PYTHON FOR DATA ANALYSIS PROJECT

MADE BY: BALLOUK HOUSSEIN AND BADOURI MOUNA

GROUP DIA2

Table of Contents

INFORMATION ABOUT THE DATASET

THE DATASET

- The chosen dataset is : **Online Video Characteristics and Transcoding Time Dataset**
- Our dataset is actually composed of two tsv files: **youtube_videos.tsv** and **transcoding_mesurment.tsv** both in a tsv format. We have retained the first one for training and insight purposes , and the second one is the actual data that will be used to train our model.
- The dataset comes from a researcher named: **Tewodros Deneke** .

DATA EXPLORATION

- We started by importing the files into two panda dataframes.
- After some printing, we got the following statistical measures on the datasets.(It can be found on the notebook)

	duration	bitrate	bitrate(video)	height	width	frame rate	frame rate(est.)
count	168286.000000	168286.00000	168286.000000	168286.000000	168286.000000	168286.000000	168286.000000
mean	271.654184	730.62149	624.363025	561.018706	368.399701	24.564592	19.884441
std	552.881871	919.15473	860.955654	359.071569	201.274180	7.396615	11.435070
min	1.000000	0.00000	0.000000	180.000000	88.000000	0.000000	0.000000
25%	55.000000	289.00000	231.000000	320.000000	240.000000	23.980000	12.000000
50%	145.000000	459.00000	349.000000	480.000000	360.000000	29.920000	25.000000
75%	289.000000	826.00000	640.000000	640.000000	480.000000	29.970000	29.970000
max	25845.000000	22421.00000	22229.000000	2592.000000	1944.000000	59.940000	30.020000

insights

	duration	width	height	bitrate	framerate	i	p	b	frames	i_size	p_size	b_size	size	o_bitrate	o_framerate	o_width	o_height	umem	utime
count	68784.000000	68784.000000	68784.000000	6.878400e+04	68784.000000	68784.000000	68784.000000	68784.000000	68784.000000	6.878400e+04	6.878400e+04	68784.0	6.878400e+04	6.878400e+04	68784.000000	68784.000000	68784.000000	68784.000000	68784.000000
mean	286.413921	624.934171	412.572226	6.937015e+05	23.241321	100.868312	6531.692210	9.147854	6641.708377	2.838987e+06	2.218057e+07	0.0	2.502294e+07	1.395036e+06	21.190862	802.336357	503.825541	228224.717900	9.996355
std	287.257650	463.169869	240.615472	1.095628e+06	7.224848	84.764791	6075.871744	92.516177	6153.342453	4.325137e+06	5.097306e+07	0.0	5.414402e+07	1.749352e+06	6.668703	609.959797	315.970438	97430.878373	16.107429
min	31.080000	176.000000	144.000000	8.384000e+03	5.705752	7.000000	175.000000	0.000000	192.000000	1.164800e+04	3.384500e+04	0.0	1.918790e+05	5.600000e+04	12.000000	176.000000	144.000000	22508.000000	0.184000
25%	106.765000	320.000000	240.000000	1.343340e+05	15.000000	39.000000	2374.000000	0.000000	2417.000000	3.933950e+05	1.851539e+06	0.0	2.258222e+06	1.090000e+05	15.000000	320.000000	240.000000	216820.000000	2.096000
50%	239.141660	480.000000	360.000000	2.911500e+05	25.021740	80.000000	5515.000000	0.000000	5628.000000	9.458650e+05	6.166260e+06	0.0	7.881069e+06	5.390000e+05	24.000000	480.000000	360.000000	219480.000000	4.408000
75%	379.320000	640.000000	480.000000	6.529670e+05	29.000000	138.000000	9155.000000	0.000000	9232.000000	3.392479e+06	1.515506e+07	0.0	1.977335e+07	3.000000e+06	25.000000	1280.000000	720.000000	219656.000000	10.433000
max	25844.086000	1920.000000	1080.000000	7.628466e+06	48.000000	5170.000000	304959.000000	9407.000000	310129.000000	9.082855e+07	7.689970e+08	0.0	8.067111e+08	5.000000e+06	29.970000	1920.000000	1080.000000	711824.000000	224.574000

dataset

Regarding the insight dataset, we notice that on average (by average we mean median here) we have :

- The videos last 145 seconds .
- With a bitrate of 459 Kb/s with 349 Kb/s used for the video. This corresponds to the data encoding per second for audio and video (bitrate (video) only shows the btirate used for the video encoding, excluding the audio bitrate).
- The video size is 480 x 360 pixels, meaning a youtube video with a resolution of 360p.
- Last but not least, The videos run on ~ 30 FPS (frame per second) which is reasonable for this quality. (30-60 FPS is what we want for a fluid video.)

A similar analysis can be done on the actual dataset. However, we can see some extra columns that we will try to explain here :

1. The i, p, b frames are 3 different types of frames in a video :

    1.1 I-Frames are the frames that don't need other frames to decode it, but it is way less compressible than the p and b frames.

    1.2 P-Frames are the frames that can be decoded using its previous frame (which is most likely an I-Frame). This is more compressible than i-frames.

    1.3 B-Frames are the frames that can be decoded using both the previous and following frames. This is the most compressible frame (its previous frame is most likely an I-Frame or a P-Frame and the following is most likely an I-Frame).

2. i_size, p_size, b_size correspond to the size of the aforementionned frames, in bytes.

3. o_bitrate, o_framerate, o_width and o_height are the output values of the measures we explained previously (Their value on Youtube, aka the final video).

4. umem is the allocated memory for the transcoding (unit unknown : either Kb, or more likely Mb)

5. utime is the total transcoding time in seconds. This is the time taken to convert the video to another encoding type (codec).

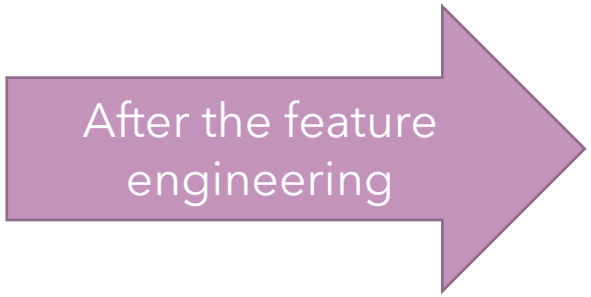
So, our goal is to predict the transcoding time, based on the features we explained (and kept) in the dataset.

DATA CLEANING

- Our dataset doesn't have any missing values. If it did, we could remove the rows with missing values or replace its value by the median or mean of the column in question (better). We notice that some columns will not affect the transcoding time and will not serve any purpose in our Machine Learning models. These columns are the following and have to be removed :
    1. The url of the video. Each video has a unique URL.
    2. The Youtube Id of the video. Each video has a unique Youtube Id
    3. The estimated frame rate. We will favor using the actual frame rate (since we have it in our dataset) for more accurate results.
- As for the category column, we think it will not influence the utime. It will, maybe, influence the duration (for example a music video will be ~ 3-5 minutes, but a gaming video can go up to 10 min-1h). So, there's no direct relationship between category and utime. But we will do some plots to make sure that our intuition is correct.

- So, we have 2 qualitative features and 18 quantitative features.
- Our response / target (aka utime) is quantitative. This means we have a **regression problem**. So we can use algorithms likes linear regression, decision trees (Random Forest, Bagging, Boosting) etc..
- HOWEVER, we see that we need some "feature engineering" for the following columns : category, codec and o_codec. They are of type "object" but we need them to be factors with few levels / categories.

