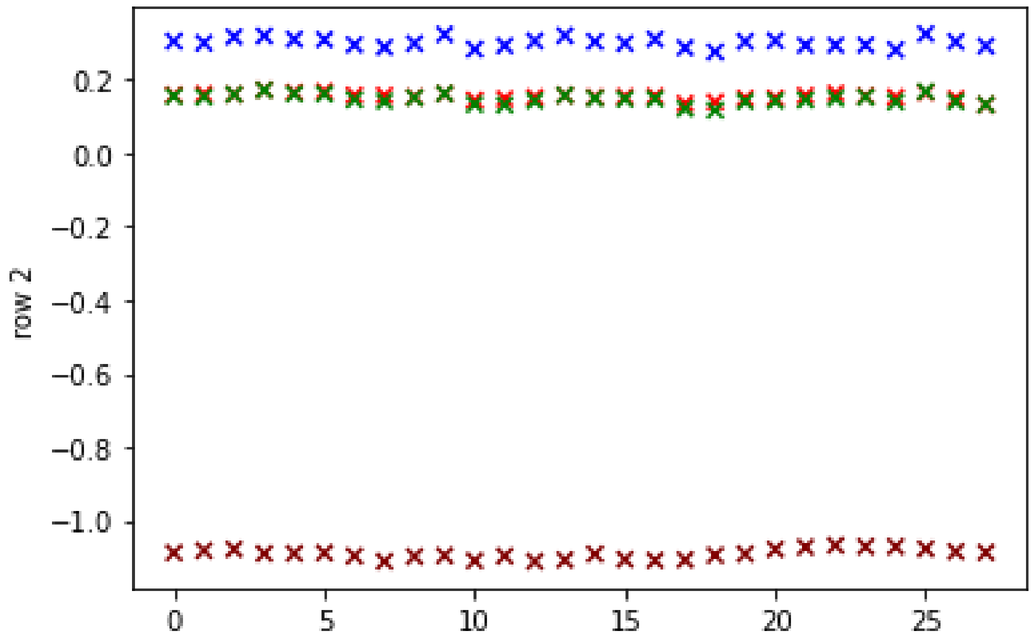
Since there are so many features (3136), we will plot the skewness of each feature by pixel row, rather than a histogram.

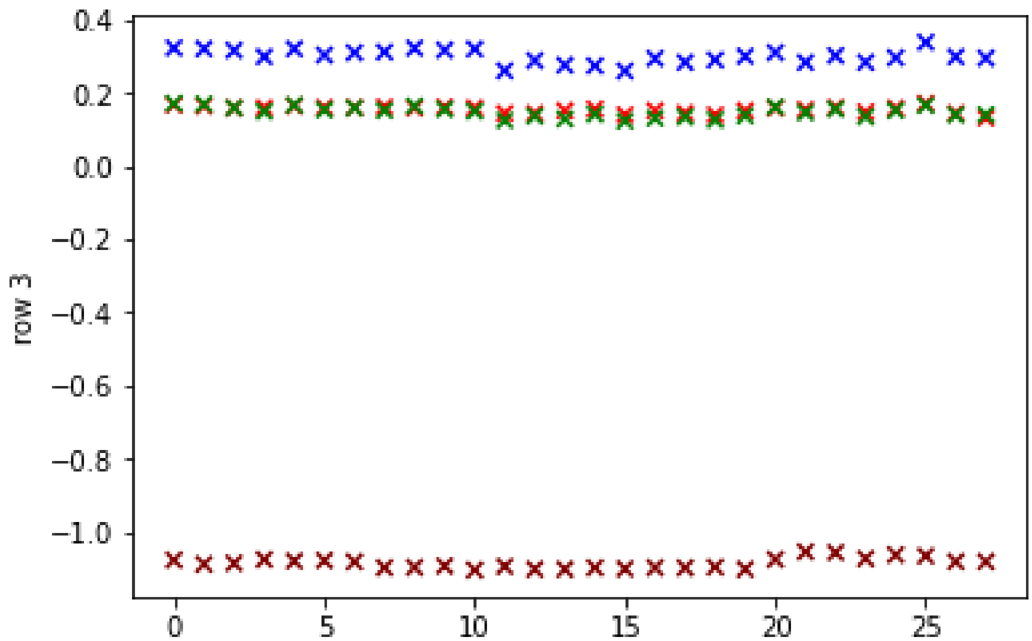
```
[9]: # centralize the data frame
central_X = X_train - X_train.mean()
# calculate standard deviations
X_std = X_train.std()
# calculate the skews of each column
skews = N_XAMPS*(central_X**3).sum()/((N_XAMPS - 1)*(N_XAMPS - 2)*X_std**3)
```

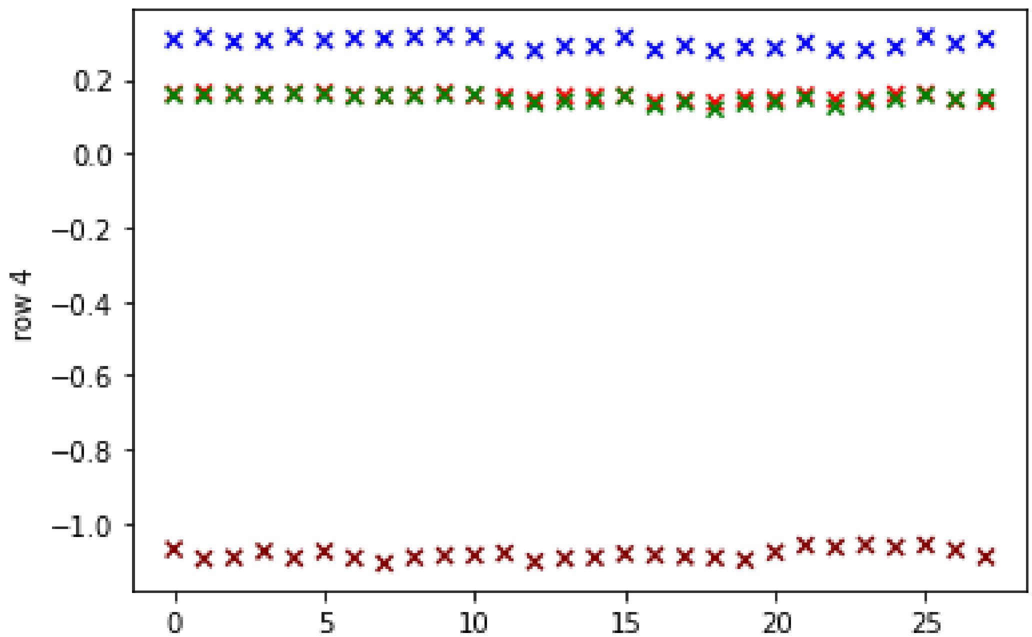
```
[10]: # let's scatter graph the features
# 28x28, color code RGB
# x's in scatter graph
scatter_x = sum(([k]*N_CHANNELS for k in range(I_WIDTH)), [])
# colors for the scatter graph
scatter_colors = (CHANNELS*I_WIDTH)
# its size
ROW_SIZE = (N_CHANNELS*I_WIDTH)

# loop through pixel rows
for i_row in range(0, I_HEIGHT):
    # offset in data frame for this row
    offset = (i_row*ROW_SIZE)
    # create the row as a list
    row = list(skews[X_train.columns[offset:(offset + ROW_SIZE)]])
    # plot the row
    plt.scatter(scatter_x, row, c=scatter_colors, marker='x')
    plt.ylabel("row {}".format(i_row))
    plt.show()
# next i_row
```

