Kilter-Grade-Estimator

Pascal Lüscher

1 Data from the Website

The data from the kilterboard is not accessible to the public. Nonetheless, there exists an official application. Similar to the majority of Android applications, the kilterboard application employs a SQLite database. It is feasible to retrieve the routes from this database provided you possess a rooted mobile device.

Fortunately, an individual has already performed this task and established the website https://kilterboard.app. As a result, I crawled this website and gathered approximately 22,000 routes.

1.1 Data Analysis

One entry had the following structure


```
{
    "uuid": "f01419e1-2672-4593-96ca-62e3655abc46",
    "layout_id": 1,
    "layout_deviant_id": 9,
    "setter_id": 1078,
    "setter_username": "jwebxl",
    "name": "swooped",
    "description": "",
    "hsm": 1,
    "edge_left": 32,
    "edge_right": 88,
    "edge_bottom": 8,
    "edge_top": 152,
    "frames_count": 1,
    "frames_pace": 0,
    "is_draft": false,
    "is_listed": true,
    "created_at": "2018-12-06T21:15:01.127371+00:00",
```

```
    "climb_stats": [
        {
            "angle": 0,
            "fa_at": "2019-12-05 16:39:44",
            "climb_uuid": "F01419E12672459396CA62E3655ABC46",
            "fa_username": "sheylo",
            "quality_average": 2.68571,
            "ascensionist_count": 35,
            "difficulty_average": 14.8857
        },
        {
            "angle": 5,
            "fa_at": "2020-06-01 23:37:50",
            "climb_uuid": "F01419E12672459396CA62E3655ABC46",
            "fa_username": "djragan",
            "quality_average": 2,
            "ascensionist_count": 2,
            "difficulty_average": 12
        }
    ],
    "placements": [
        {
            "x": 20,
            "y": 0,
            "type": "FEET-ONLY",
            "ledPosition": 15
        },
        {
            "x": 16,
            "y": 5,
            "type": "START",
            "ledPosition": 244
        },
        {
            "x": 14,
            "y": 35,
            "type": "FINISH",
            "ledPosition": 221
        }
    ],
    "total_ascents": 4778
}
```


Listing 1: Example of one crawled Entry

as you see, there are several ratings per route, depending on the angle. There are also properties that should have no impact on the rating, like setter_username and the creation date.

1.1.1 Layouts

There is the layout_id and layout_deviant_id property. This specifies which Kilterboard is in use. A quick Histogram shows that the layout 1 is omnypresent and I can drop the others:

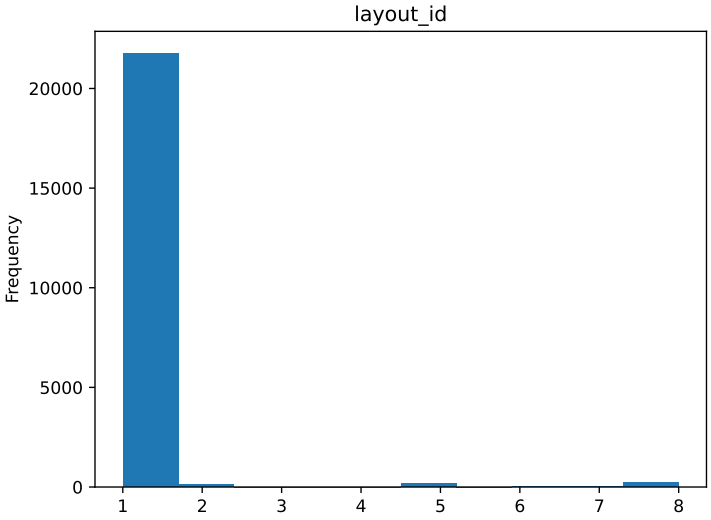


Figure 1: Histogram layout id

The same is true for the layout_deviant_id. The deviant 9 is omnypresent.

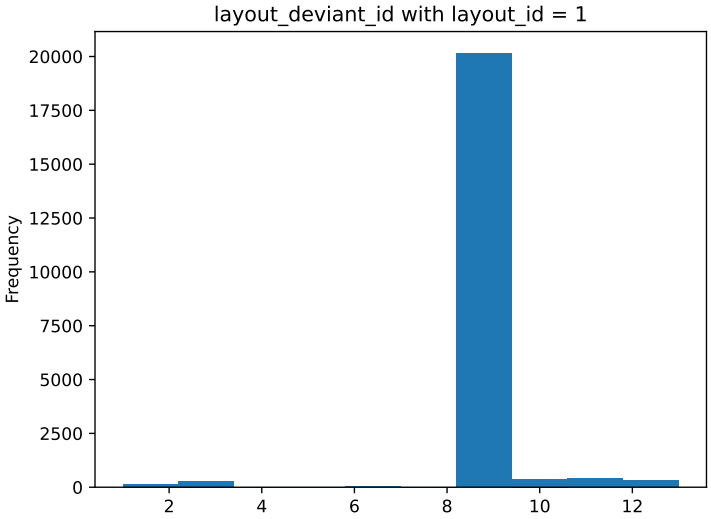


Figure 2: Histogram layout deviant id

for completness sake, the two histograms as text

```
layout_id
1     21778
2       148
5       196
6         2
7        12
8       207
```

```
layout_deviant_id
1.0          8
2.0        138
3.0        299
6.0         35
7.0         17
9.0      20151
10.0       382
11.0       410
12.0       307
13.0        11
```

1.1.2 Placements

The placements are the hold itself. The Kilterboard has 35×36 Grid. I first filtered out the outliers that had Y coordinates 36 and higher (it's 0-indexed), there were 61 cases. I then analyzed the XY-Combinations for the placement. My goal was to create a One-Hot Encoding for every hold on the grid. Every hold can be classified as foot, hand, starting or finishing hold. I then created a one hot encoding for all positions and types (Feet only, Middle, Start and Finish) FO_1_0, FO_1_1, 1_2, until S_35_35. Out of the 1225 ($35 \cdot 35$) positions, there were only 476 in use. When considering the possibilities with the type combination ($35 \cdot 35 \cdot 4 = 4900$) there were only 1637 combinations used. As a first step, I go with the relativly sparse matrix.

1.1.3 Target Variable

My goal is to predict the difficulty, there is a difficulty average per angle. In a first model, i will use this variable as is. Maybe I will convert these in categories.