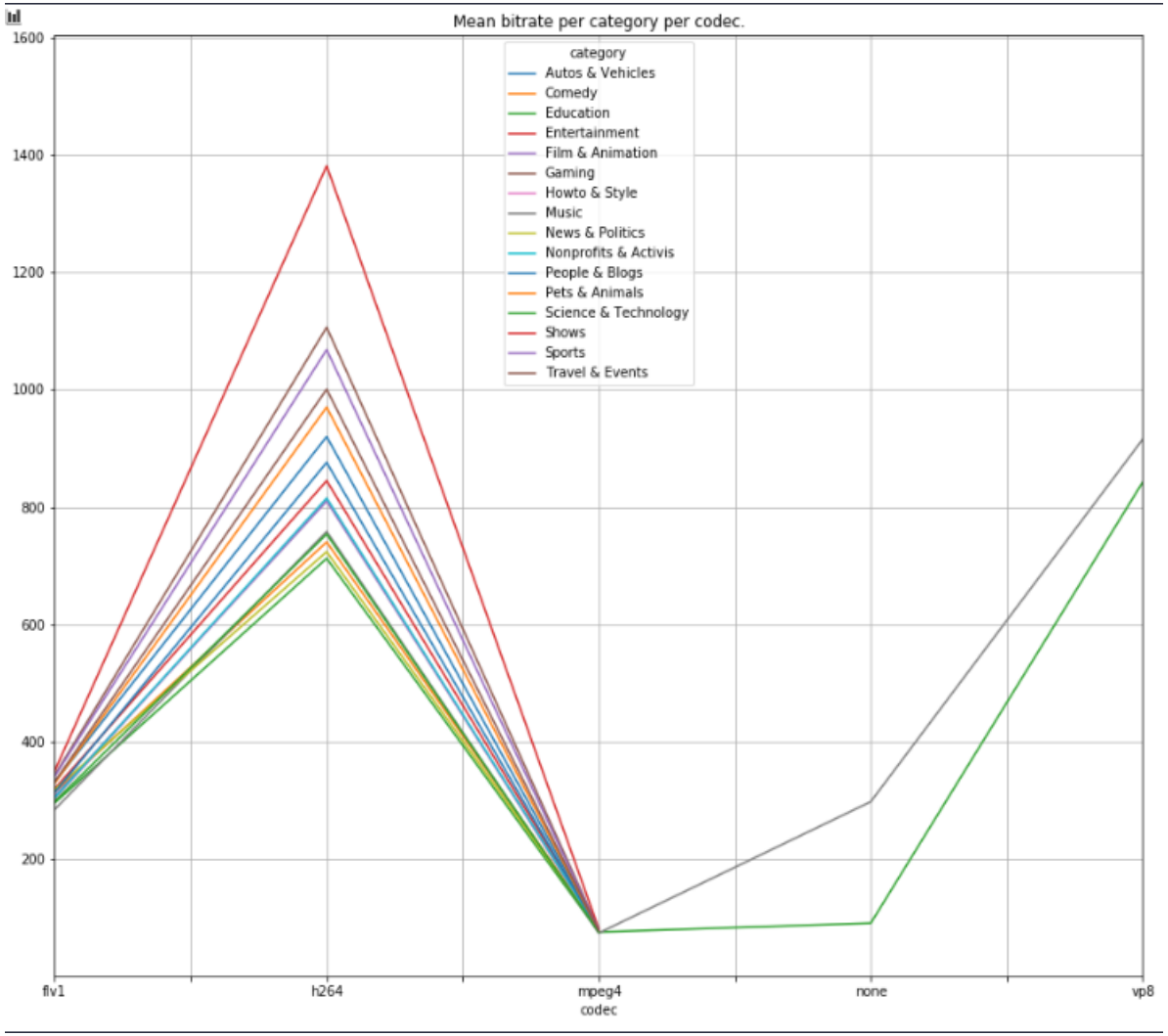
```
duration        float64
codec            object
width             int64
height            int64
bitrate           int64
framerate       float64
i                 int64
p                 int64
b                 int64
frames            int64
i_size            int64
p_size            int64
b_size            int64
size              int64
o_codec          object
o_bitrate         int64
o_framerate     float64
o_width           int64
o_height          int64
umem              int64
utime           float64
dtype: object
```




```
duration        float64
codec          category
width             int64
height            int64
bitrate           int64
framerate       float64
i                 int64
p                 int64
b                 int64
frames            int64
i_size            int64
p_size            int64
b_size            int64
size              int64
o_codec        category
o_bitrate         int64
o_framerate     float64
o_width           int64
o_height          int64
umem              int64
utime           float64
dtype: object
```

DATA VISUALIZATION

PLOTTING

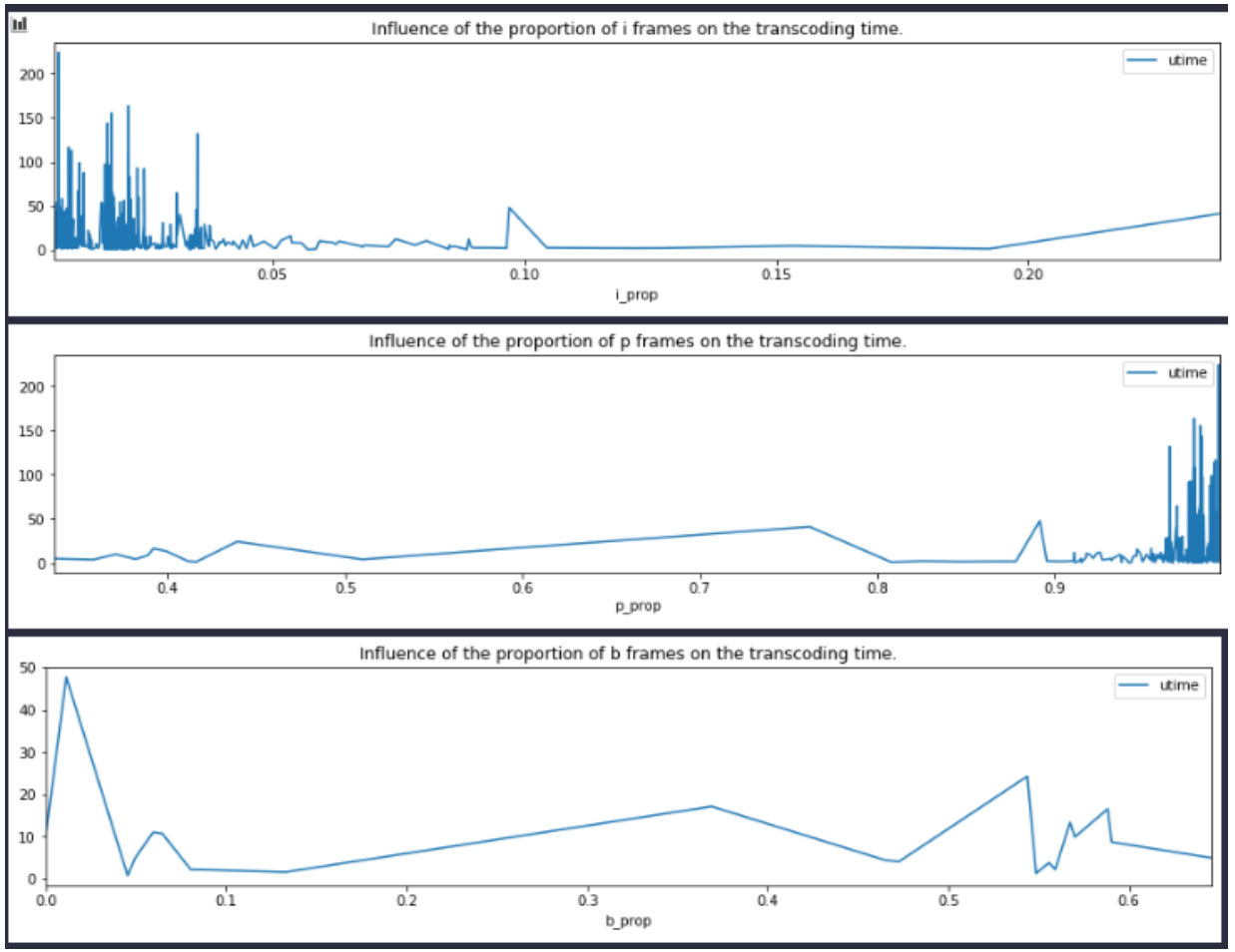


We can see that on average, Entertainement / shows / Travel videos tend to have a higher bitrate using h264 while education videos for example have the lowest bitrate.