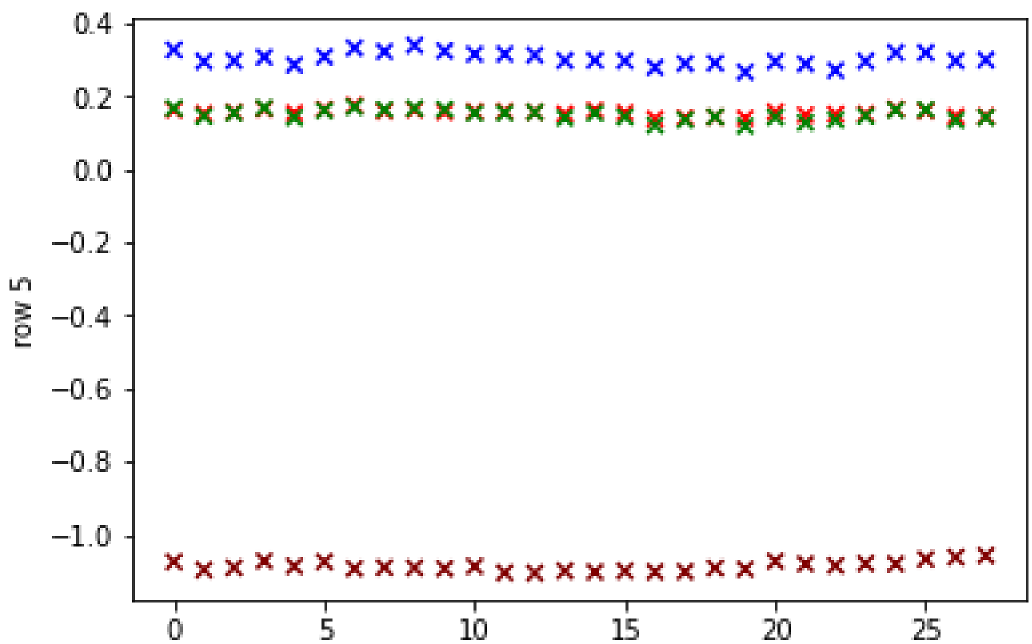

0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
row 0
0
5
10
15
20
25

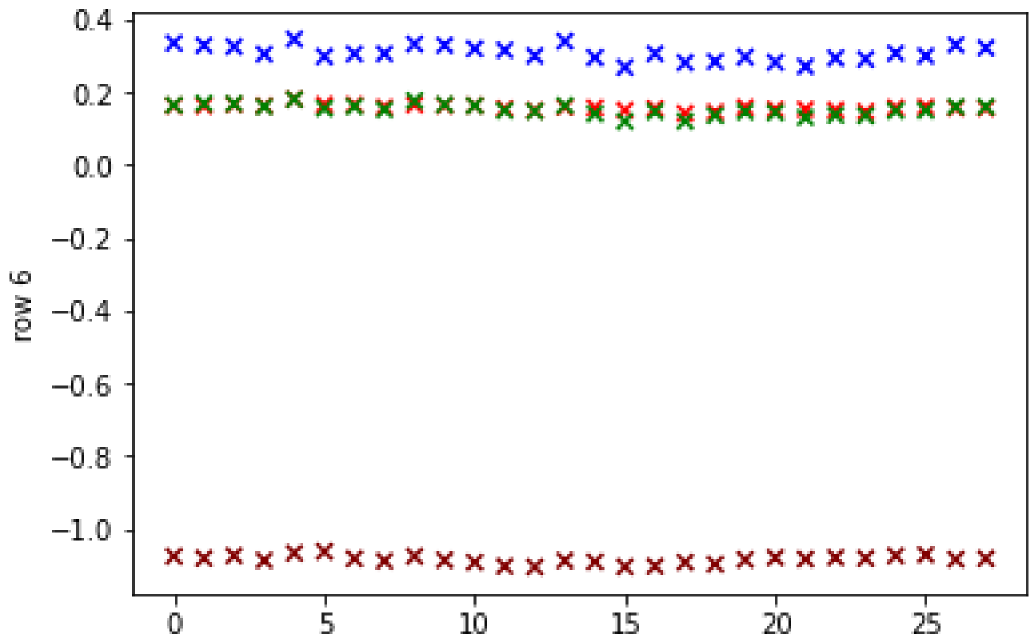

0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
row 1
0
5
10
15
20
25

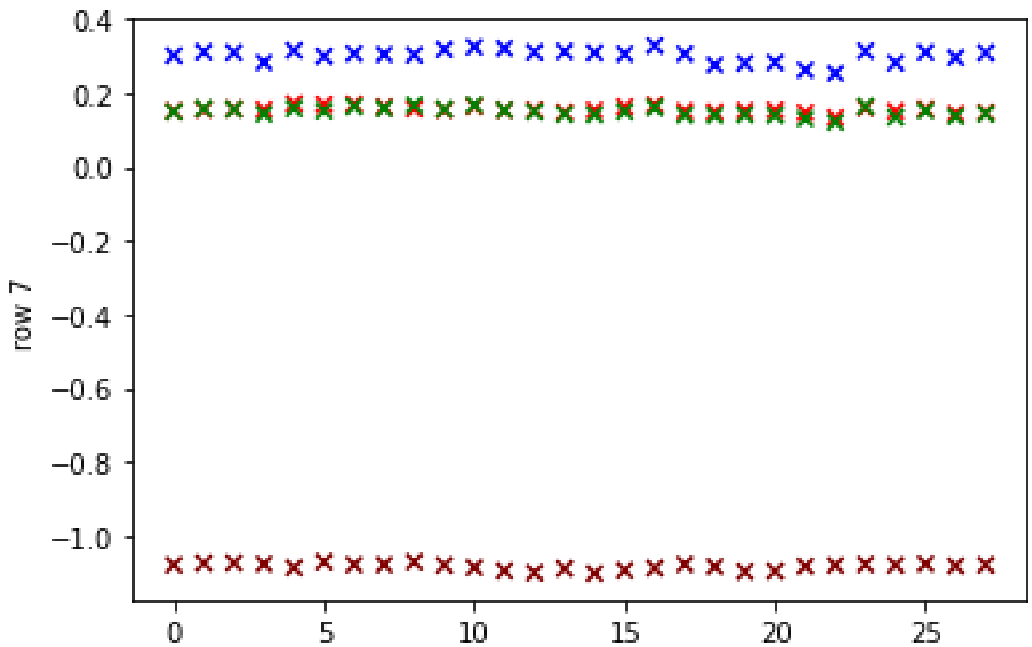

row 2
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

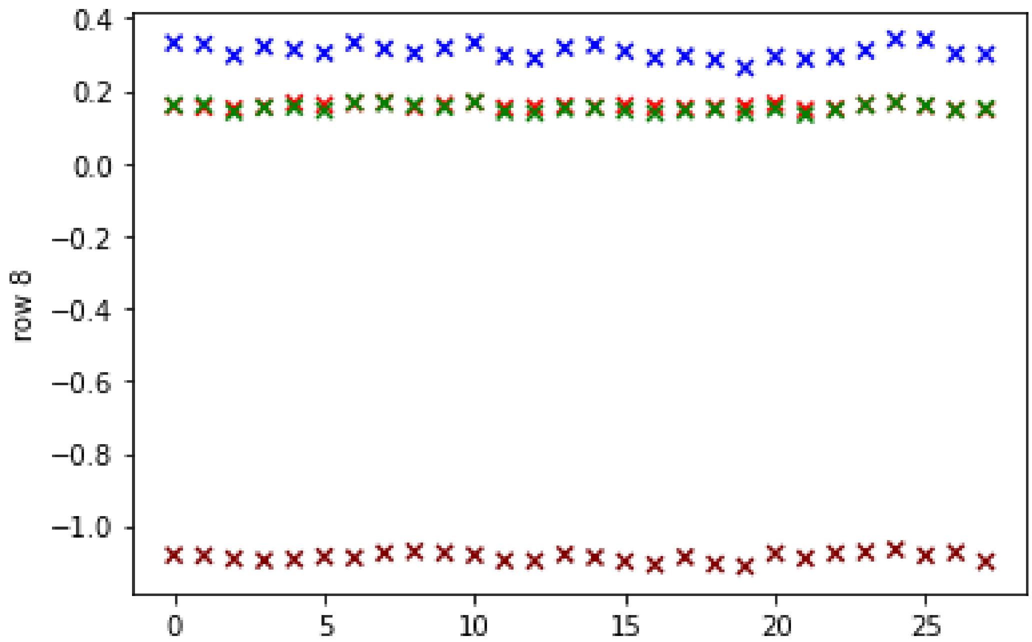

row 3
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

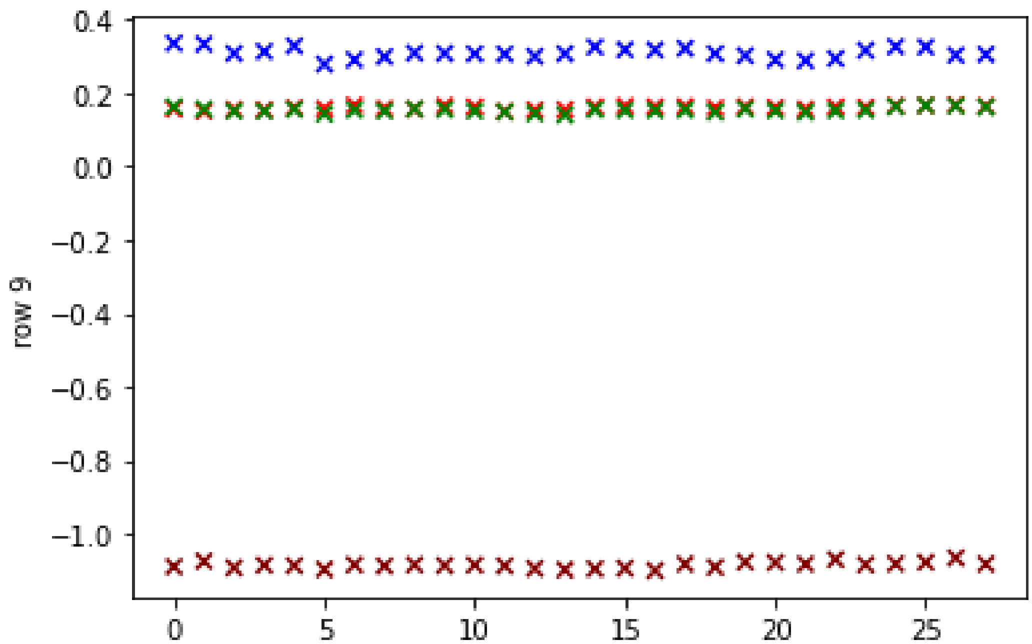

row 4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

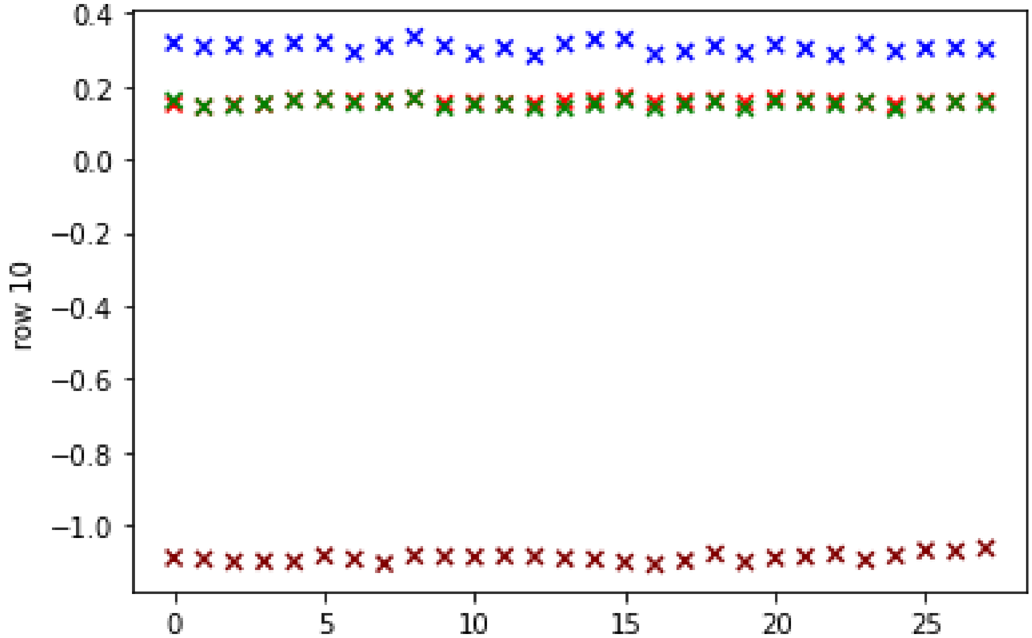

row 5
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

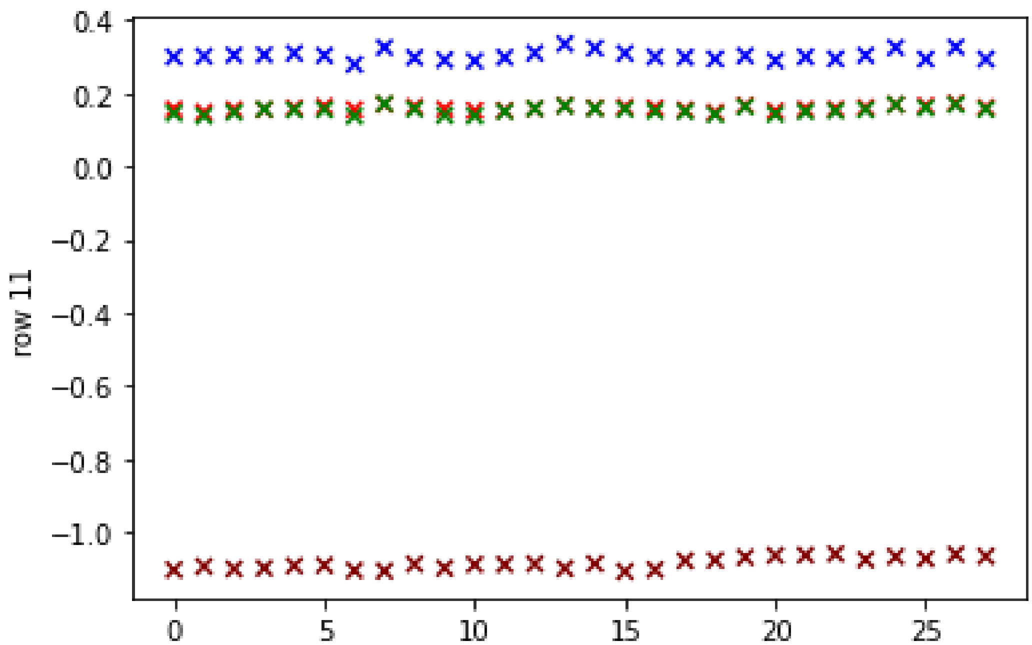

row 6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

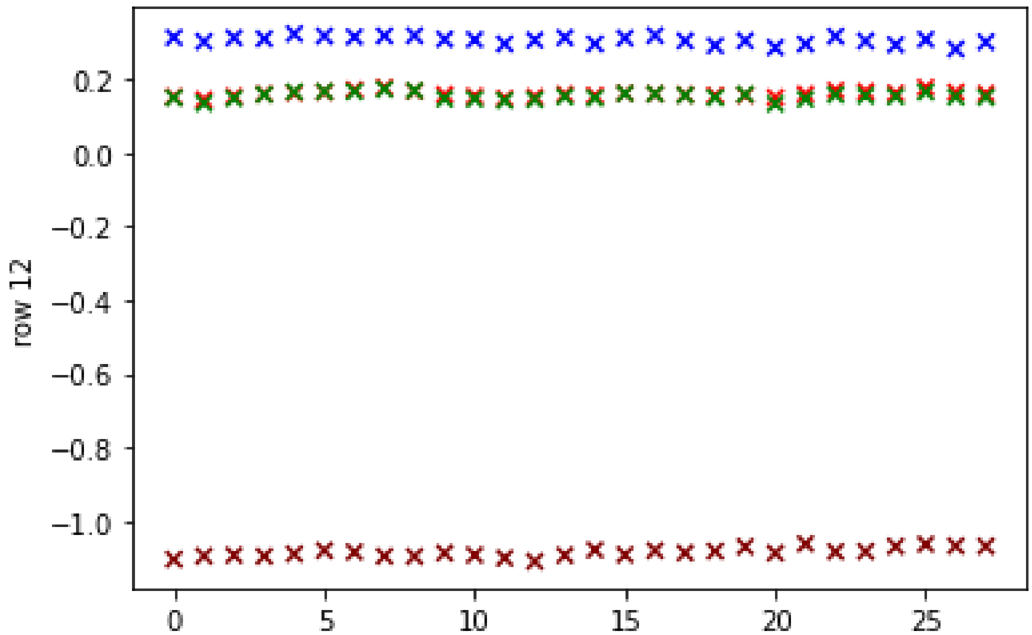