1.2 DataConversion

The output model had the following structure:

```
    {
        "Uuid":"f01419e1-2672-4593-96ca-62e3655abc46",
        "Angle":0,
        "DifficultyAverage":14.8857,
        "FO_0_0":true,
        "F_0_0":false,
        "M_0_0":false,
        "S_0_0":false,
        "FO_0_1":true,
        "F_0_1":false,
        "M_0_1":false,
        ...
        "S_34_35":false
    }
```

Listing 2: Example of one crawled Entry

2 Data from the App

As mentioned previously, the kilterboard app is an app that uses a sqlite database. I went ahead and used an online service called genymotion to emulate a rooted android device. I installed the kilterboard app, connected adb (android debugging bridge) and pulled the database to my local cumputer. The total cost for this was 0.40 CHF, genymotion is billed by the minute (0.05 CHF per minute)

I found over 200'000 routes in the app database. The app database also gave me further insight in the structure.

The climbs are stored in a table called climbs. Then there is the climbs_stats table that stores the angels and average difficulties.

2.1 Placements

The placements arent stored in a nice way, there is a column in the climbs table called frames with the following structure:

"p1234r12p1238r13..."

as you can see, the string is divided in pXXXXrYY numbers. The p part is the placement id and the r part is the placement role (foot-only, middle, start, finish)

2.2 Difficulty

The difficulty conversion is stored in a table called difficulty_grades

Difficulty	Grade
10	4a/V0
11	4b/V0
12	4c/V0
13	5a/V1
14	5b/V1
15	5c/V2
16	6a/V3
17	6a+/V3
18	6b/V4
19	6b+/V4
20	6c/V5
21	6c+/V5
22	7a/V6
23	7a+/V7
24	7b/V8
25	7b+/V8
26	7c/V9
27	7c+/V10
28	8a/V11
29	8a+/V12
30	8b/V13
31	8b+/V14

with this information, I decided to go with a categorized approach. The difficulty gap between 5a and 5b is way less than the gap between 7a and 7b.

The distribution seems to be a bit lopsided, the majority of the data begins at 5c and stretches up until around 7b.

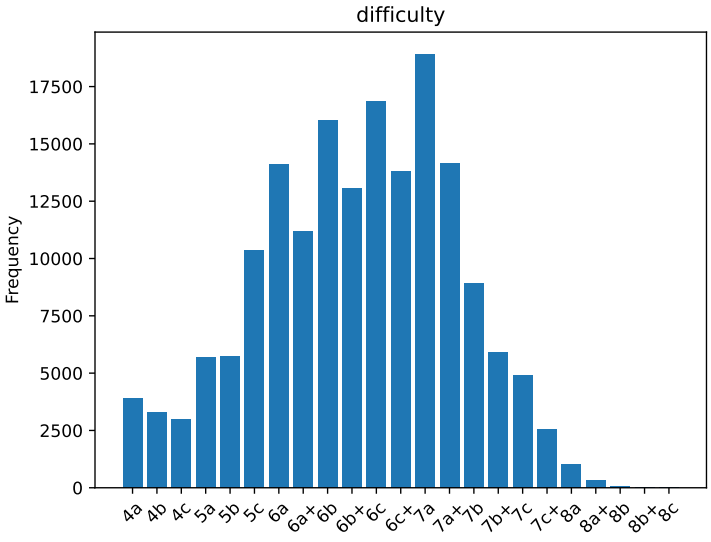


Figure 3: Histogram difficulties

2.3 DataConversion

Since the findings from the website data were the same as the ones in the sqlite database, I went forward and implemented a program to transform the data in a new sqlite table.

For a first model, i didn't distinguish between the different placement roles. By the way: sqlite has a column limit of 2000 if you ever need to know ;)

uuid	p1447	p1073	...	p4845	angle	difficulty
0062FB249C1645BD98ED3440E8B4669C	0	1	...	1	45	7b/V8
0070b5e309a2484887a97bc75b7c1bfb	1	0	...	0	20	6b/V4

Table 1: climbs_cleaned example

3 Logistic Regression (Base Model)

```
from sklearn.linear_model import LogisticRegression

logistic_model = LogisticRegression(
    max_iter=2000,
    n_jobs=16,
    solver='saga',
    verbose=1)
logistic_model.fit(X_train, y_train)
```

Listing 3: Simple logistic regression

This model took around 5 minutes to train. It has an accuracy of 19%, which sounds pretty bad.