H264 seems to have the fastest and best bitrate and speed over size ratio.

As we see, the category of the video doesn't really have a direct impact on the utime (or anything) but we can see from this graph that videos from certain categories tend to be longer and larger in size, so it needs higher bitrate or higher encoding time.

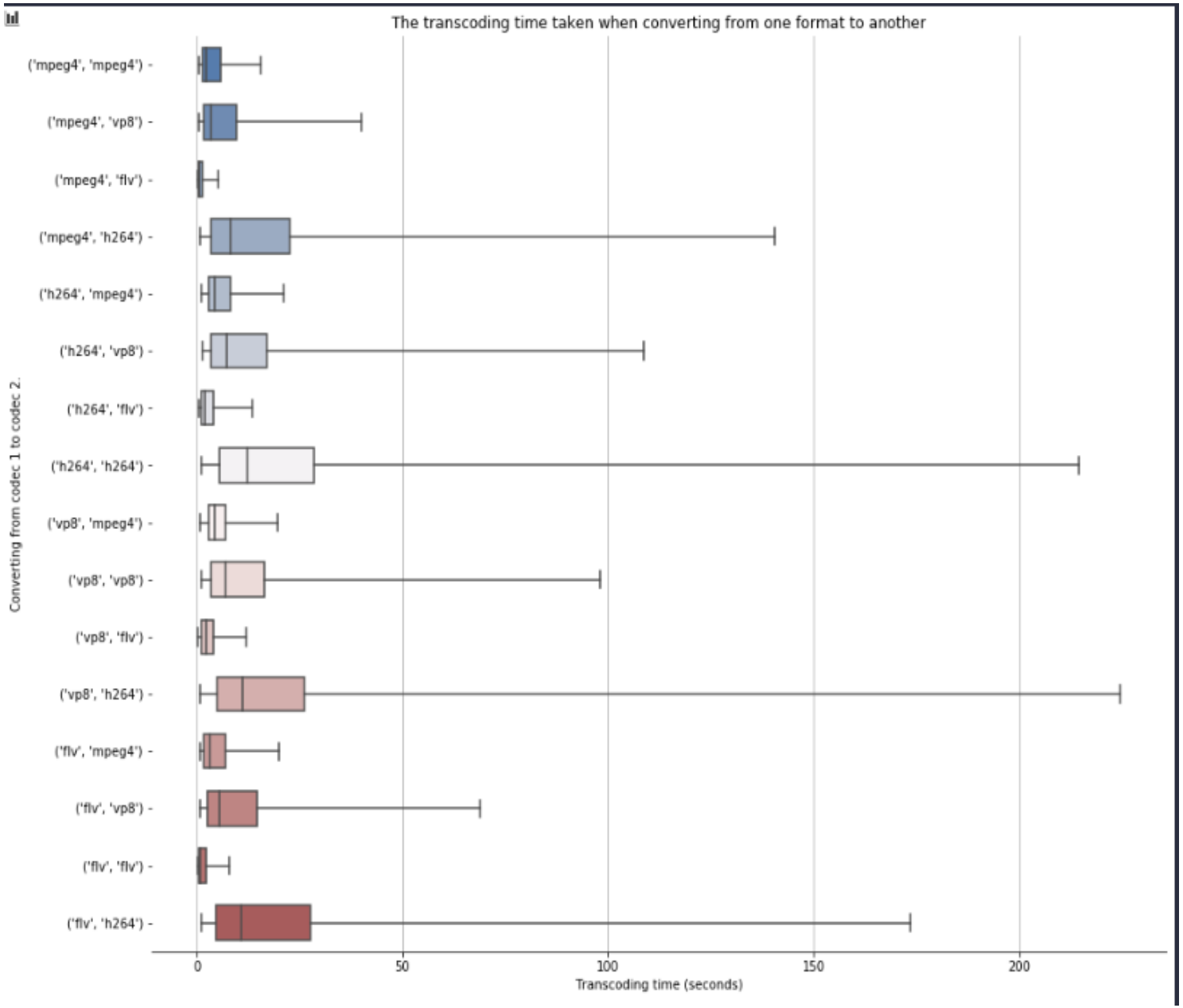
```
dataset['i_prop'] = dataset['i'] / dataset['frames']
dataset['p_prop'] = dataset['p'] / dataset['frames']
dataset['b_prop'] = dataset['b'] / dataset['frames']
dataset
```



We created the following columns, so we can interpret the influence of the numbers: i, b, p on the transcoding time.

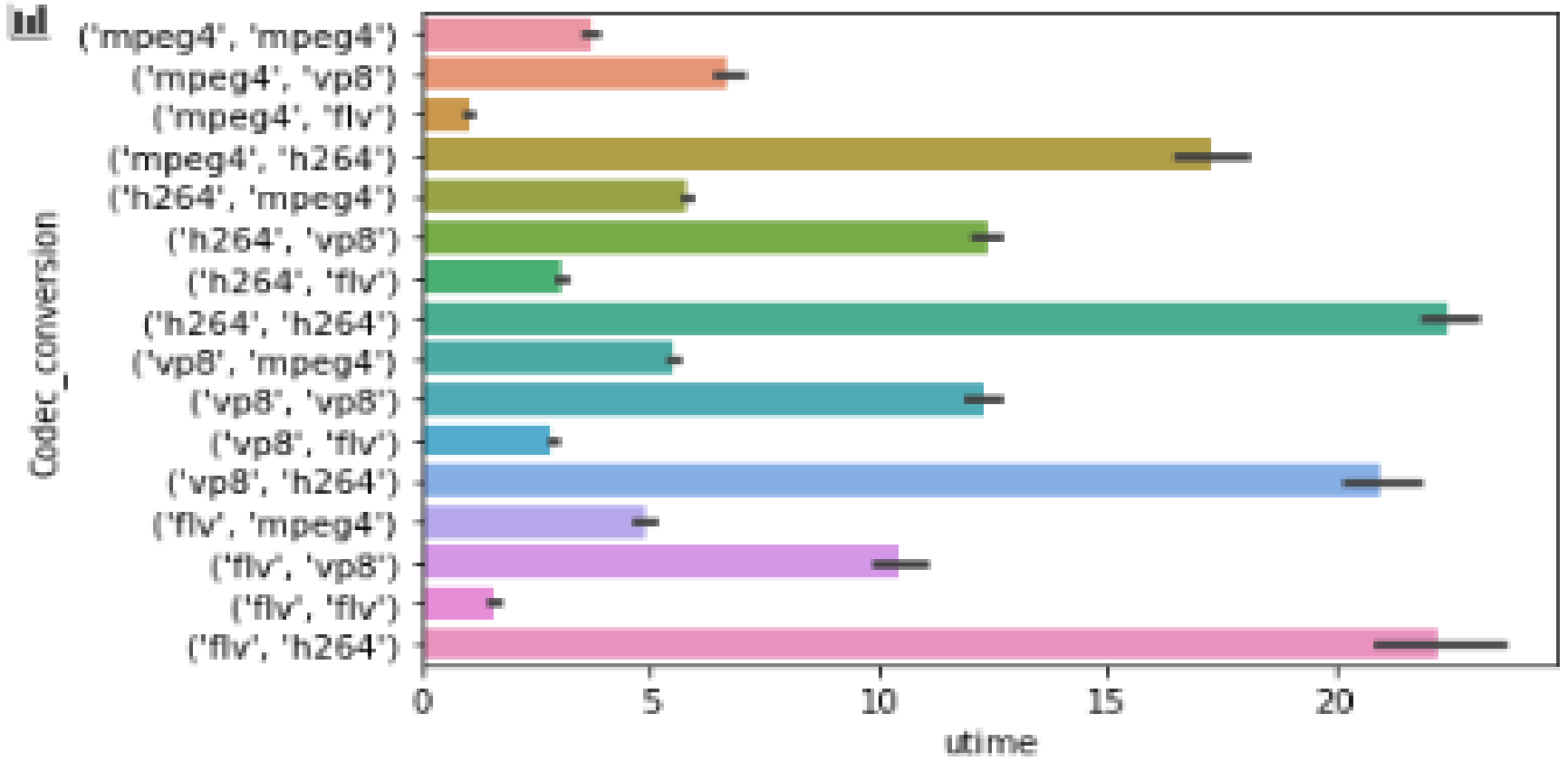
Plotting our pivot tables, we noticed the following trend :

1. The more i-frames we have, the lower the transcoding time. This is logical since i-frames don't need other frames to be decoded.
2. The more p-frames we have, the higher the transcoding time. Same, since we need the previous frame to decode it.
3. The more b-frames we have, the faster the transcoding time.