row 7
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25


row 8
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

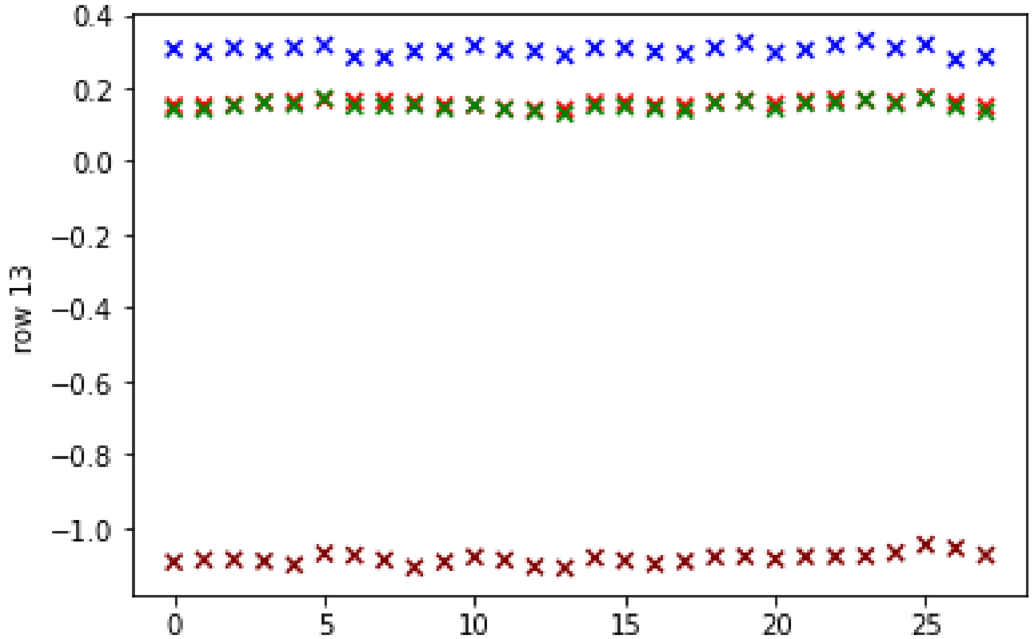

row 9
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25


row 10
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

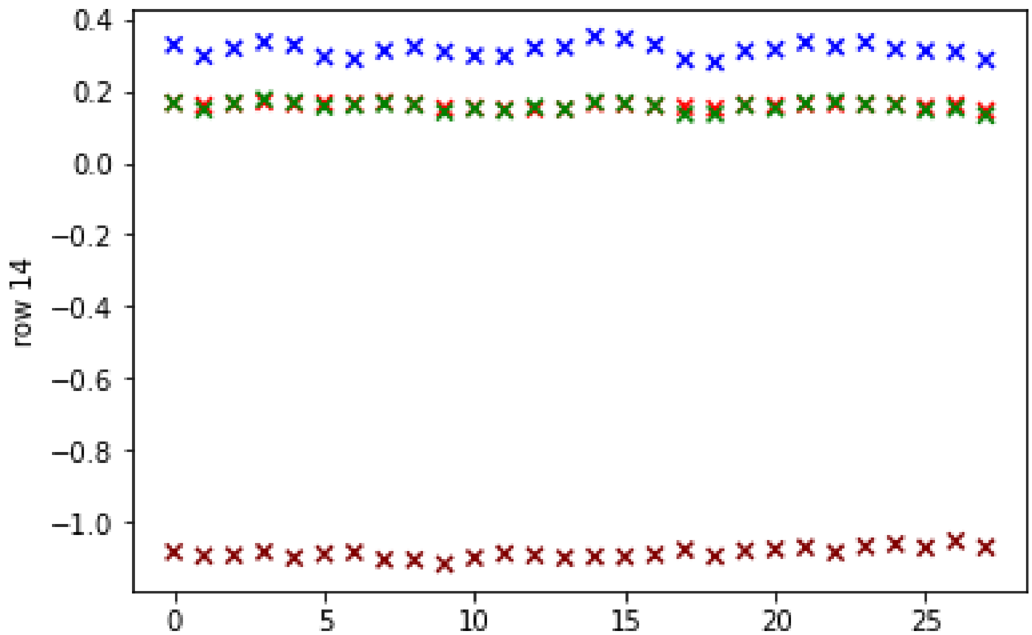

row 11
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25



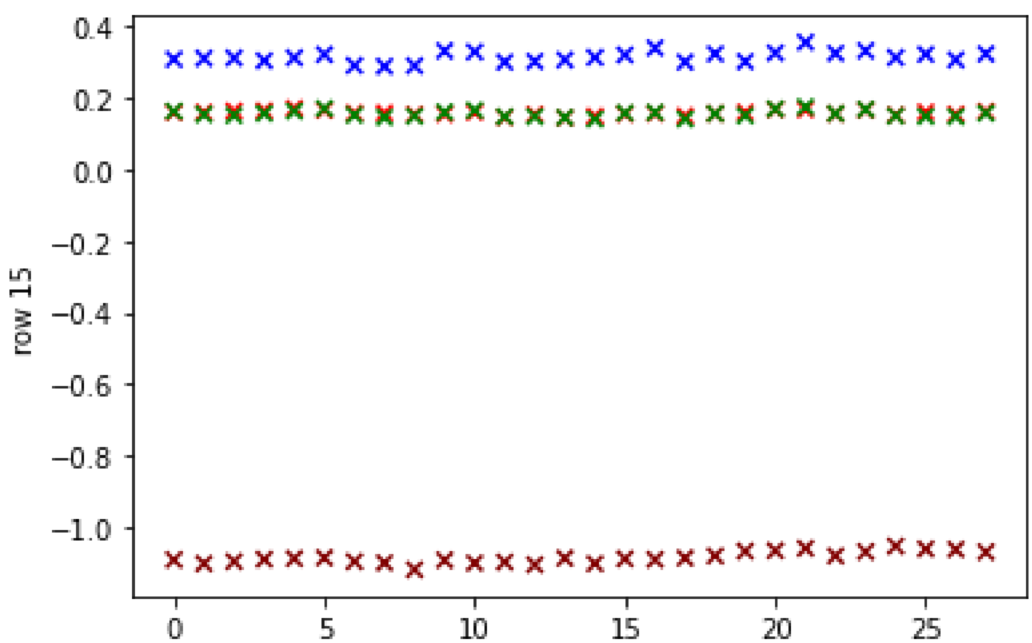
row 12
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25



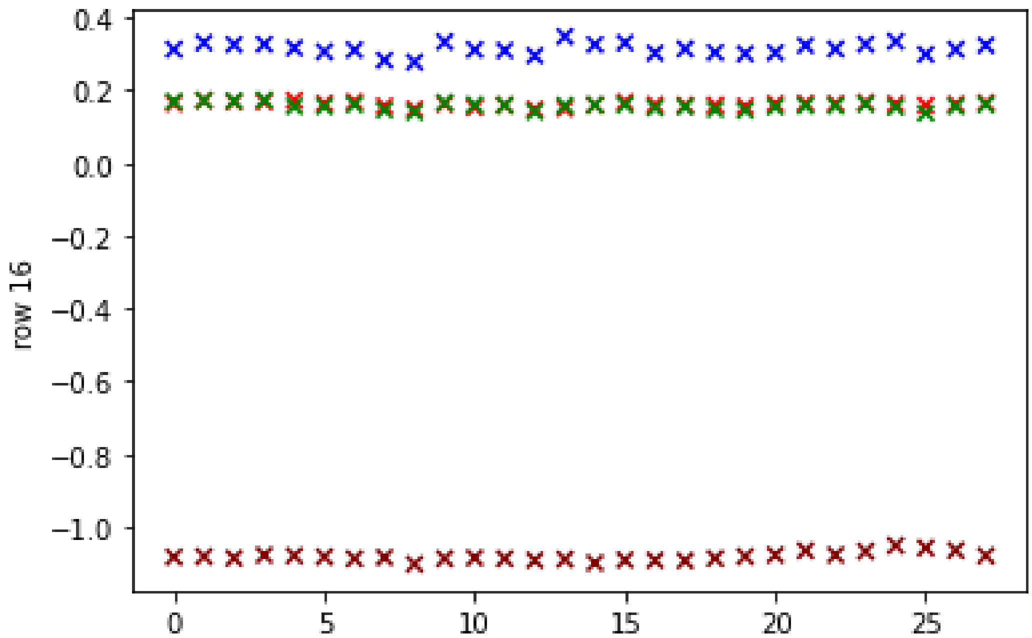
row 13
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

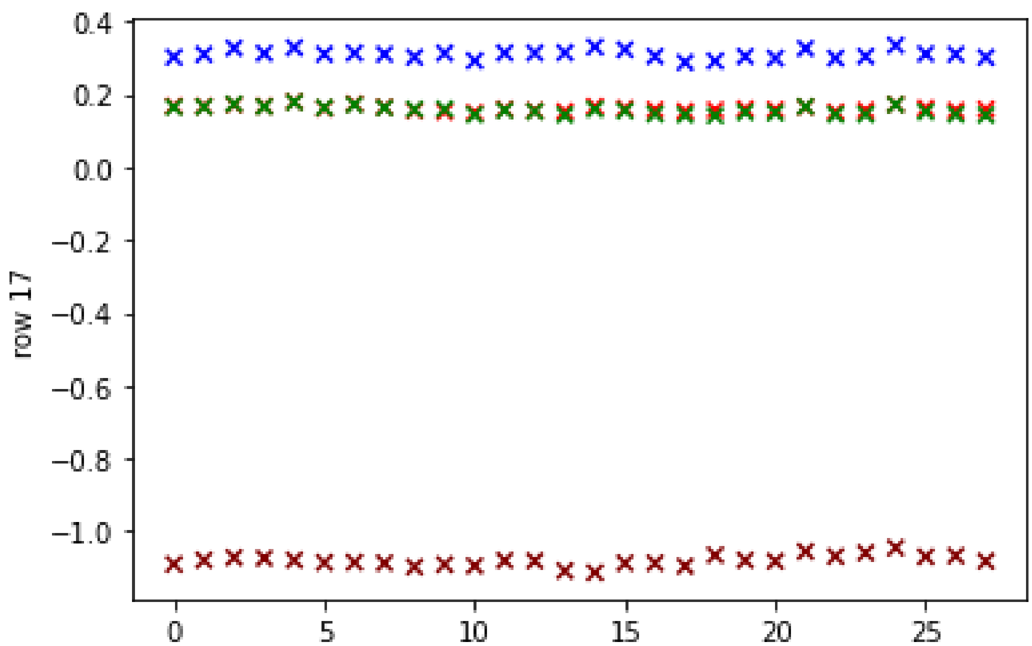


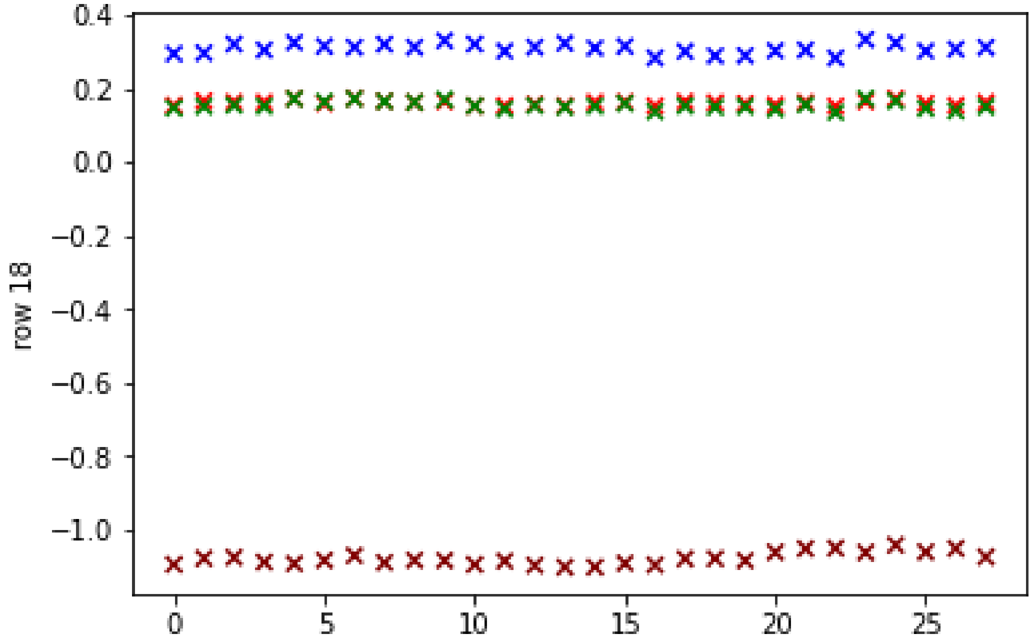
row 14
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25