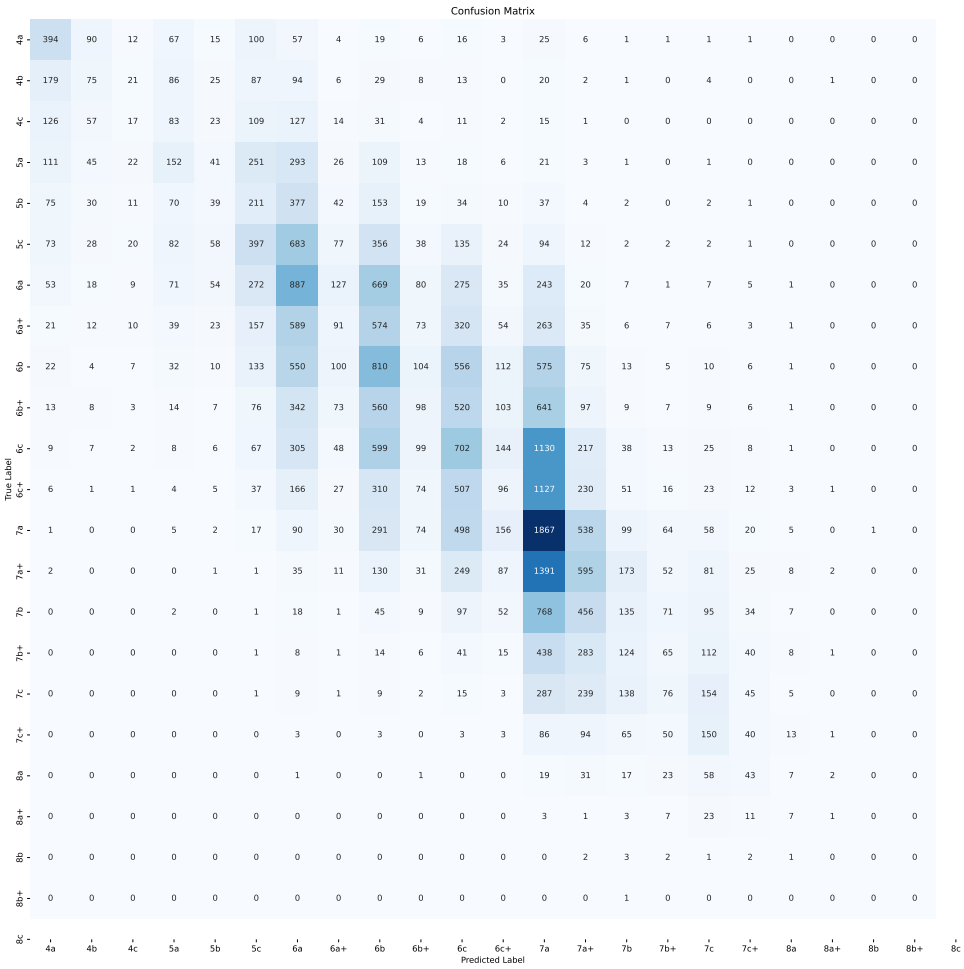


Figure 4: Confusion matrix logistic regression

A look at the confusion matrix shows that it isn't that bad. The difference between 7a and 7a+. I trained another model without the + difficulties (6a+ became 6a) and the accuracy became 32%.

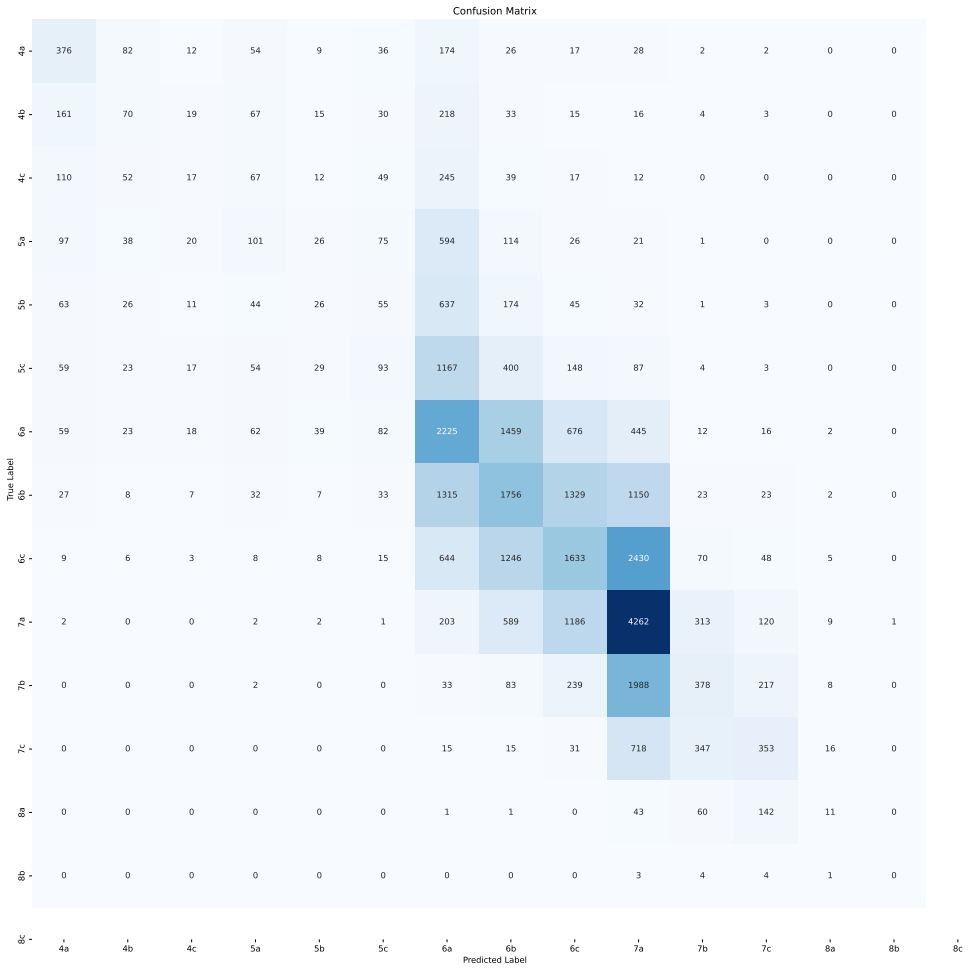


Figure 5: Confusion matrix logistic regression simplified

3.1 Evaluation

This is just a base model. Since the difficulties are close to each others, I will try a regression model, just to double check if this could perform better.

4 Linear Regression

The first step was to convert the classes back to ordinal values, I simply used an ordinal encoder

```
    categories = [
        '4a', '4b', '4c',
        '5a', '5b', '5c',
        '6a', '6a+', '6b', '6b+', '6c', '6c+',
        '7a', '7a+', '7b', '7b+', '7c', '7c+',
        '8a', '8a+', '8b', '8b+', '8c']
    encoder = OrdinalEncoder(categories=[categories])
    y = encoder.fit_transform(y.values.reshape(-1, 1))
```

Listing 4: Encode categories ordinal

Then I fitted a model and had a look at the results:

```
    from sklearn.linear_model import LinearRegression

    linear_model = LinearRegression()
    linear_model.fit(X_train, y_train)
```

Listing 5: Linear regression model

MAPE: 408364560813897.9

Learning: If you have 0 as outputs, MAPE is a very bad measurement, since you have ∞ percentage error

I therefore added 1 to the input and output vector

- MAPE: 34%
- MAE: 2.01
- MSE: 6.71

This doesn't seem too bad, around ± 2 grades wrong. Let's look at the confusion matrix:

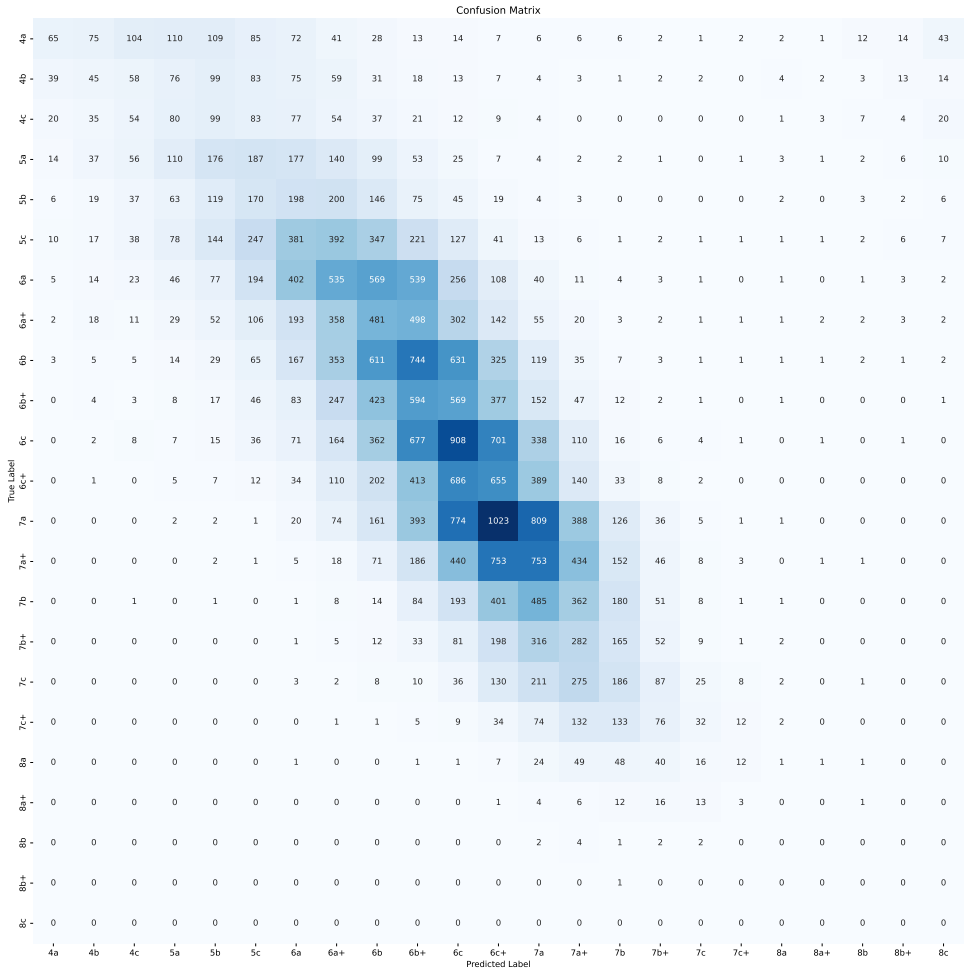


Figure 6: Confusion matrix linear regression

The accuracy is 16%, so it performs worse compared to the logistic regression.

But it wouldn't be fair to compare it without also comparing the simplified version (6a+ $\rightarrow$ 6a)

- MAPE: 25%
- MAE: 1.18
- MSE: 2.53

- Accuracy: 29% (converted back by rounding and clipping)

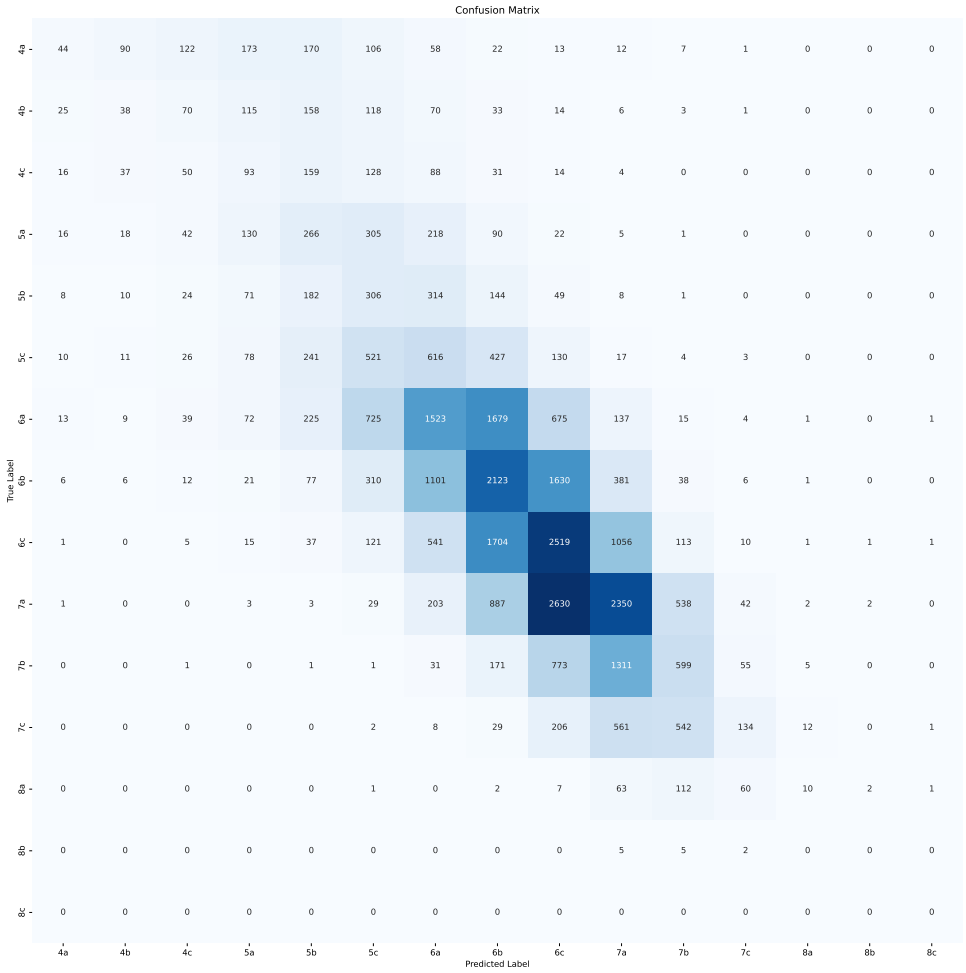


Figure 7: Confusion matrix linear regression simplified

The accuracy is lower, than the logistic regresison. I will therefore continue to go forward with classifiers instead of regressors.

5 XGBoost

XGBoost requires the labels to be in numerical format, therefore I used a LabelEncoder to encode my y values. Then I fitted an xgboost model:

```
from xgboost import XGBClassifier
clf = XGBClassifier(device='cuda')
clf.fit(X_train, y_train)
```

Listing 6: XGBoost model definition

The result, quite underwhelming, an accuracy of 19.7%.