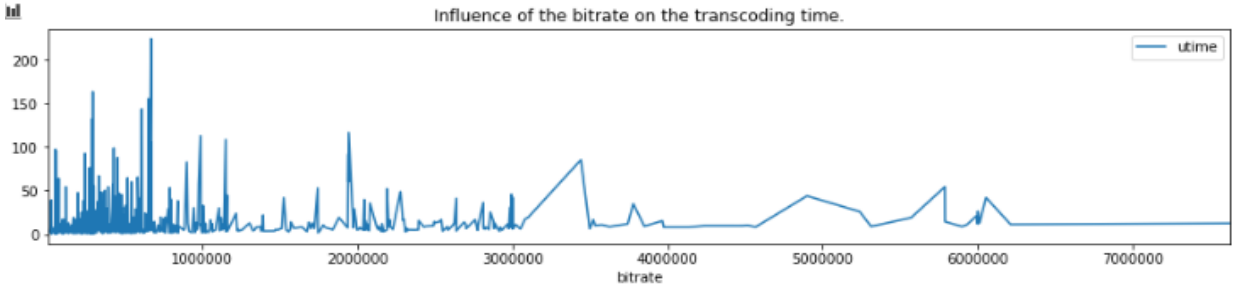


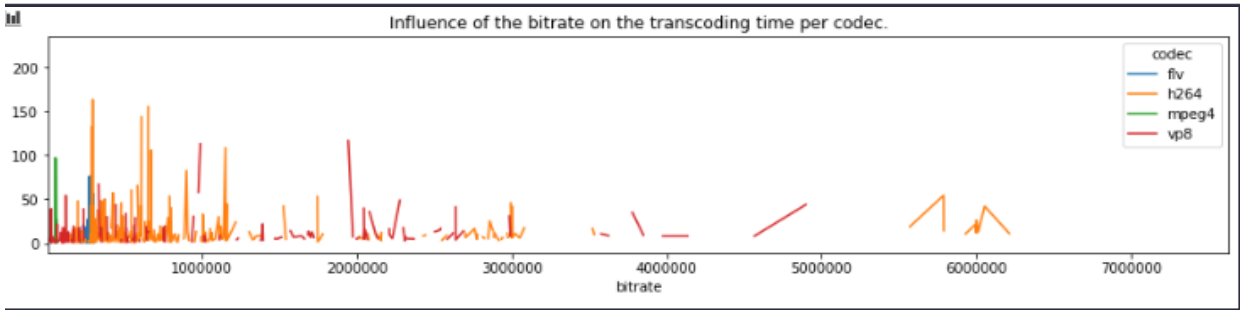
We notice, by looking at the median of each box, that the fastest transcoding time is when we are converting to 'flv' format, followed by the 'mpeg4' format not so far behind.

As for the 'h264' format, we see that on average it takes the longest to convert to. But we can see that its box plot is the largest out of all the other boxes, this means that we have a wide variety of transcoding time for this format (The interquartile Q3 - Q1 is larger than the rest. So, it covers more values.)



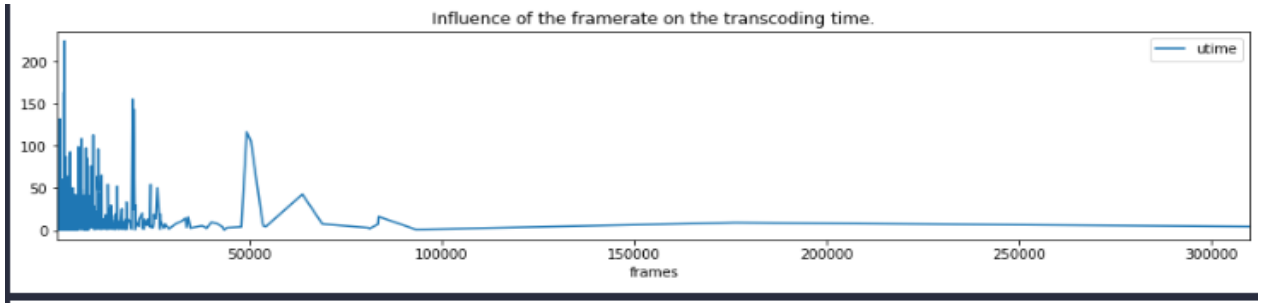
The following barplot confirms what has been discussed on the previous slide.

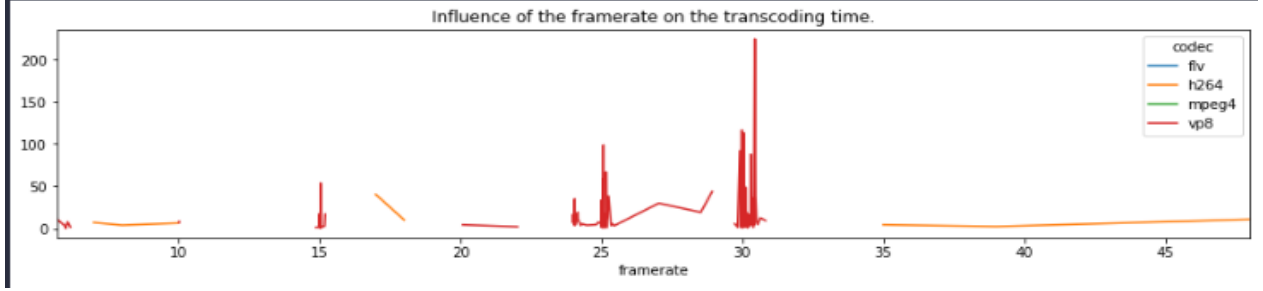




We see that the higher the bitrate, the faster the transcoding time, which makes sense.

However, we see that, sometimes, some codec take less time than others : example, at the far right of the graph, we see that vp8 and h264 took almost the same time even though vp8 had a much lower bitrate. =>But this could be an exception due to the size / duration of the video.



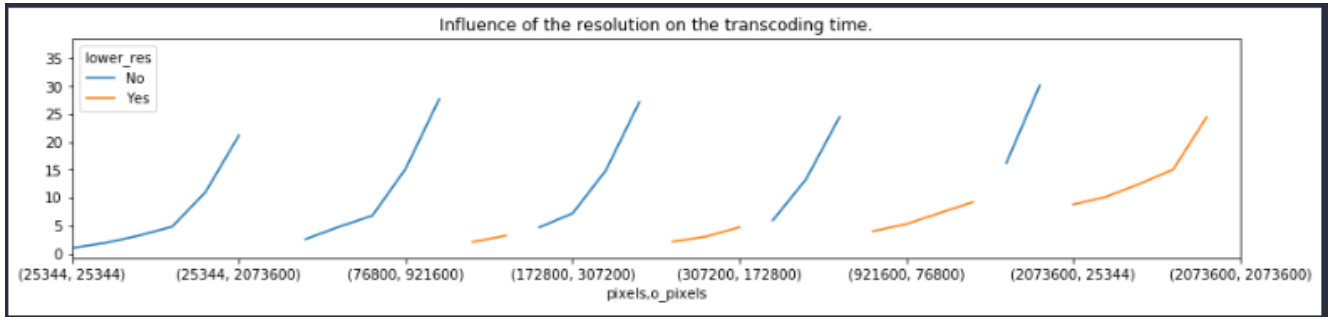


As for the influence of the framerate:
We notice that most if the videos have a framerate of 25-30 FPS (except some videos using h264 that have > 35 FPS and some using vp8 or h264 having < 20 FPS).