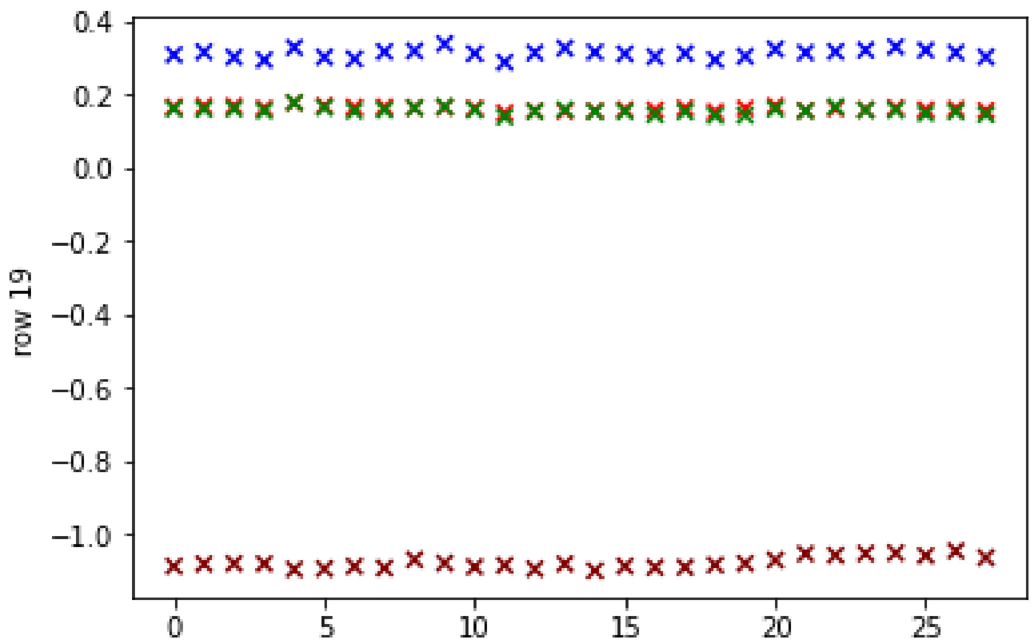
row 15
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25


row 16
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

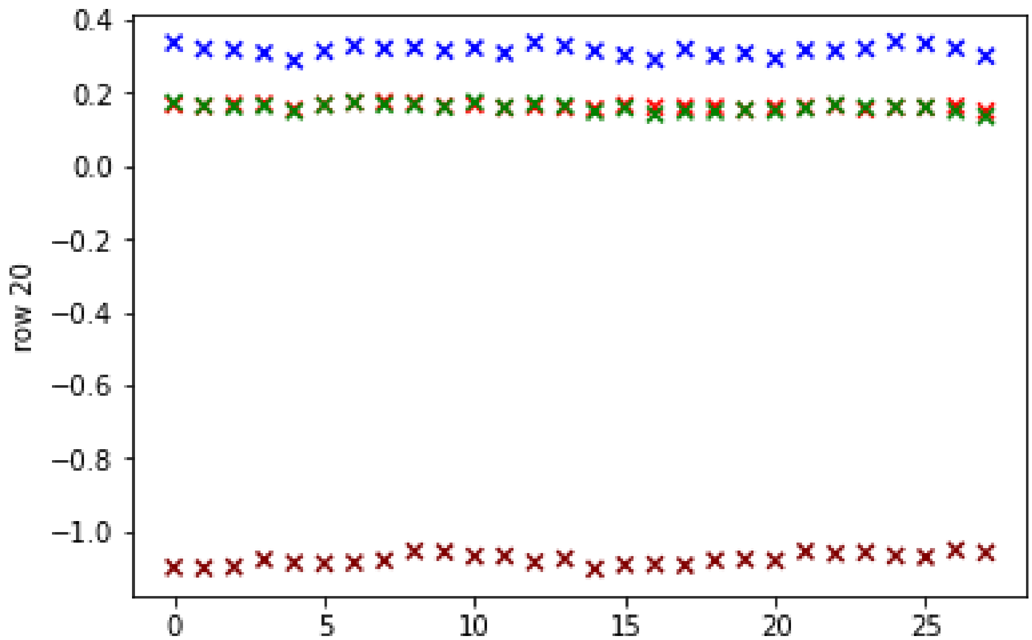

row 17
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25

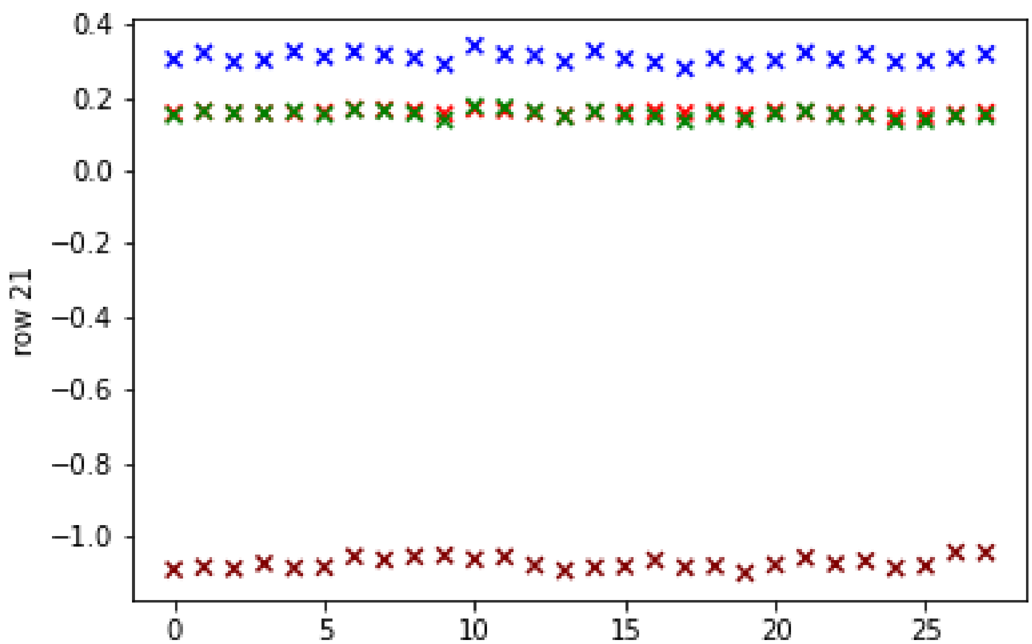


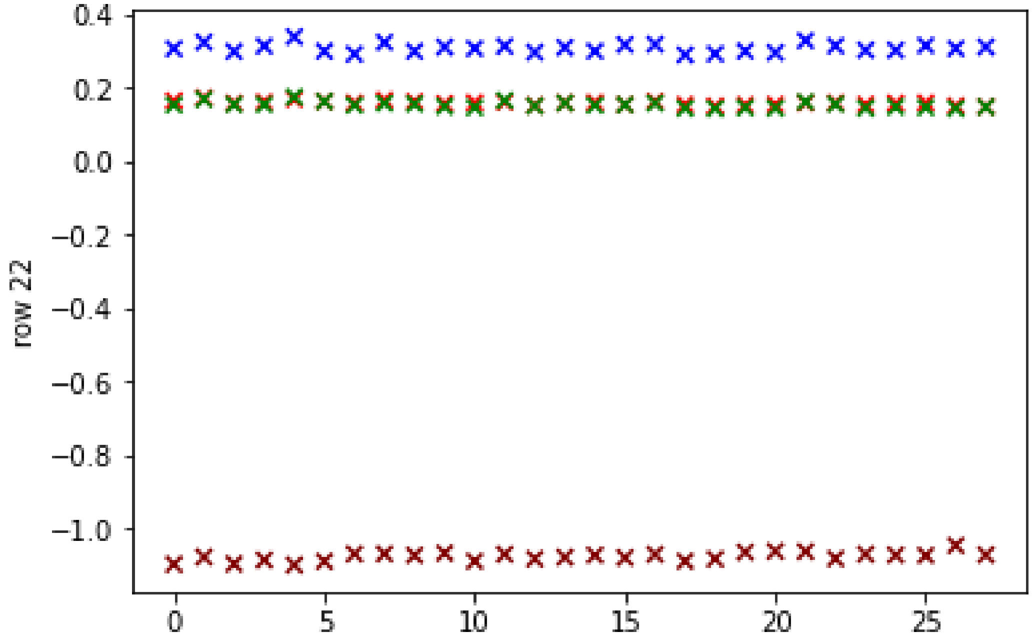
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
row 18
0
5
10
15
20
25



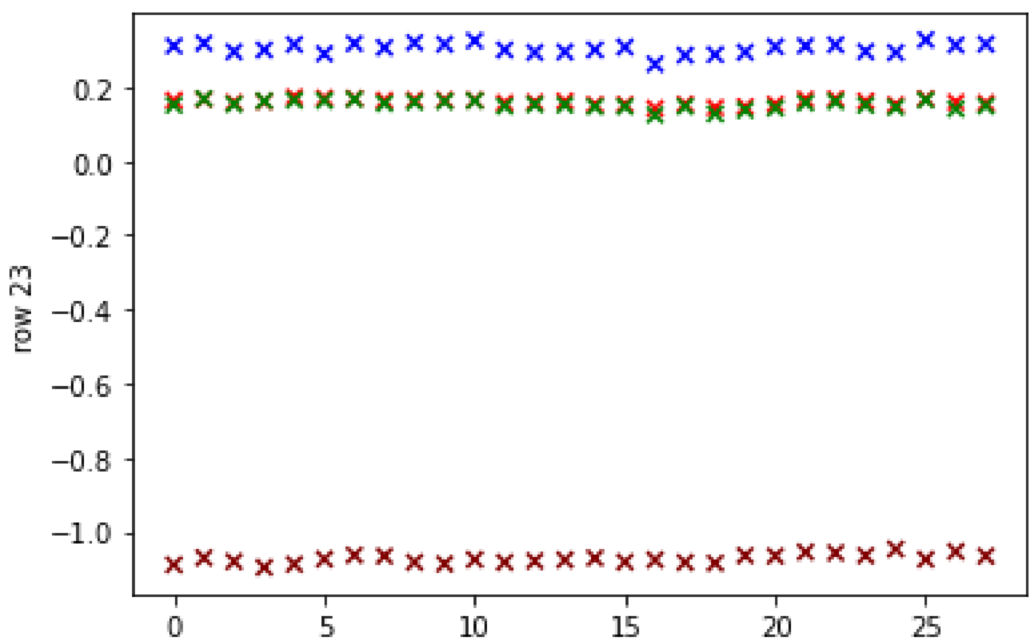
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
row 19
0
5
10
15
20
25