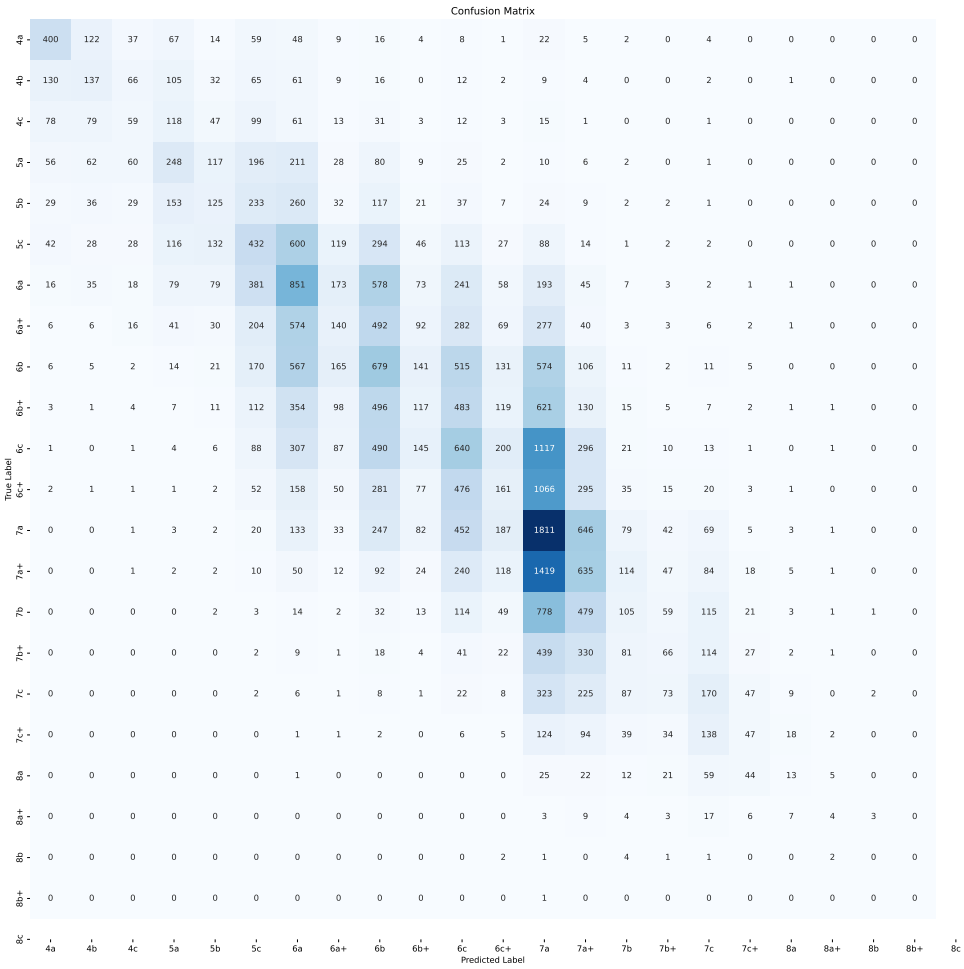


Figure 8: Confusion matrix XGBoost

Since there are several hyper parameters, I performed a gridsearch, which run over 10h on my computer. The best model performed slightly better