So, we can't really say that the more frames we have the faster the transcoding time, since it also depends on the framerate of the video too.

```
dataset['pixels'] = dataset['width'] * dataset['height']
dataset['o_pixels'] = dataset['o_width'] * dataset['o_height']
dataset['lower_res'] = np.where(dataset['o_pixels'] < dataset['pixels'], 'Yes','No')

pixel_table = dataset.pivot_table('utime',index=['pixels','o_pixels'],columns = 'lower_res')

pixel_table.plot(figsize=(15,3),grid=False,title="Influence of the proportion of i frames on the transcoding time.")
```

We created a column called pixels which stores the number of pixels in the video (which is width*height). Then we created a binary feature “lower_res” which takes yes as value output pixels are less than the input pixels, which means we are reducing the quality/resolution.



Using these columns, we plotted the graph, and we can understand that, in general, the more pixels we have (meaning the higher the resolution / quality of the video) the slower the transcoding time.

However, we can see that if we lowered the resolution when transcoding (meaning the number of output pixels is lower than the original number of pixels.), it will take less time to transcode. This means, that for the same number of pixels, it will take less time to transcode if we choose a lower resolution when transcoding than if we kept or increased the resolution.

MACHINE LEARNING

FEW INFORMATION BEFORE WE START

- Since our response is quantitative, we will use models that will solve a regression problem.
- To be able to split the data we will use the *train_test_split* function, with an 80-20 distribution, and seeding it with random_state in order to get the same result each round.
- We found out that if the column is always in the same value, sci-kit will think it's missing data, so we will use *dataset.fillna(dataset.mean()), this means that we'll replace the missing data with the mean of the column in question.*

DATA PRE-PROCESSING

- First of all we will split the dataset into X (it will take all the columns except utime) and y (that will take the response « utime »).
- We split the data into train and test using *train_test_split*.
- Since not all columns have the same unit of measurements, we will scale the features, however we have two categorical features (codec,o_codec), so we will one hot encode them (the code will be given in the next slide.
- *To assess our model, we will use cross_val_score, as well as the following metrics : mean_squared_error* and *R2_squared_error*.
  - Generally speaking, we want our MSE as close to 0 as possible (the smaller it is the better our model).
  - The closer R^2 is to 1 the more useful the model. Careful R^2 will tell us how helpful the model is (it represents the amount of data explained by our model), considering it is correct, so it will not judge if the model is correct or not.

```
from sklearn.preprocessing import StandardScaler, OneHotEncoder

scaler = StandardScaler()
one_hot_encoder = OneHotEncoder(sparse=False)

X_train_num  = X_train_tempo
X_train_num = X_train_num.drop(['codec','o_codec'],axis=1)

X_train_cat = X_train_tempo
X_train_cat = X_train_cat[['codec','o_codec']]

scaled_col = scaler.fit_transform(X_train_num)
encoded_col = one_hot_encoder.fit_transform(X_train_cat)

X_train = np.concatenate([scaled_col,encoded_col],axis=1)


X_test_num  = X_test_tempo
X_test_num = X_test_num.drop(['codec','o_codec'],axis=1)

X_test_cat = X_test_tempo
X_test_cat = X_test_cat[['codec','o_codec']]

scaled_col_test = scaler.transform(X_test_num)
encoded_col_test = one_hot_encoder.transform(X_test_cat)

X_test  = np.concatenate([scaled_col_test,encoded_col_test],axis=1)
```

TRAINING OUR MODEL

- We created two dictionaries, one will save the R^2 for each model and the other the MSE for each model as well.
- At first glance, we will train our data with default hyperparameters, seeding when necessary (trees for example).(two examples will be given in slide n°22 and 23)
- We will only use models that we have seen during this course and the ML course.
  - We can do a function that will test all the available models and choose the best one out of all of them. However, we deemed that it will not be very useful, as you will see next, we will have scores of 98 or 99%.
  - We plotted the correlation between all variables, so we can re-train our models with only the significant variables, however for the same causes just above we will skip it . (The correlation plot will be on the next slide and on the notebook).