0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
row 20
0
5
10
15
20
25

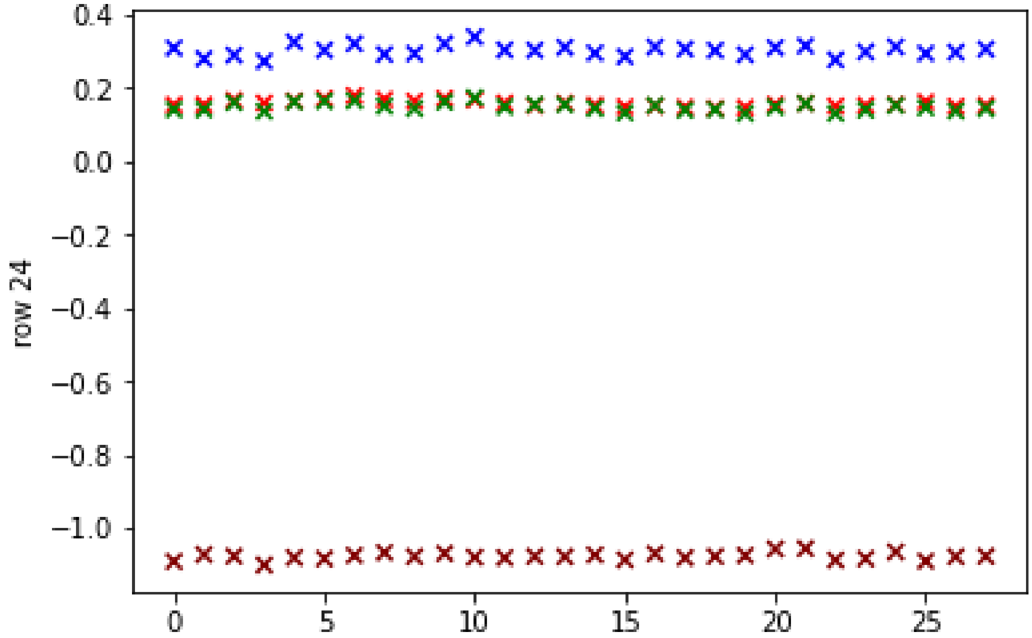

0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
row 21
0
5
10
15
20
25

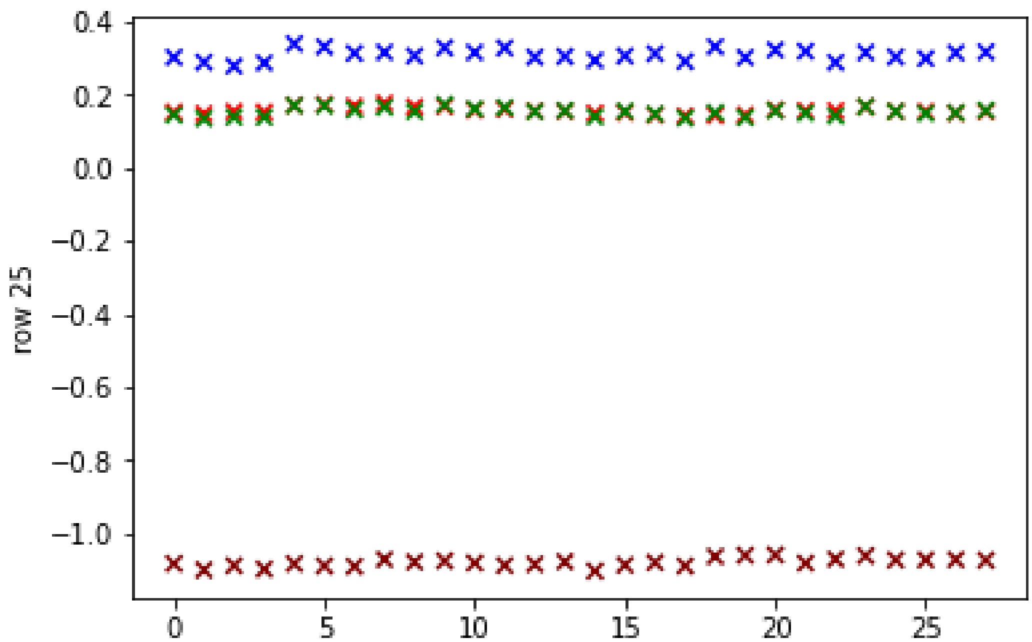


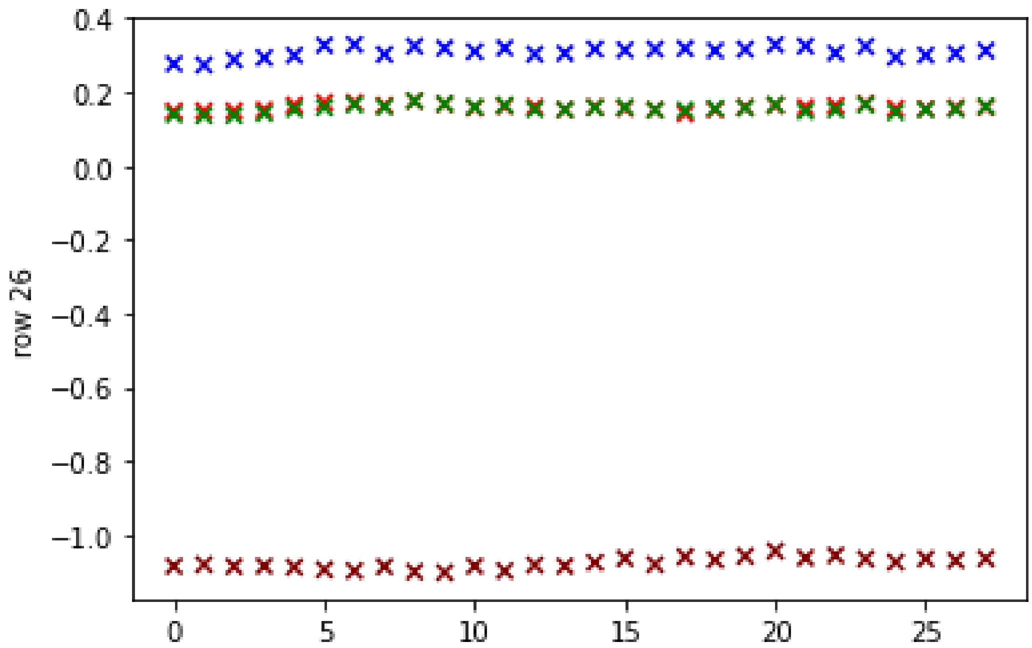
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
row 22
0
5
10
15
20
25

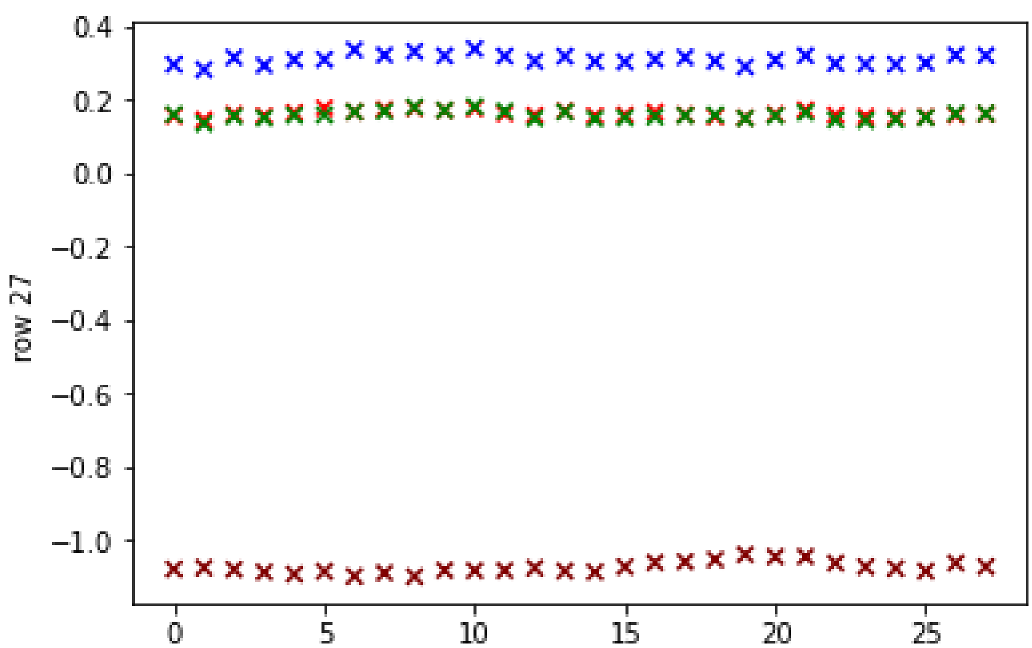


0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
row 23
0
5
10
15
20
25


row 24
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25


row 25
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
5
10
15
20
25





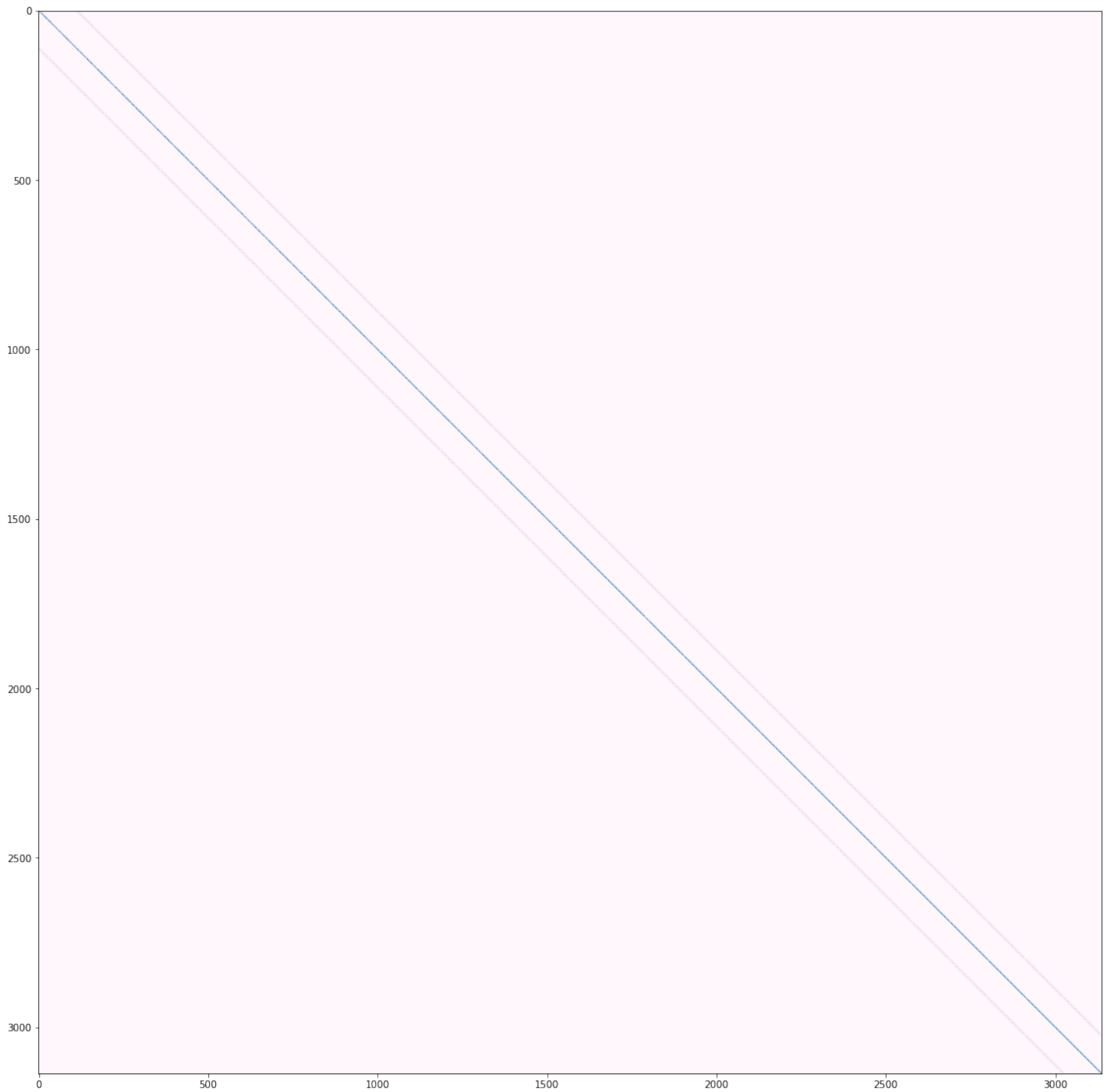
We can see from these scatter plots that: * the red, green and blue channel are consistently approximately semetric ($\in [-0.5, 0.5]$) (especially the red and green colors). Thus they are non-long tail distributions. * the near infrared (NIR) channel are consistently highly skewed negative ($\in [-\infty, -1.0]$). Thus this is a long tail distribution.

Therefore, if there were missing values for us to replace, we would use the mean for the red, green and blue channels, and the median for the NIR channel.

Next, we see the correlations to find any features that we can remove.

[11]:
```
# calculate the square correlations
X_cor2 = X_train.cov()**2
X_var = X_std**2
X_varprod = pd.DataFrame([(v0*v1) for v1 in X_var] for v0 in X_var)
# normalize X_cor2 using the product of the corresponding variances
X_cor2 = (X_cor2 / X_varprod)
```

[12]:
```
# find strong correlations
is_strong_corr = (X_cor2 >= R2_TOLERANCE)
# graph results
plt.figure(figsize=(20, 20))
plt.imshow(is_strong_corr, cmap='PuBu')
plt.show()
```



[13]:
```
# create a table of the strong correlations per feature
i_strong_corr_per_feat = pd.DataFrame(is_strong_corr[k].to_numpy().nonzero() for␣
 ↪k in range(N_FEATS))
i_strong_corr_per_feat
```

[13]:

	0
0	[0, 1, 4, 112, 116]
1	[0, 1, 2]
2	[1, 2]
3	[3, 115]
4	[0, 4, 5, 8, 112, 116, 120]
...	...
3131	[3019, 3131]

```
3132  [3016, 3020, 3128, 3132, 3133, 3134]
3133                      [3132, 3133, 3134]
3134                      [3132, 3133, 3134]
3135                            [3023, 3135]

[3136 rows x 1 columns]
```

Going by domain knowledge of the features, we know that they are all linearly independent.

There seem to be strong correlations at about 112 flattened pixels from each pixel in either direction of most pixels corresponding to $112\,\text{px} \times \frac{1\,\text{row}}{28\,\text{px}} = 4\,\text{rows}$ up or down respectively. However, it was not consistent enough to rely on for reducing dimensionality.

Finally, the labels are one-hot encoded, representing

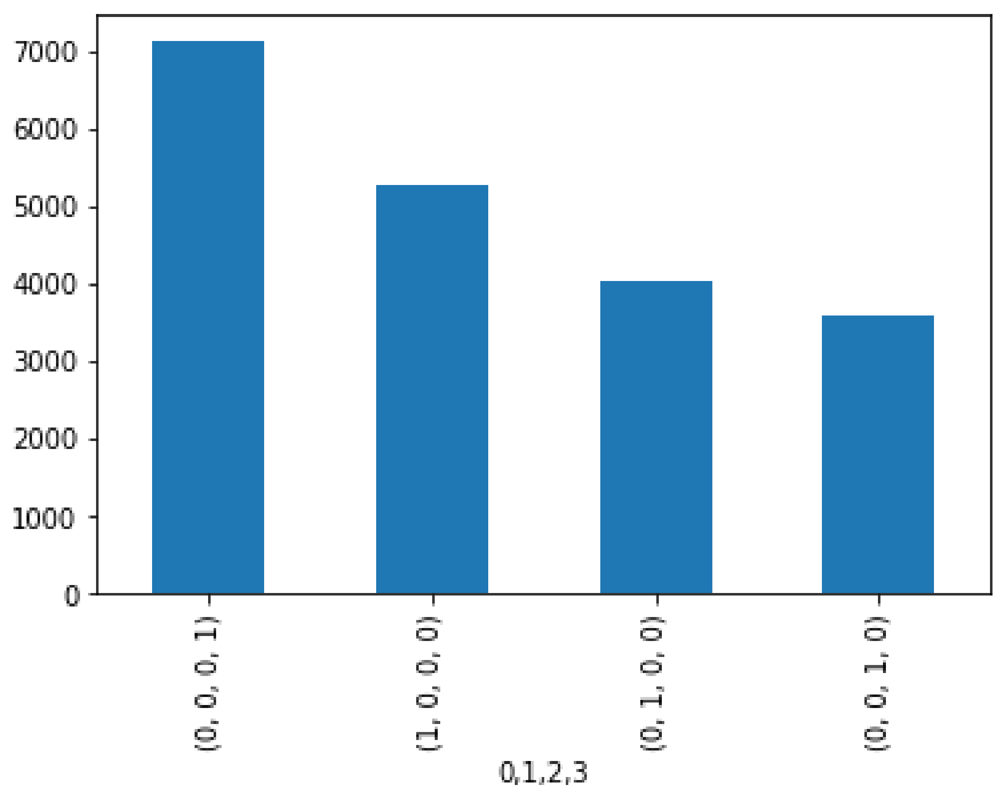
terrain type	1000	0100	0010	0001
barren_land	1	0	0	0
trees	0	1	0	0
grassland	0	0	1	0
none	0	0	0	1

Now to see how balanced the target data is, we use a bar plot.