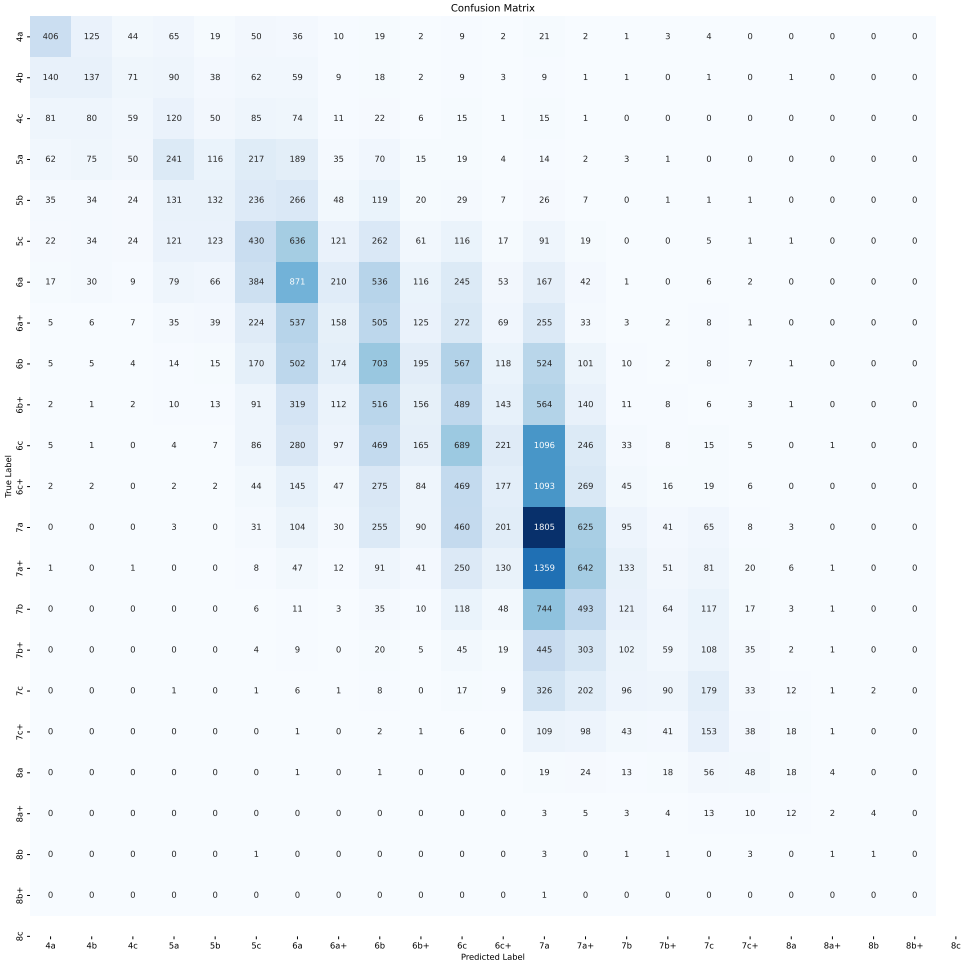


Figure 9: Confusion matrix XGBoost hyperparameter tuned

5.1 Evaluation

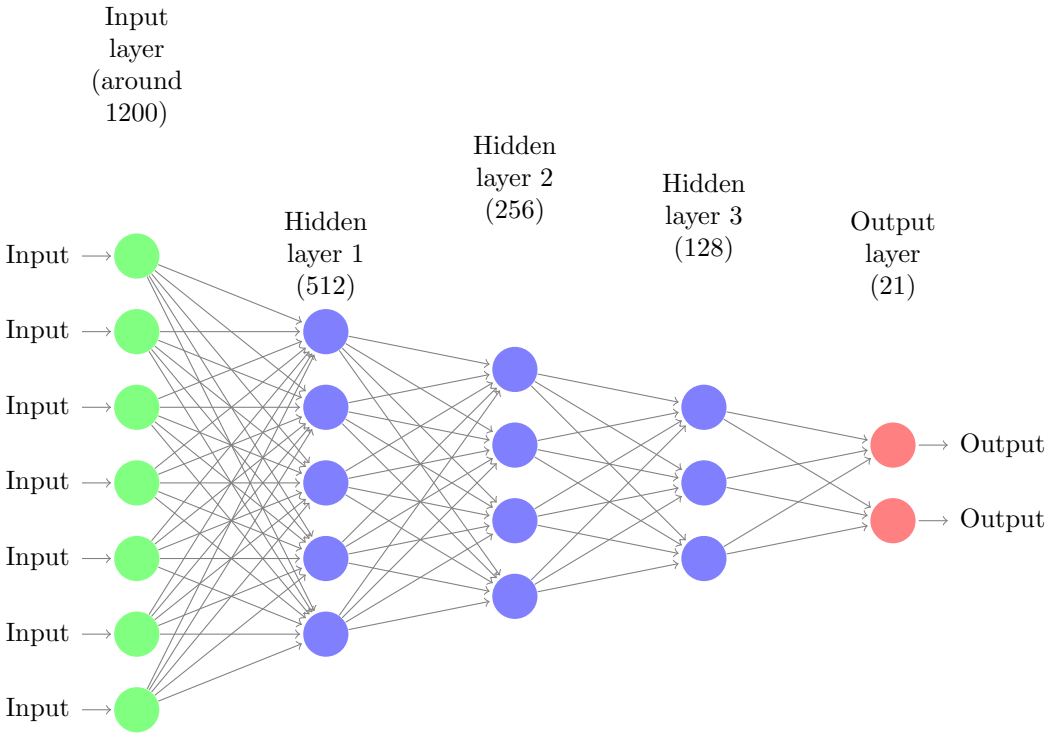
Sadly not that much better...

6 NN

Time for the big guns...

and hate tensorflow for not supporting Cuda 12.4..

Therefore let's learn PyTorch (or in modern words: let ChatGPT do it)



I created a neural network in the most basic architecture, gradually get smaller.

6.1 Evaluation

And the results were stunningly boring again, first the learning curve:

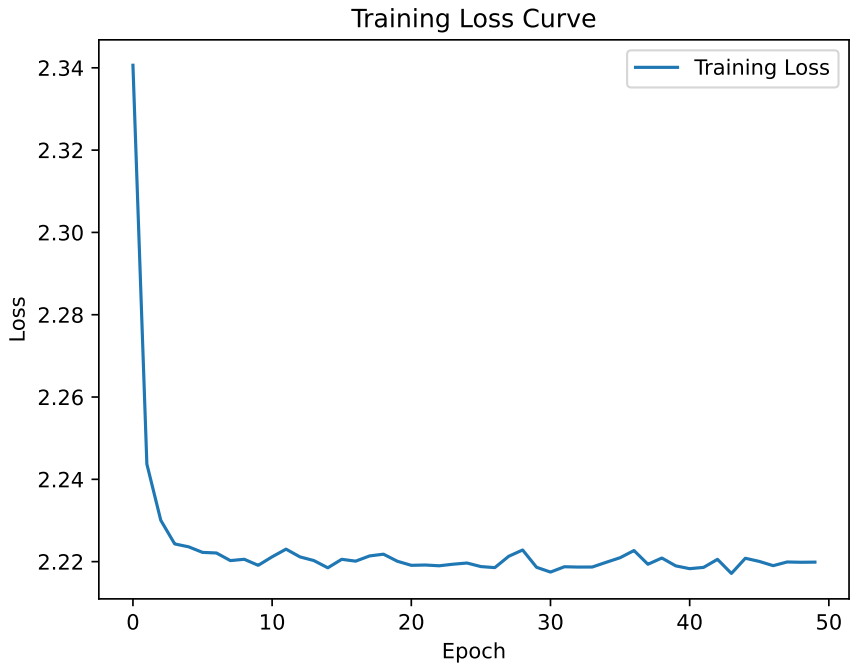
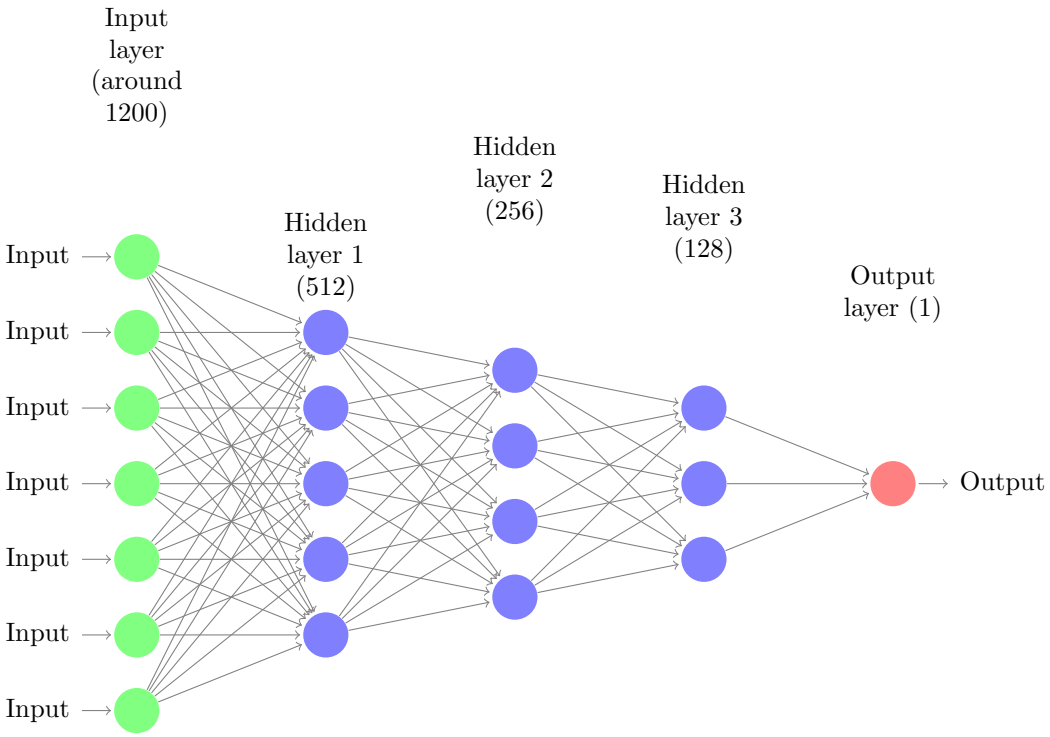