	duration	width	height	bitrate	framerate	i	p	b	frames	i_size	p_size	b_size	size	o_bitrate	o_framerate	o_width	o_height	umem	utime
duration	1	0.072634	0.0504463	0.00475852	-0.0305593	0.773659	0.850175	0.0615231	0.851053	0.4266	0.406651	nan	0.416951	0.000148989	0.000698152	0.000377109	0.000577201	0.00928152	0.00553252
width	0.072634	1	0.990467	0.815328	0.396052	0.0778287	0.238482	-0.0638234	0.235592	0.601446	0.560013	nan	0.575222	-0.00102549	-0.000244993	-8.75601e-05	-6.09139e-05	0.0351095	0.129861
height	0.0504463	0.990467	1	0.799082	0.45631	0.0933766	0.23447	-0.0690761	0.231766	0.592107	0.534646	nan	0.550593	-0.00100408	-0.000204195	-2.83569e-05	-1.21982e-05	0.0414518	0.128479
bitrate	0.00475852	0.815328	0.799082	1	0.255464	0.0592625	0.112427	-0.0459381	0.111137	0.44632	0.616313	nan	0.615846	-0.00132779	-0.000283893	-9.57003e-05	-8.83725e-05	0.0431424	0.1552
framerate	-0.0305593	0.396052	0.45631	0.255464	1	0.263127	0.332138	0.0267589	0.331983	0.306534	0.196526	nan	0.209523	-0.00168384	0.00018931	0.000102464	9.98094e-05	0.0596857	0.0793361
i	0.773659	0.0778287	0.0933766	0.0592625	0.263127	1	0.831221	0.12247	0.836373	0.537019	0.393288	nan	0.413247	-0.000739009	0.000139367	0.00121953	0.00123883	0.0342065	0.0184892
p	0.850175	0.238482	0.23447	0.112427	0.332138	0.831221	1	0.0654222	0.999844	0.665995	0.605745	nan	0.623506	-0.00178834	0.000239999	0.000393492	0.000481633	0.0312829	0.0332006
b	0.0615231	-0.0638234	-0.0690761	-0.0459381	0.0267589	0.12247	0.0654222	1	0.0813207	-0.0309689	-0.0264878	nan	-0.0264062	0.00240789	0.000262522	0.00080441	0.000834745	0.00602944	0.00513953
frames	0.851053	0.235592	0.231766	0.111137	0.331983	0.836373	0.999844	0.0813207	1	0.664543	0.603138	nan	0.620952	-0.0017398	0.000242844	0.000417432	0.000505185	0.0314509	0.0331146
i_size	0.4266	0.601446	0.592107	0.44632	0.306534	0.537019	0.665995	-0.0309689	0.664543	1	0.713713	nan	0.75178	-0.00239849	-0.000284909	0.000225884	0.000181558	0.0359314	0.0647115
p_size	0.406651	0.560013	0.534646	0.616313	0.196526	0.393288	0.605745	-0.0264878	0.603138	0.713713	1	nan	0.998433	-0.00366138	-9.55236e-05	0.000217516	0.00020775	0.0304608	0.0976444
b_size	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan
size	0.416951	0.575222	0.550593	0.615846	0.209523	0.413247	0.623506	-0.0264062	0.620952	0.75178	0.998433	nan	1	-0.0036307	-0.00011498	0.000221752	0.000209024	0.0315409	0.0970958
o_bitrate	0.000148989	-0.00102549	-0.00100408	-0.00132779	-0.00168384	-0.000739009	-0.00178834	0.00240789	-0.0017398	-0.00239849	-0.00366138	nan	-0.0036307	1	-0.000338349	4.84091e-05	8.09108e-05	0.00158459	0.155479
o_framerate	0.000698152	-0.000244993	-0.000204195	-0.000283893	0.00018931	0.000139367	0.000239999	0.000262522	0.000242844	-0.000284909	-9.55236e-05	nan	-0.00011498	-0.000338349	1	0.000534486	0.000541592	0.00216439	0.104043
o_width	0.000377109	-8.75601e-05	-2.83569e-05	-9.57003e-05	0.000102464	0.00121953	0.000393492	0.00080441	0.000417432	0.000225884	0.000217516	nan	0.000221752	4.84091e-05	0.000534486	1	0.994492	0.388346	0.523388
o_height	0.000577201	-6.09139e-05	-1.21982e-05	-8.83725e-05	9.98094e-05	0.00123883	0.000481633	0.000834745	0.000505185	0.000181558	0.00020775	nan	0.000209024	8.09108e-05	0.000541592	0.994492	1	0.382064	0.519649
umem	0.00928152	0.0351095	0.0414518	0.0431424	0.0596857	0.0342065	0.0312829	0.00602944	0.0314509	0.0359314	0.0304608	nan	0.0315409	0.00158459	0.00216439	0.388346	0.382064	1	0.663301
utime	0.00553252	0.129861	0.128479	0.1552	0.0793361	0.0184892	0.0332006	0.00513953	0.0331146	0.0647115	0.0976444	nan	0.0970958	0.155479	0.104043	0.523388	0.519649	0.663301	1

```python
# KNN

from sklearn.neighbors import KNeighborsRegressor

KNN = KNeighborsRegressor()
KNN.fit(X_train,y_train)

pred_KNN = KNN.predict(X_test)

print(f"Using KNN : {round(KNN.score(X_test,y_test) * 100,2)} % of the data is explained (R2) and we have a MSE error of ~> {round(mean_squared_error(y_test,pred_KNN),2)}.")


score_dict["KNN"] = [round(KNN.score(X_test,y_test) * 100,2)]
mse_dict["KNN"]= [round(mean_squared_error(y_test,pred_KNN),2)]

cross_val = cross_val_score(KNN,X_train,y_train,cv=5)

print(f"\nUsing cross validation, we get the following scores at each fold :\n\n{cross_val}")
```

```
Using KNN : 93.62 % of the data is explained (R2) and we have a MSE error of ~> 16.33.

Using cross validation, we get the following scores at each fold :

[0.91863823 0.92265796 0.93006439 0.92658839 0.92622835]
```

```python
# Random Forest

RF = RandomForestRegressor(random_state=14)
RF.fit(X_train,y_train)

pred_RF = RF.predict(X_test)

print(f"Using Random Forest : {round(RF.score(X_test,y_test) * 100,2)} % of the data is explained (R2) and we have a MSE error of ~> {round(mean_squared_error(y_test,pred_RF),2)}.")

score_dict["Random Forest"] = round(RF.score(X_test,y_test) * 100,2)
mse_dict["Random Forest"]= round(mean_squared_error(y_test,pred_RF),2)

cross_val = cross_val_score(RF,X_train,y_train,cv=5)

print(f"\nUsing cross validation, we get the following scores at each fold :\n\n{cross_val}")
```

```
Using Random Forest : 98.99 % of the data is explained (R2) and we have a MSE error of ~> 2.59.

Using cross validation, we get the following scores at each fold :

[0.9862611  0.98285495 0.98439427 0.98339799 0.98369596]
```

COMPARING OUR MODELS

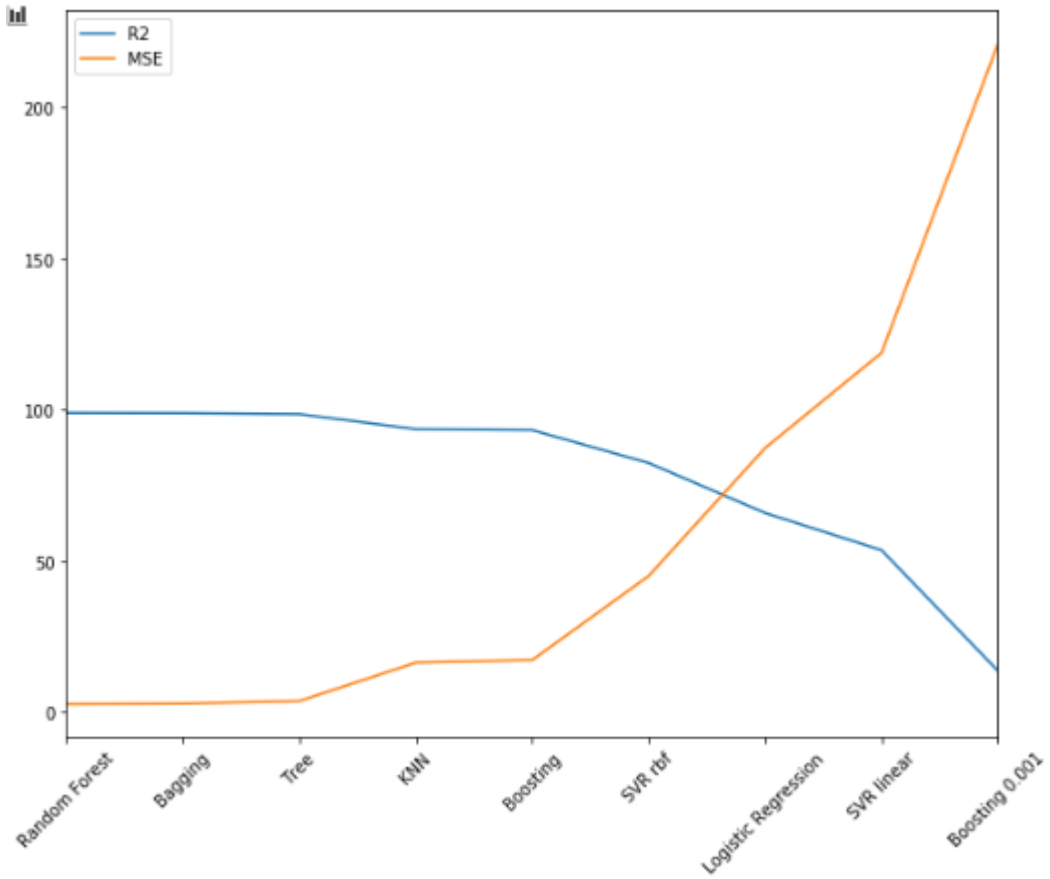
First, we will create the panda dataframe from the dictionaries in the following fashion:

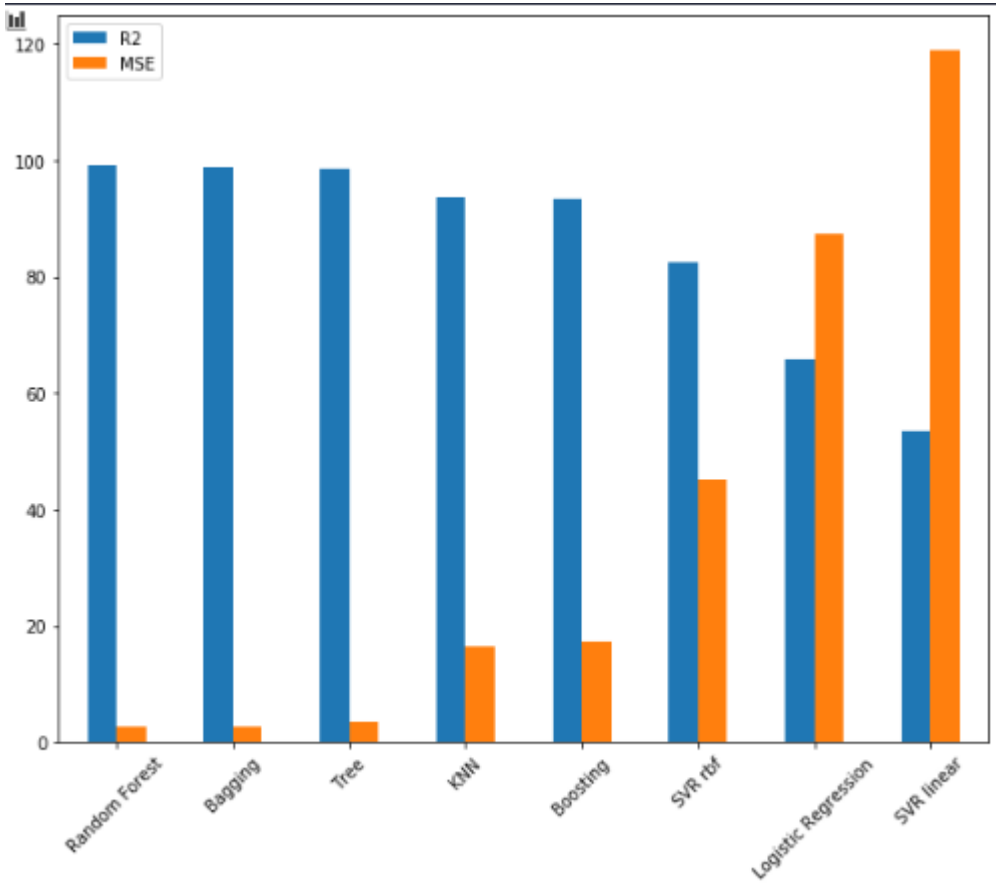
```
performances = pd.DataFrame.from_dict(score_dict).T
MSE = pd.DataFrame.from_dict(mse_dict).T

dataframe_no_grid_search = pd.concat([performances,MSE],axis = 1)
dataframe_no_grid_search.columns = ["R2","MSE"]
dataframe_no_grid_search = dataframe_no_grid_search.sort_values(by='R2',ascending=False)

dataframe_no_grid_search
```

	R2	MSE
Random Forest	98.99	2.59
Bagging	98.92	2.75
Tree	98.58	3.63
KNN	93.62	16.33
Boosting	93.28	17.19
SVR rbf	82.41	45.02
Logistic Regression	65.86	87.36
SVR linear	53.55	118.87
Boosting 0.001	13.58	221.16

The following function will be used to test the hyperparameters during grid search:

```
def test_hyperparametres(algo, hyperparametres):
    grid       = grid_search.GridSearchCV(algo, hyperparametres, n_jobs=-1)
    grid.fit(X_train, y_train)

    return grid.best_score_, grid.best_estimator_
```





First, we plot our dataframe (first figure), we see that for all models when R^2 goes up the MSE goes down, which is logical. Here we see all the models we used in our notebook .

But we see that for the boosting model with a learning rate of 0.001, the R^2 is so low and the MSE is so high, which is a bad model, so we will drop it from the dataframe, and we will plot it again, but this time using barplots, so it is easier to interpret.

GRID SEARCH

- Now that we trained our models, we will use grid search to tune them and find the most optimal hyperparameters.
  - This is an important step especially for the trees since they can overfit the data (their high scores are already a sign of over-fitting)
- Looking at the plots of slide n°25, we see that Decision trees performed really well alongside KNN. So, for our grid search, we will take all the models except logistic regression and SVR.
- Let's explain how the grid search works:
  - Find the best hyperparameters to tune: example min_sample_leaf and min_sample_split for the trees, which are used to prevent the model from memorizing the results, hence avoiding the over-fitting.
  - We then give 3 to 4 values around the default value of each chosen hyperparameters.
  - Then the function will test all combinations and will retain the one with the best score.
  - We redo steps 2 and 3 but around returned value, until we are happy with the results or until the score converges.
- On the next slides we will show a full example code of one model : notice how we reduced the hyperparameter intervals, based on what is returned by the grid search.

```python
# Iter 1

algo = BaggingRegressor(random_state=10)

params = {"n_estimators" : [10,50,100,150],
          "oob_score" : [True,False],  # True if we want to test our model on unseen data
          "max_samples":[0.5,0.7,1.0],
          "max_features":[0.5,0.7,1.0],
          #"warm_start":[True,False], # all three are available only if oob_score is False
          #"bootstrap":[True,False],
          #"bootstrap_features":[True,False]
          }

grid = GridSearchCV(algo, params, n_jobs=-1)
grid.fit(X_train, y_train)

print (f"We get the following score : {grid.best_score_} using these hyperparameters :\n {grid.best_estimator_}")

# R2 and MSE

pred = grid.best_estimator_.predict(X_test)

print(f"\n~> {round(grid.best_estimator_.score(X_test,y_test) * 100,2)} % of the data is explained (R2) and we have a MSE error of ~> {round(mean_squared_error(y_test,pred),2)}.")


score_dict_opti["Bagging"] = [round(grid.best_estimator_.score(X_test,y_test) * 100,2)]
score_dict_opti_train["Bagging"] = [round(grid.best_score_*100,2)]
mse_dict_opti["Bagging"]= [round(mean_squared_error(y_test,pred),2)]
```

```
We get the following score : 0.9828664717954971 using these hyperparameters :
 BaggingRegressor(base_estimator=None, bootstrap=True, bootstrap_features=False,
                  max_features=1.0, max_samples=1.0, n_estimators=150,
                  n_jobs=None, oob_score=True, random_state=10, verbose=0,
                  warm_start=False)

~> 99.1 % of the data is explained (R2) and we have a MSE error of ~> 2.31.
```

```
    # Iter 2

    algo = BaggingRegressor(random_state=10)

    params = {"n_estimators" : [125,150,175],
              "oob_score" : [True],
              "max_samples":[0.9,0.95,0.99,1.0],
              "max_features":[0.9,0.95,0.99,1.0],
              #"warm_start":[True,False], # all three are available only if oob_score is False
              #"bootstrap":[True,False],
              #"bootstrap_features":[True,False]
              }

    grid = GridSearchCV(algo, params, n_jobs=-1)
    grid.fit(X_train, y_train)

    print (f"We get the following score : {grid.best_score_} using these hyperparameters :\n {grid.best_estimator_}")

    # R2 and MSE

    pred = grid.best_estimator_.predict(X_test)

    print(f"\n~> {round(grid.best_estimator_.score(X_test,y_test) * 100,2)} % of the data is explained (R2) and we have a MSE error of ~> {round(mean_squared_error(y_test,pred),2)}.")


    score_dict_opti["Bagging"] = [round(grid.best_estimator_.score(X_test,y_test) * 100,2)]
    score_dict_opti_train["Bagging"] = [round(grid.best_score_*100,2)]
    mse_dict_opti["Bagging"]= [round(mean_squared_error(y_test,pred),2)]
```

```
We get the following score : 0.9841656768776839 using these hyperparameters :
 BaggingRegressor(base_estimator=None, bootstrap=True, bootstrap_features=False,
                  max_features=0.95, max_samples=1.0, n_estimators=175,
                  n_jobs=None, oob_score=True, random_state=10, verbose=0,
                  warm_start=False)

~> 99.09 % of the data is explained (R2) and we have a MSE error of ~> 2.33.
```