[14]:
```
# find the label and its label counts
y_train_counts = y_train.value_counts()
r_diff = 2*abs(y_train_counts[0] - y_train_counts[1])/(y_train_counts[0] +
 ↪y_train_counts[1])

# print and bar plot of the label
print(r'===y_train===')
print(y_train)
print()
print(r'===value counts===')
print(y_train_counts)
print()
print(r"rate of difference = {}".format(r_diff))
y_train_counts.plot(kind=r'bar')
plt.show()
```

```
===y_train===
       0  1  2  3
0      0  1  0  0
1      1  0  0  0
2      0  0  0  1
3      1  0  0  0
4      1  0  0  0
...   .. .. .. ..
```

```
19995   0   0   1   0
19996   1   0   0   0
19997   1   0   0   0
19998   0   1   0   0
19999   0   0   1   0

[20000 rows x 4 columns]

===value counts===
0  1  2  3
0  0  0  1     7120
1  0  0  0     5276
0  1  0  0     4032
   0  1  0     3572
dtype: int64

rate of difference = 1  2  3
0  0  0   NaN
      1   NaN
   1  0   NaN
1  0  0   NaN
dtype: float64
```



We see that these total

encoding	multiplicity
1000	5276
0100	4032
0010	3572
0001	7120
$\sum$	20000.

Now consider that we have a vector $\vec{v}$ of 4 Boolean integers. Well in our case,

$$\mathbf{1}_4^\intercal \mathbf{v} = 1.$$

Thus for any order, it is the case that $v_4 = 1 - (v_1 + v_2 + v_3)$. Therefore, we can remove one of these label columns and have a linearly independent set of columns. I choose to remove `0001` because it is the last column.

[15]:
```
# columns to keep for linear independence
y_lind_cols = y_train.columns[0:-1]
# remove the last column of the the labels
y_train_lind = y_train[y_lind_cols]
# confirm the removal
print(y_train_lind.shape)
```

```
(20000, 3)
```

1.2 Split the data and normalize the features of examples according to training data

[16]:
```
# read in the testing data (stored in a separate set of files)
X_test = pd.read_csv(X_TEST_FILENAME, header=None, index_col=None)
y_test = pd.read_csv(Y_TEST_FILENAME, header=None, index_col=None)

# remove the last label column for test data for linear independence too
y_test_lind = y_test[y_lind_cols]

# data shape constants
(N_TEST_XAMPS, N_TEST_FEATS) = X_test.shape
(_, N_TEST_LBLS) = y_test_lind.shape

# confirm the shapes
print("# test examples:\t{}".format(N_TEST_XAMPS))
print("# test features:\t{}".format(N_TEST_FEATS))
print("# test labels:\t{}".format(N_TEST_LBLS))
```

```
# test examples:        20000
# test features:        3136
```

```
# test labels:  3
```

Next we normalize. However, first we check if there are outliers (1.5 IQRs outside of the range $[Q_1, Q_3]$) on **x**'s data frame.

```
[17]: def find_outliers(X):
          r'''
           Finds the outliers in data frame X.
           @param X : pd.DataFrame = to search for outliers
           @return (inlier_min, inlier_max, outliers) = lower and upper
           bounds of outliers, and the outliers in X
           '''
          # find the interquartile range
          q1, q3 = (X.quantile(q=q) for q in np.array([1, 3])*0.25)
          iqr = q3 - q1
          # calculate limits of the outlier
          inlier_min = (q1 - 1.5*iqr)
          inlier_max = (q3 + 1.5*iqr)
          # find any values out of range
          outliers = ((X < inlier_min) | (X > inlier_max))
          # return the result
          return (inlier_min, inlier_max, outliers)
      # def find_outliers(X)
```

```
[18]: # check if outliers
      (inlier_min, inlier_max, outliers) = find_outliers(X_train)
      # print the limits
      print(r'===outlier lower limit===')
      print(inlier_min)
      print()
      print(r'===outlier upper limit===')
      print(inlier_max)
      print()
      # print whether any outliers
      print(r'===has outliers?===')
      print(outliers.any())
```

```
===outlier lower limit===
0        6.5
1       25.5
2       24.5
3       72.5
4        6.5
        ...
3131    72.5
3132     9.0
3133    28.0
3134    23.0
3135    72.5
```

```
Length: 3136, dtype: float64

===outlier upper limit===
0       250.5
1       221.5
2       196.5
3       252.5
4       250.5
        ...
3131    252.5
3132    249.0
3133    220.0
3134    199.0
3135    252.5
Length: 3136, dtype: float64

===has outliers?===
0       True
1       True
2       True
3       True
4       True
        ...
3131    True
3132    True
3133    True
3134    True
3135    True
Length: 3136, dtype: bool
```

Since we found at least 1 column with outliers, we will use standardization (Z-score normalization), rather than min-max normalization.

[19]:
```
# standardization
means = X_train.mean(axis=0)
stds = X_train.std(axis=0)

# find the z-scores, replacing X_train, X_test
X_train, X_test = ((X - means)/stds for X in (X_train, X_test))

# for the mean and standard deviation
for name, stat in (('mean', X_train.mean()), ('standard devaition', X_train.
 ↪std())):
    (_, _, stat_outliers) = find_outliers(stat)
    print(r"===standardized X_train {} statistics===".format(name))
    print(stat.describe())
    print()
# next name, stat
```

```
===standardized X_train mean statistics===
count    3.136000e+03
mean     5.560037e-19
std      1.500497e-16
min     -3.765876e-16
25%     -1.129763e-16
50%     -1.421085e-18
75%      1.108447e-16
max      3.765876e-16
dtype: float64

===standardized X_train standard devaition statistics===
count    3.136000e+03
mean     1.000000e+00
std      9.023635e-17
min      1.000000e+00
25%      1.000000e+00
50%      1.000000e+00
75%      1.000000e+00
max      1.000000e+00
dtype: float64
```

From the statistics of the mean and standard deviation, we verify that all features' means $\bar{x} \approx 0$ and $S_X \approx 1$ as expected.

1.3 Training the model

Now let's attempt the linear model.

[20]:
```
linear = LinearRegression()
linear.fit(X_train, y_train_lind)
```

[20]:
```
LinearRegression()
```

Let's inspect the weights (with bias first) to see which features are as important in determining the model.

[21]:
```
print("bias:\t{}".format(linear.intercept_))
print("weights:\t{}".format(linear.coef_))
print("# weights:\t{}".format(linear.coef_.shape))
```

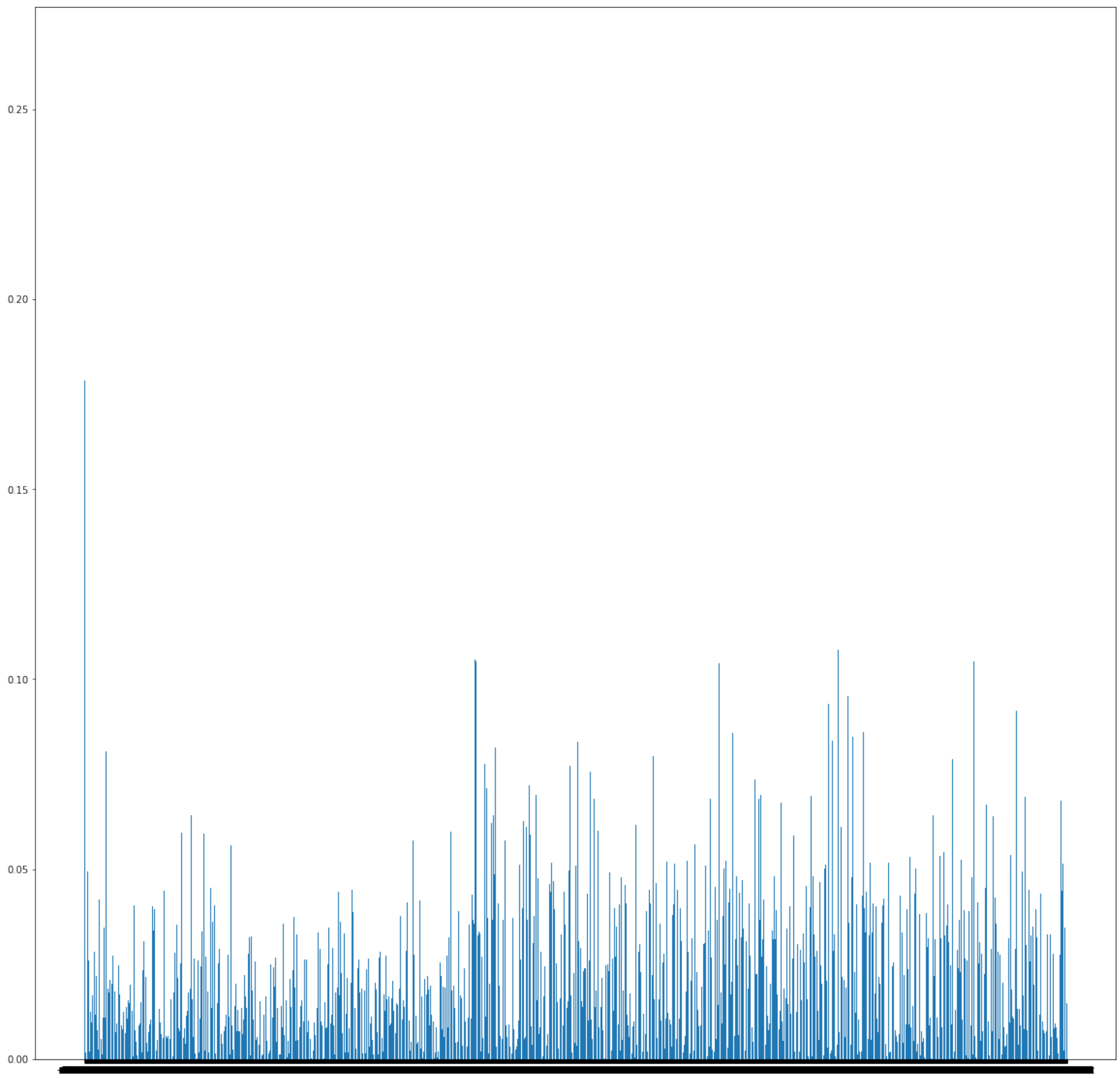
```
bias:   [0.2638 0.2016 0.1786]
weights:        [[ 0.00349978  0.03350413 -0.02830229 ... -0.00076971
-0.01991985
  -0.01042235]
 [-0.03050444  0.00311614  0.0097701  ...  0.040247    0.00532699
  -0.00593741]
 [-0.0461447   0.02486673  0.00224196 ... -0.0174027   0.01254749
```

```
 -0.00033398]]
# weights:      (3, 3136)
```

[22]:
```python
# get the learned weights
W = np.insert(linear.coef_, 0, linear.intercept_)
# flatten weights to a vector
weights = W.flatten()
# get the weight magnitude
w_mag = abs(W).flatten()
w_mag
```

[22]:
```
array([0.2638    , 0.2016    , 0.1786    , ..., 0.0174027 , 0.01254749,
       0.00033398])
```

[23]:
```python
# plot the bar graph
# the bar graph from the data frame would be too compact, so we use plt.bar.
# 3 significant figures in scientific notation should show all
# (3 + 3196*3) = 9411 independent features.
plt.figure(figsize=(20, 20))
plt.bar(["{:+.3e}".format(weight) for weight in weights], w_mag)
plt.show()
```



Find the measures of error agains the training data.

[24]:
```
# check the model fit to training data
y_train_pred = linear.predict(X_train)

# mean errors
mae_train = mean_absolute_error(y_train_pred, y_train_lind)
mse_train = mean_squared_error(y_train_pred, y_train_lind)
rmse_train = np.sqrt(mse_train)

print()
print(r'===fit to the training set===')
print('MAE\t', mae_train)
print('MSE\t', mse_train)
```

```
print('RMSE\t', rmse_train)
```

```
===fit to the training set===
MAE      0.2269431044423952
MSE      0.08822828611344057
RMSE     0.2970324664299183
```

1.4 Evaluating the model

Find the measures of error agains the testing data.