Figure 10: Learning curve of NN

As you can see, the loss doesn't really get better after a few epochs. And the accuracy is still in the 20s. At this point I think a regression solver might work better. Let's try a NN as a regression solver:

7 NN Regression



Let's start simple and reuse the classifier NN and just exchange the output layer to be a single node. and train it for 100 epochs:

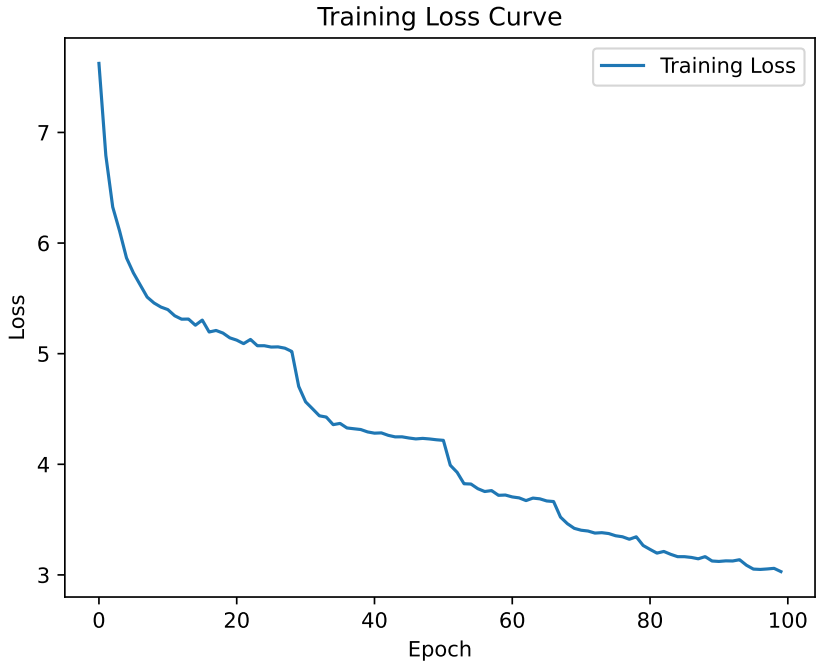


Figure 11: Learning curve of NN after 100 epochs

well that looks like I need some more training time, still the results after 100 epochs are ok:

- MSE: 3.5249
- MAE: 1.4194
- MAPE: 0.2341

The absolute error shows that it's ±1.41 grade. In my context this means a 6a could be a 6a+ or a 5c. I think this is pretty good (as a comparison: the baseline linear regression was MAE 2.01)

The longer trained model yielded similar results:

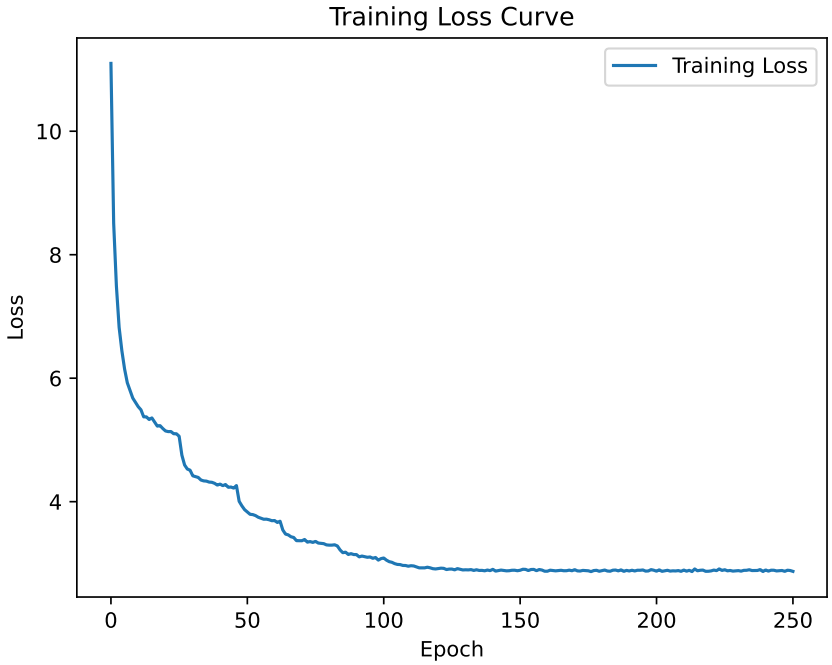


Figure 12: Learning curve of NN after 250 epochs

- MSE: 3.5030
- MAE: 1.4117
- MAPE: 0.2320

8 Better quality Train Dataset

Maybe the models perform not so good because there are a lot of routes that are only climbed by one climber. And this climber sets the grade, so there is no cross-validation.

The distribution graph shows that this is indeed the case for many routes:

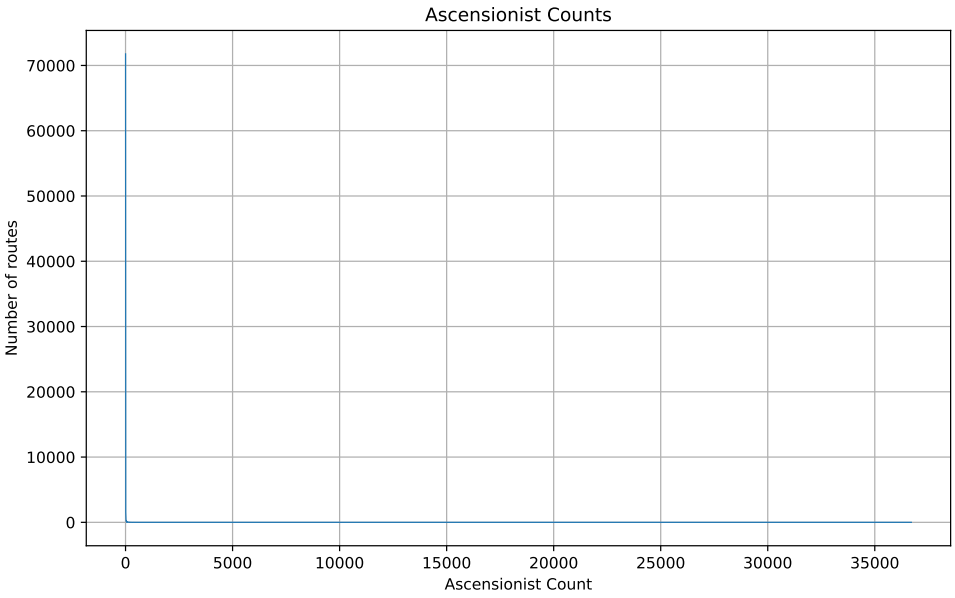


Figure 13: Ascensionist Count

I decided to train a model with only routes that have at least 3 ascensionist. By doin this I reduce the dataset size from around 173′000 to 51′000

9 NN quality-data

First again the learning curve

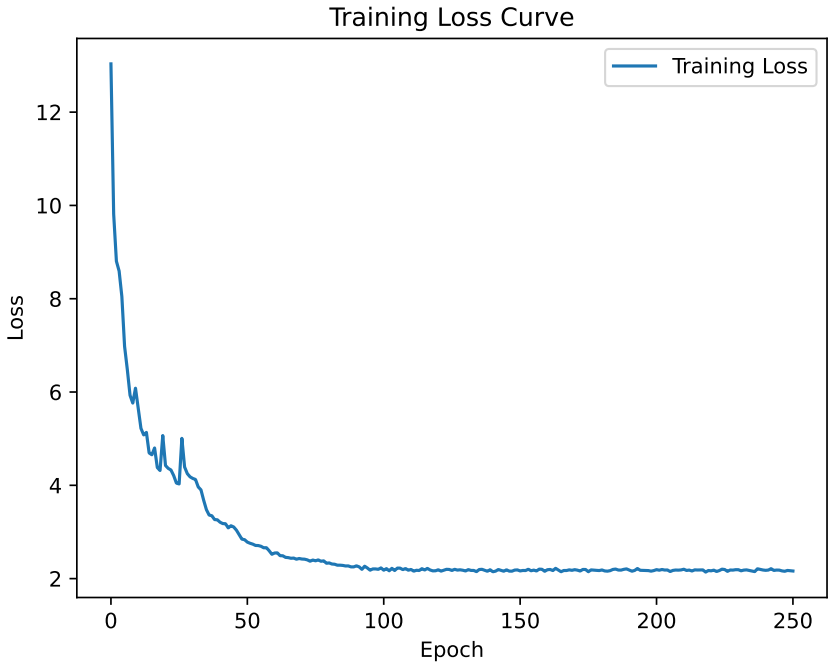


Figure 14: Learning curve of NN

After a bit more than 100 epochs, the learning curve converges. let's have a look at the results:

- MSE: 2.9150
- MAE: 1.2868
- MAPE: 0.2250

I'm now satisfied with the results, a MAE of 1.28 is quite good!

Still I'll train a model with simplified classes ($6a+ \rightarrow 6a$)

Look at these results:

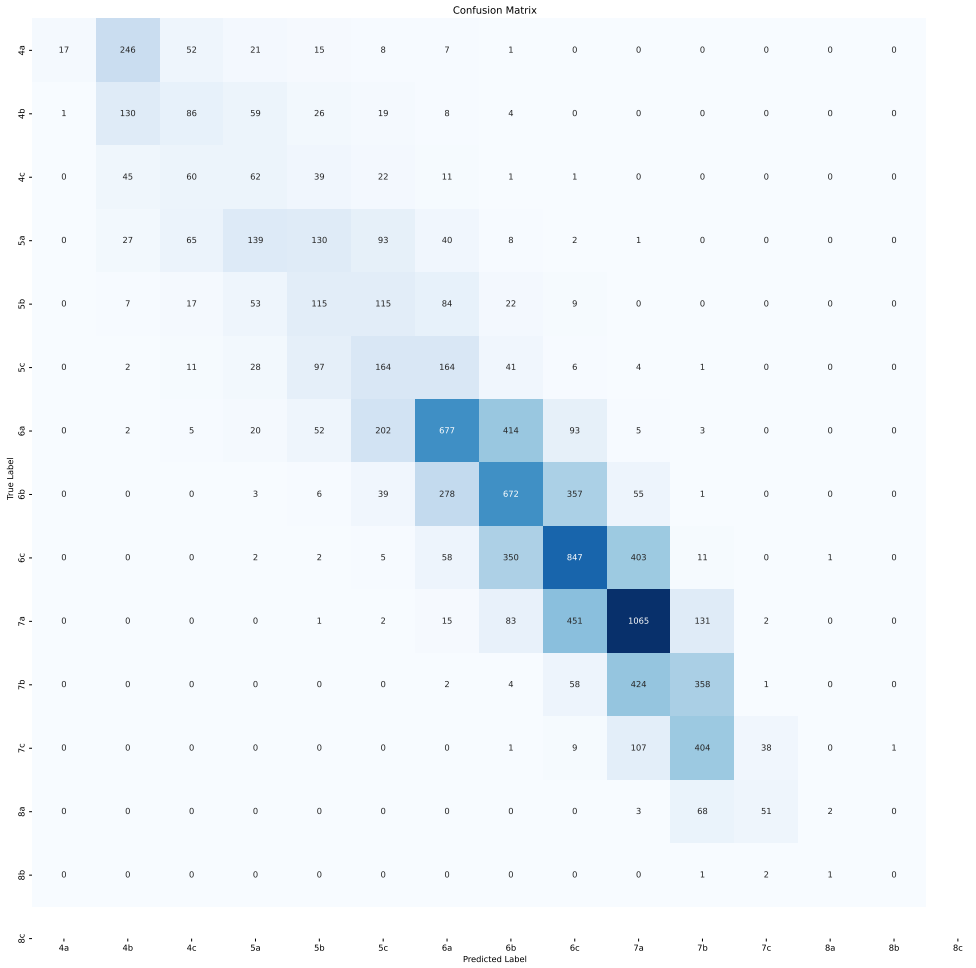


Figure 15: Confusion matrix NN simplified

They look pretty good, some metrics :

- MSE: 1.2149
- MAE: 0.8097
- MAPE: 0.1740