[25]:
```
# check the model fit to testing data
# use a data frame wrapper for y_test_pred
y_test_pred = pd.DataFrame(linear.predict(X_test))

# mean errors
mae_test = mean_absolute_error(y_test_pred, y_test_lind)
mse_test = mean_squared_error(y_test_pred, y_test_lind)
rmse_test = np.sqrt(mse_test)

print()
print(r'===fit to the testing set===')
print('MAE\t', mae_test)
print('MSE\t', mse_test)
print('RMSE\t', rmse_test)
```

```
===fit to the testing set===
MAE      0.27122718739782964
MSE      0.12413254391711742
RMSE     0.35232448668396216
```

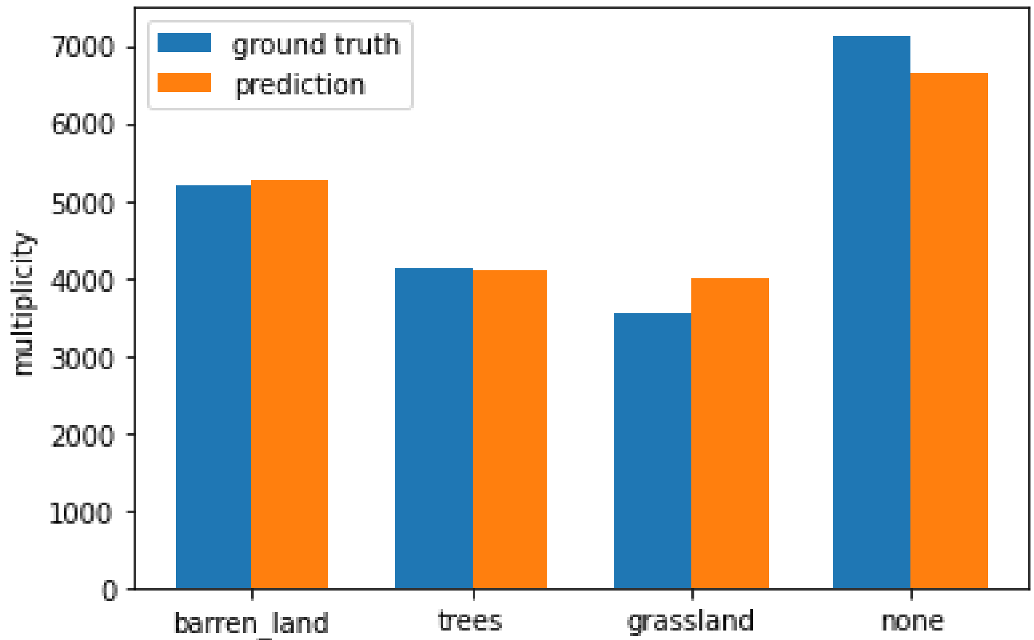
Let us analyze visually. First, let's reconstruct the linear dependent column of the labels.

[26]:
```
# add the dependent column to the prediction
# v4 = 1 - (v1 + v2 + v3)
y_test_pred_dep = pd.concat((y_test_pred, (1 - y_test_pred.sum(axis=1))),␣
↪axis=1, ignore_index=True)
```

[27]:
```
labels = [r'barren_land', r'trees', r'grassland', r'none']
x = np.arange(len(labels))  # the label locations
width = 0.35  # the width of the bars

# print label counts
print(r"=== y value counts===")
print(y_test.sum(axis=0))
print()
print(r"=== y' value row sums===")
```

```
print(y_test_pred_dep.sum(axis=0))
print()

fig, ax = plt.subplots()
rects1 = ax.bar(x - width/2, y_test.sum(axis=0), width, label='ground truth')
rects2 = ax.bar(x + width/2, y_test_pred_dep.sum(axis=0), width,␣
 ↪label='prediction')

ax.set_ylabel('multiplicity')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend()

plt.show()
```

```
=== y value counts===
0    5194
1    4117
2    3554
3    7135
dtype: int64

=== y' value row sums===
0    5268.622657
1    4080.350228
2    4010.181841
3    6640.845274
dtype: float64
```



By visual inspection, the model produces a close fit. However, let us normalize the mean errors.

First, let's check for overfit.

[29]:
```
print("MAE test:train ratio:\t{}".format(mae_test/mae_train))
print("RMSE test:train ratio:\t{}".format(rmse_test/rmse_train))
```

```
MAE train:test ratio:	0.8367269764498956
RMSE train:test ratio:	0.8430650654615408
```

We find that the test-train error ratios *MAE* and *RMSE* are each greater than approximated 1. This means that the testing errors are strictly greater than the training errors. Thus, we have a case of overfit. We will later solve that through ridge regularization.

Next, let's compare the *MAE* and *RMSE* of the test data with the mean absolute deviation around the median and standard deviation respectively of the training data.

[103]:
```
# find the norms
L1_y_train = (abs(y_train_lind).dot(np.array([1, 1, 1])))
L2_y_train = (((y_train_lind**2).dot(np.array([1, 1, 1])))**(1.0/2.0))

# for MAE, we find the ratio to the mean absolute deviation around the median of␣
↪the L1-norm
q2 = L1_y_train.quantile(q=0.5)
mad = np.linalg.norm((L1_y_train - q2), ord=1)/N_XAMPS
mae_iqr_test = mae_test/mad
```

```
# for RMSE, we find the ratio by the standard deviation of the L2-norm
std = L2_y_train.std()
rmse_std_test = rmse_test/std

# display the results
print("MAD:\t{}".format(mad))
print("SD:\t{}".format(std))
print()

print("MAE test:MAD train:\t{}".format(mae_iqr_test))
print("RMSE test:SD train:\t{}".format(rmse_std_test))
```

```
MAD:	0.356
SD:	0.4788271752659708

MAE test:MAD train:	0.7618741219040158
RMSE test:SD train:	0.7358072074507027
```

Now since the ratios are both < 1, this means that the mean absolute error is much less than one interquartile range, and the root mean square error is less than one standard deviation.

This means that the model has a strong predictive power. However, because both are close to 1, it is a moderately strong predictive power.

1.5 Ridge regularization to reduce overfitting

Finally, let's apply ridge regularization to see if we can reduce overfitting.

Let's establish the baseline for the linear regression.

Here, I attempt to calculate accuracy, recall, precision and F_1 score.

[54]:
```
# find the performance of the standard linear regression
q2_train = np.quantile(y_train_lind, q=0.5, axis=0)

# for accuracy
N_PREDICTIONS, N_LABELS = y_test_pred.shape
correct = (abs(y_test_lind - y_test_pred) <= ACCURACY_TOLERANCE**(1.0/
 ↪float(N_LABELS))).all(axis=1)
n_correct = correct.sum()

# for recall, precision
ps = (y_test_pred > q2_train).all(axis=1)
n_tp = (correct & ps).sum()
n_tn = (n_correct - n_tp)
n_fp = (ps.sum() - n_tp)
n_fn = (N_PREDICTIONS - n_correct - n_fp)

accuracy = n_correct/N_PREDICTIONS
recall = n_tp/(n_tp + n_fn)
```

```
precision = n_tp/(n_tp + n_fp)
f1 = mean((recall**-1, precision**-1))**-1

print('threshold for positive is >= ', q2_train)
print()
print('accuracy \t', accuracy)
print('recall    \t', recall)
print('precision\t', precision)
print('f1 score \t', f1)
```

```
threshold for positive is >=  [0. 0. 0.]

accuracy         0.3844
recall           0.32501155802126674
precision        0.30288668677294267
f1 score         0.3135593220338983
```

Here, we use a tolerance of 0.5 for a positive label. Then we use macro-averaging because the labels are independent.

[81]:
```
LABEL_TOLERANCE = 0.5                                        # minimum for a set␣
 ↪bit in a label
# calculate the scores through macro-averaging
accuracy = accuracy_score(y_test_lind, y_test_pred >= LABEL_TOLERANCE)
recall = recall_score(y_test_lind, y_test_pred >= LABEL_TOLERANCE,␣
 ↪average='macro')
precision = precision_score(y_test_lind, y_test_pred >= LABEL_TOLERANCE,␣
 ↪average='macro')
f1 = f1_score(y_test_lind, y_test_pred >= LABEL_TOLERANCE, average='macro')
# display the results
print('accuracy \t', accuracy)
print('recall    \t', recall)
print('precision\t', precision)
print('f1 score \t', f1)
```

```
accuracy         0.58665
recall           0.4493992231017458
precision        0.6065020921872838
f1 score         0.5036355376125148
```

Next, let's compare the accuracy, recall, precision and F1 scores for the range $[-10\,\mathrm{dB}..11\,\mathrm{dB}]$.

[90]:
```
lambda_db = np.array(range(-10, 11)) # different regularization hyper parameters␣
 ↪in dB
lambda_lin = 10**(lambda_db/20)       # convert to linear scale

# lists to plot
recalls = [ recall ]
precisions = [ precision ]
```

```
# list for maximum ('best') F1 score
f1_scores = [ f1 ]

for i_lambda, lambda_val in enumerate(lambda_lin):
    # test the model fit to testing data
    ridge = Ridge(alpha=lambda_val)
    ridge.fit(X_train, y_train_lind)
    y_test_pred = ridge.predict(X_test)

    # calculate the scores through macro-averaging
    accuracy = accuracy_score(y_test_lind, y_test_pred >= LABEL_TOLERANCE)
    recall = recall_score(y_test_lind, y_test_pred >= LABEL_TOLERANCE,␣
↪average='macro')
    precision = precision_score(y_test_lind, y_test_pred >= LABEL_TOLERANCE,␣
↪average='macro')
    f1 = f1_score(y_test_lind, y_test_pred >= LABEL_TOLERANCE, average='macro')

    # append to the lists
    recalls.append(recall)
    precisions.append(precision)
    f1_scores.append(f1)

    # display result for this lambda
    print(fr'===regularization parameters = {lambda_db[i_lambda]} dB =␣
↪{lambda_val}===')
    print('accuracy \t', accuracy)
    print('recall   \t', recall)
    print('precision\t', precision)
    print('f1 score \t', f1)
    print()
# next lambda_val
```

```
===regularization parameters = -10 dB = 0.31622776601683794===
accuracy         0.58715
recall           0.44922446696412854
precision        0.6075849506159723
f1 score         0.5037411510816644

===regularization parameters = -9 dB = 0.35481338923357547===
accuracy         0.5872
recall           0.4492886435781704
precision        0.6077762406861051
f1 score         0.5038066425016782

===regularization parameters = -8 dB = 0.3981071705534972===
accuracy         0.5873
recall           0.4494337852962524
```

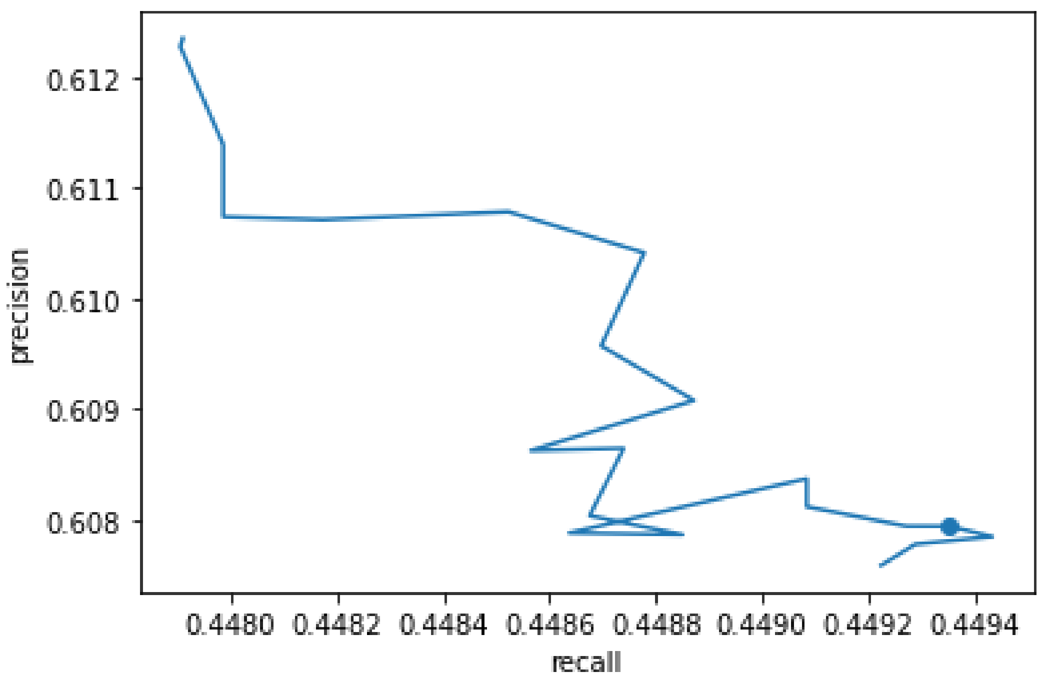
```
precision       0.607844995207374
f1 score        0.5039229368765102

===regularization parameters = -7 dB = 0.44668359215096315===
accuracy        0.58735
recall          0.4493528201922122
precision       0.6079392009526936
f1 score        0.5038767122931914

===regularization parameters = -6 dB = 0.5011872336272722===
accuracy        0.58735
recall          0.449271855088172
precision       0.6079398509472012
f1 score        0.5038157451076203

===regularization parameters = -5 dB = 0.5623413251903491===
accuracy        0.58755
recall          0.4490842730210176
precision       0.6081144087457719
f1 score        0.5036484574642933

===regularization parameters = -4 dB = 0.6309573444801932===
accuracy        0.5876
recall          0.4490842730210176
precision       0.6083680618451495
f1 score        0.503697137969315

===regularization parameters = -3 dB = 0.7079457843841379===
accuracy        0.58735
recall          0.4486370071517444
precision       0.6078782887164116
f1 score        0.5031616356837981

===regularization parameters = -2 dB = 0.7943282347242815===
accuracy        0.58745
recall          0.44885028804432964
precision       0.6078626211036924
f1 score        0.5032986730138832

===regularization parameters = -1 dB = 0.8912509381337456===
accuracy        0.58755
recall          0.4486755319067122
precision       0.60803368099758
f1 score        0.5031846728113596

===regularization parameters = 0 dB = 1.0===
accuracy        0.58775
recall          0.44873970852075407
```

```
precision        0.6086427593859304
f1 score         0.5033432141352383

===regularization parameters = 1 dB = 1.1220184543019633===
accuracy         0.58775
recall           0.4485649523831367
precision        0.6086225636833118
f1 score         0.5032103547844932

===regularization parameters = 2 dB = 1.2589254117941673===
accuracy         0.588
recall           0.448872024309299
precision        0.6090748139440184
f1 score         0.5035466488795411

===regularization parameters = 3 dB = 1.4125375446227544===
accuracy         0.5881
recall           0.44869726817168165
precision        0.6095694906461742
f1 score         0.5035110328600362

===regularization parameters = 4 dB = 1.5848931924611136===
accuracy         0.5885
recall           0.4487782332757218
precision        0.6104109099997941
f1 score         0.5037766968830625

===regularization parameters = 5 dB = 1.7782794100389228===
accuracy         0.5885
recall           0.44852251203406435
precision        0.6107817976750681
f1 score         0.5036020810947259

===regularization parameters = 6 dB = 1.9952623149688795===
accuracy         0.58845
recall           0.44817299975882957
precision        0.6107110409285051
f1 score         0.5032615372821362

===regularization parameters = 7 dB = 2.2387211385683394===
accuracy         0.58855
recall           0.44798541769167527
precision        0.6107363934997151
f1 score         0.5030861896774851

===regularization parameters = 8 dB = 2.51188643150958===
accuracy         0.58875
recall           0.44798541769167527
```

```
precision        0.6113871721572975
f1 score         0.5032827113710475

===regularization parameters = 9 dB = 2.8183829312644537===
accuracy         0.5891
recall           0.44790445258763506
precision        0.6122752617389601
f1 score         0.5034322630651811

===regularization parameters = 10 dB = 3.1622776601683795===
accuracy         0.58925
recall           0.4479094003626063
precision        0.6123493532783794
f1 score         0.5033598847989803
```

We can compare recall and precision by plotting them as follows where the disc represents the model with no regularization.

[91]:
```
plt.plot(recalls[1:], precisions[1:])
plt.scatter(recalls[0], precisions[0])
plt.xlabel('recall')
plt.ylabel('precision')
plt.show()
```



Finally, the maximum F1 score gives us.

[92]:

```
i_model, F1_score = max(enumerate(f1_scores), key=lambda e: e[1])
if (i_model > 0):
    model_name = r"regularization = {} dB = {}".format(lambda_db[i_model - 1],␣
 ↪lambda_lin[i_model - 1])
else:
    model_name = r'no regularization'
# end if (i_model > 0)

print(r"Best model is '{}' with F1 score = {}.".format(model_name, F1_score))
```

```
Best model is 'regularization = -8 dB = 0.3981071705534972' with F1 score =
0.5039229368765102.
```

Part III

Sampling script

```python
r'''
 Canonical : https://github.com/lduran2/CIS3715_DataScience_2022/blob/main/final/
 ↪sample_dataset.py
 Samples a set of X, y from examples of dataset files.
 By        : Leomar Durán <https://github.com/lduran2>
 For       : CIS 3715/Principles of Data Science

 CHANGELOG :
    v1.2.7 - 2022-04-19t01:37Q
        `mainarg` for each arg, delete `df` after use

    v1.2.6 - 2022-04-12t17:01Q
        `sampleCsvFile` returns tuple of lists

    v1.2.5 - 2022-04-12t16:10Q
        lists->tuples where possible

    v1.2.4 - 2022-04-11t01:30Q
        fixed df headers written

    v1.2.3 - 2022-04-10t23:45Q
        stacks->iterators, implemented associated (X,y)

    v1.2.2 - 2022-04-10t22:16Q
        fixed df index written, IDed feat/tgt files

    v1.2.1 - 2022-04-10t21:35Q
        modularized `sample_dataset.py`

    v1.2.0 - 2022-04-10t17:22Q
        sampling is done

    v1.1.0 - 2022-04-10t14:56Q
        counting row of each file

    v1.0.0 - 2022-04-07t07:34Q
        gets the options for sampling
 '''

import sys                          # gets command line arguments
from getopt import getopt           # to handle command line options
import csv                          # for list read
import random                       # for sampling
import pandas as pd                 # for dataframe
from pathlib import Path            # for handling paths
from cqs_iter import CqsIter        # for iterator
```

```
# constants
K_EXAMPLES = 20000                  # default to sampling 20,000
OPTIONS = r'n:'                     # short commandline option names
LONG_OPTIONS = (r'nsamples=', )     # long commandline option names
SEED = 42                           # seed for sampling

def main(K_EXAMPLES=K_EXAMPLES):
    # apply the seed
    random.seed(SEED)

    # get the options from commandline
    option_to_values, argv = getopt(sys.argv[1:], OPTIONS, LONG_OPTIONS)

    # loop through options (key, value) pair
    for kopt, opt_value in option_to_values:
        # store if K_EXAMPLES key
        if (kopt in (r'-n', r'--nsamples')):
            K_EXAMPLES = int(opt_value)
        # end if (kopt in (r'-n', r'--nsamples'))
    # next kopt

    # report number of samples
    print(r"===sampling to {} examples===".format(K_EXAMPLES))

    # loop through filenames in argv
    for X_filename in argv:
        # sample X, y represented by that filename
        mainarg(X_filename, K_EXAMPLES)
    # next X_filename
# end def main(K_EXAMPLES=K_EXAMPLES)

def mainarg(X_inname, K_EXAMPLES=K_EXAMPLES):
    r'''
     Samples a set of X, y given a dataset's filename.
     @param X_inname : str = filename '*/X_*' representing a dataset
        X, y
     @param K_EXAMPLES : int = # indices to sample
     '''
    print()
    # set up file names
    # input paths X, y
    X_inpath = Path(X_inname)
    y_inpath = X_inpath.parent / r''.join((r'y', X_inpath.name[1:]))
    # output paths X, y
    samp_label = (r'_samp', str(K_EXAMPLES), X_inpath.suffix)
    X_outpath = X_inpath.parent / r''.join((X_inpath.stem, *samp_label))
    y_outpath = X_inpath.parent / r''.join((y_inpath.stem, *samp_label))
    # report reading/writing
```

```
    print(r'===reading from===')
    print(X_inpath)
    print(y_inpath)
    print(r'===writing to===')
    print(X_outpath)
    print(y_outpath)
    # perform conversion
    with open(X_inpath) as X_csvfile, open(y_inpath) as y_csvfile:
        # sample the csvfiles
        # y_csvfile 1st because it will be used to find
        #   dimensionality and is smaller
        csvfiles = (y_csvfile, X_csvfile)
        outlists = tuple([] for k in range(len(csvfiles)))
        sampleCsvFile(outlists, csvfiles, K_EXAMPLES)
        # loop through dataframes
        for (outlist, outpath) in zip(outlists, (y_outpath, X_outpath)):
            # report the writing
            print(r"writing to {}".format(outpath))
            # convert to dataframe
            df = pd.DataFrame(outlist)
            # write to each csv file
            df.to_csv(outpath, header=False, index=False)
            # delete the reference to dataframe
            del df
        # next (outlist, outpath)
    # end with X_csvfile, y_csvfile
# end def mainarg(X_inname, K_EXAMPLES)

def sampleCsvFile(dests, infiles, K_EXAMPLES):
    r'''
     Samples K_EXAMPLES from infiles into destination lists,
     representing dataframes.
     @param dests : tuplelike<listlike> = K-sampled examples from
        infiles to be output
     @param infiles : file = source files in CSV format
     @param K_EXAMPLES : int = # indices to sample
     @return the tuple of K-sampled lists, dest.
     '''
    # create readers for rows as lists
    csvins = tuple(csv.reader(infile) for infile in infiles)
    # get the dimensionality (from 1st file)
    (N_EXAMPLES, _) =  shape_2d(csvins[0])
    # rewind the file
    infiles[0].seek(0)
    # report number of examples
    print(r"===sampling from {} examples===".format(N_EXAMPLES))
    # sample the indices
    i_samples = sampleIndexes(N_EXAMPLES, K_EXAMPLES)
```

```
    # for each csv input file and destination listlike
    for i_csvin, (csvin, dest) in enumerate(zip(csvins, dests)):
        # report file being copied into memory
        print(r"{}. copying from {}".format(i_csvin, infiles[i_csvin]))
        # iterator on samples
        iit = CqsIter(iter(i_samples))
        # copy the csv file
        idxdCpy(dest, csvin, iit)
    # next csvin
    # return the dataframe
    return dests
# end def sampleCsvFile(dests, infiles, K_EXAMPLES)

def shape_2d(tuple2d):
    r'''
     Finds the shape of bidimensional tuple-like representing a
     dataframe.
     @param tuple2d : tuplelike<tuplelike> = to traverse
     @return (N_EXAMPLES, N_FEATURES) = # of examples, # features of
        the dataframe
     '''
    # loop through the rows
    for irow, row in enumerate(tuple2d):
        pass
    # next row
    # store the dimensionality of the file
    (N_EXAMPLES, N_FEATURES) = (irow, len(row))
    # return the dimensionality
    return (N_EXAMPLES, N_FEATURES)
# end def shape_2d(tuple2d)

def sampleIndexes(N, K):
    r'''
     Creates a list of the K sampled indices [0..N[.
     @param N : int = # elements in original iterable
     @param K : int = # indices to sample
     @return list of sampled indices.
     '''
    # indices to be sampled
    iis = range(N)
    # sample them
    i_samples = random.sample(iis, K)
    # sort them
    i_sorted = sorted(i_samples)
    # return sorted indexes
    return i_sorted
# end def sampleIndexes(N, K)
```

```python
def idxdCpy(dest, src, iit):
    r'''
     Copies elements whose index matches those from the index iterator
     in order from src in dest.
     @param dest : list<T> = destination list
     @param src : iterable = source iterable
     @param iit : CqsIter<int> = the index iterator
     @return the list copy, dest.
     '''
    for idx, el in enumerate(src):
        # return if no more samples
        if (not(iit)):
            return dest
        # end if (not(iit))
        consumeIdxd(dest.append, idx, el, iit)
    # next el
    return dest
# end def idxdCpy(dest, src, iit)

def consumeIdxd(consume, idx, el, iit):
    r'''
     Calls the consume callback on the given element if its index is
     next in the index iterator.
     @param consume : function<? super T> = to consume elements
     @param idx : int = index of current element
     @param el : T = element to consume
     @param iit : CqsIter<int> = the index iterator
     '''
    # consume only if element is next indexed
    if (idx != iit.get()):
        return
    # end if (idx != iit.get())
    # add this element
    consume(el)
    # pop the sample found
    iit.next()
# end def consumeIdxd(consume, idx, el, iit)

if __name__ == "__main__":
    main()
# if __name__ == "__main__"
```

Part IV

Command-query separated iterator

```
r'''
 Canonical : https://github.com/lduran2/CIS3715_DataScience_2022/blob/main/final/
 ↪cqs_iter.py
 A command-query separated version of iterator.
 By        : Leomar Durán <https://github.com/lduran2>
 For       : CIS 3715/Principles of Data Science

 @see https://wiki.kluv.in/a/CommandQuerySeparation (wiki.c2.com)

 CHANGELOG :
    v1.0.1 - 2022-04-11t00:41Q
        `isValid`->`__bool__`, implemented `__iter__`

    v1.0.1 - 2022-04-10t23:26Q
        handling no more elements in iterator

    v1.0.0 - 2022-04-10t22:59Q
        initial implementation
 '''

class CqsIter:
    # Represents no more elements in an iterator.
    __default = object()

    def __init__(self, it):
        r'''
         Adapts the iterator `it`.
         @param it : iter = the backing iterator
         '''
        self._it = it
        CqsIter.__next(self)
    # end def __init__(self, it)

    def next(self):
        r'''
         Moves the iterator to the next element.
         '''
        CqsIter.__next(self)
    # def next(self)

    def __bool__(self):
        r'''
         Returns true if the iterator is valid, i.e., if there is a current␣
 ↪element.
         @return true if the iterator is valid; false otherwise.
         '''
        return (self._curr is not CqsIter.__default)
```

```
    # def isValid(self)

    def get(self, default=__default):
        r'''
         Returns the current element whereto the iterator points.
         @param default : object = to return when no more elements
         @return the current element whereto the iterator points.
         '''
        # if no more elements
        if (self._curr is CqsIter.__default):
            # and no default, then raise exception
            if (default is CqsIter.__default):
                raise StopIteration
            else:
                # otherwise, return the given default
                return default
            # end if (default is CqsIter.__default) ||
        return self._curr
    # end def get(self, default=CqsIter.__default)

    def __iter__(self):
        r'''
         Returns the backing iterator.
         @return the backing iterator.
         '''
        return self._it

    def __next(self):
        r'''
         Private function implements #next.
         @see #next
         '''
        self._curr = next(self._it, CqsIter.__default)
    # end def __next(self)

# end class CqsIter
```