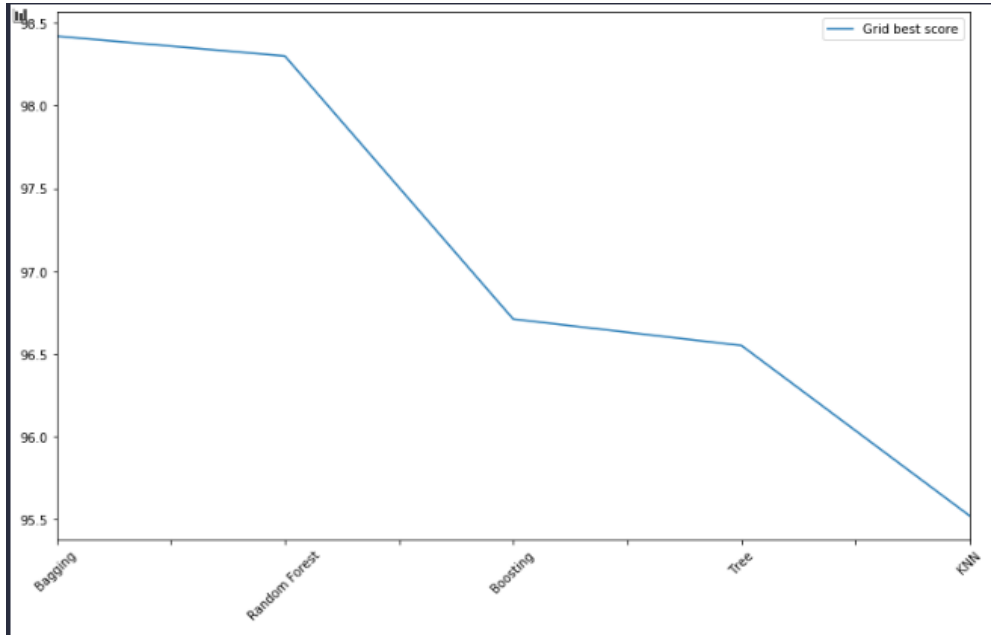
```
    # Iter 3

    algo = BaggingRegressor(random_state=10)

    params = {"n_estimators" : [175], # We keep 150 since with 175 we get a better score but slightly worse R2 and higher MSE ~> overfitting
              "oob_score" : [True],
              "max_samples":[1.0],
              "max_features":[0.925,0.95,0.975],
              #"warm_start":[True,False], # all three are available only if oob_score is False
              #"bootstrap":[True,False],
              #"bootstrap_features":[True,False]
              }

    grid = GridSearchCV(algo, params, n_jobs=-1)
    grid.fit(X_train, y_train)

    print (f"We get the following score : {grid.best_score_} using these hyperparameters :\n {grid.best_estimator_}")

    # R2 and MSE

    pred = grid.best_estimator_.predict(X_test)

    print(f"\n~> {round(grid.best_estimator_.score(X_test,y_test) * 100,2)} % of the data is explained (R2) and we have a MSE error of ~> {round(mean_squared_error(y_test,pred),2)}.")


    score_dict_opti["Bagging"] = [round(grid.best_estimator_.score(X_test,y_test) * 100,2)]
    score_dict_opti_train["Bagging"] = [round(grid.best_score_*100,2)]
    mse_dict_opti["Bagging"]= [round(mean_squared_error(y_test,pred),2)]
```

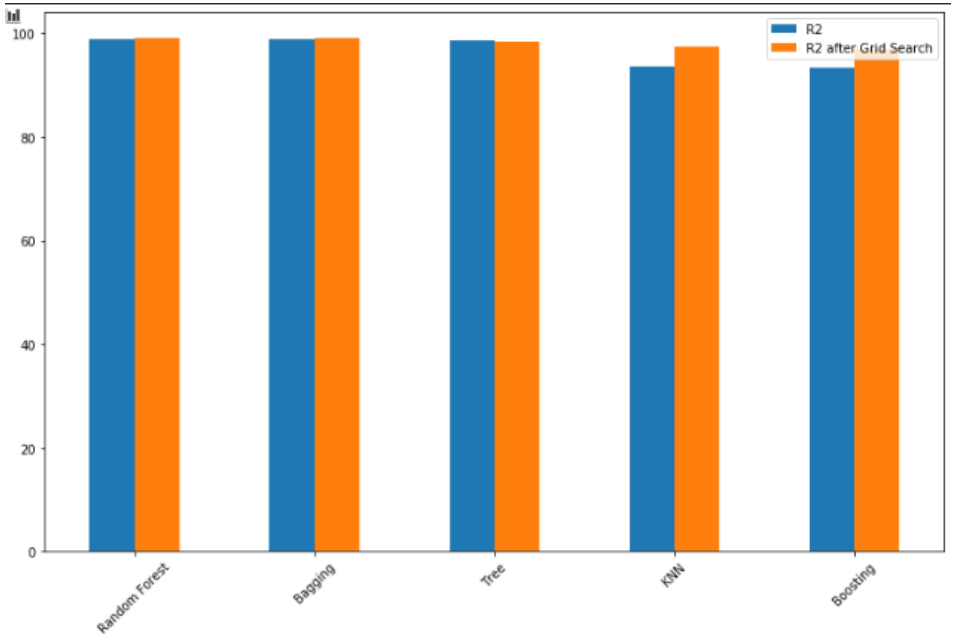
```
We get the following score : 0.9841656768776839 using these hyperparameters :
 BaggingRegressor(base_estimator=None, bootstrap=True, bootstrap_features=False,
                  max_features=0.925, max_samples=1.0, n_estimators=175,
                  n_jobs=None, oob_score=True, random_state=10, verbose=0,
                  warm_start=False)

~> 99.09 % of the data is explained (R2) and we have a MSE error of ~> 2.33.
```

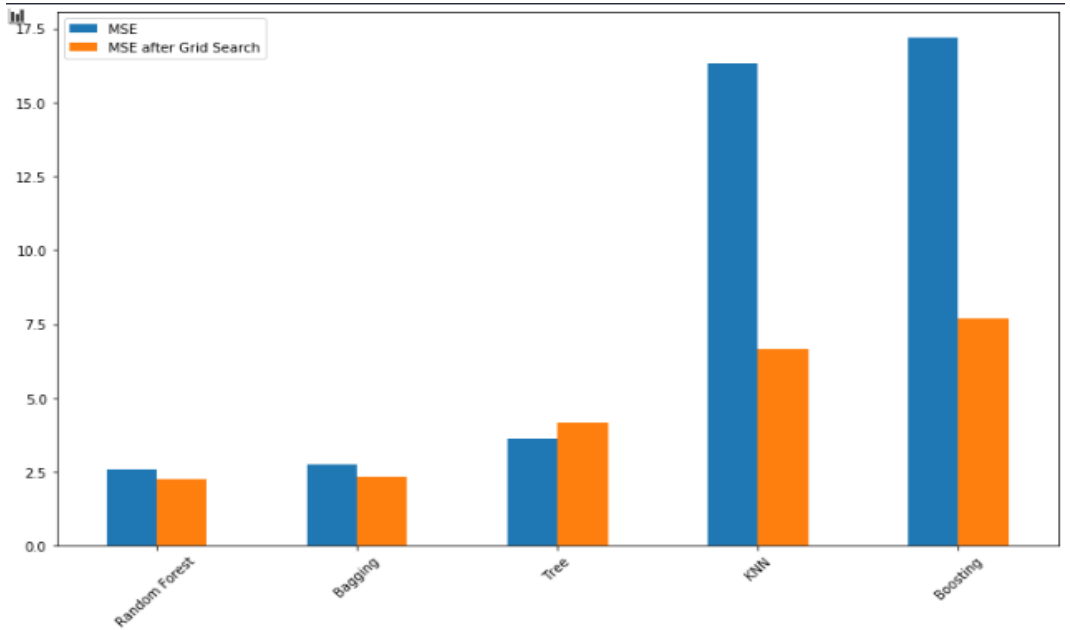
THE LAST PLOTTING PHASE: WHICH MODEL TO USE ?

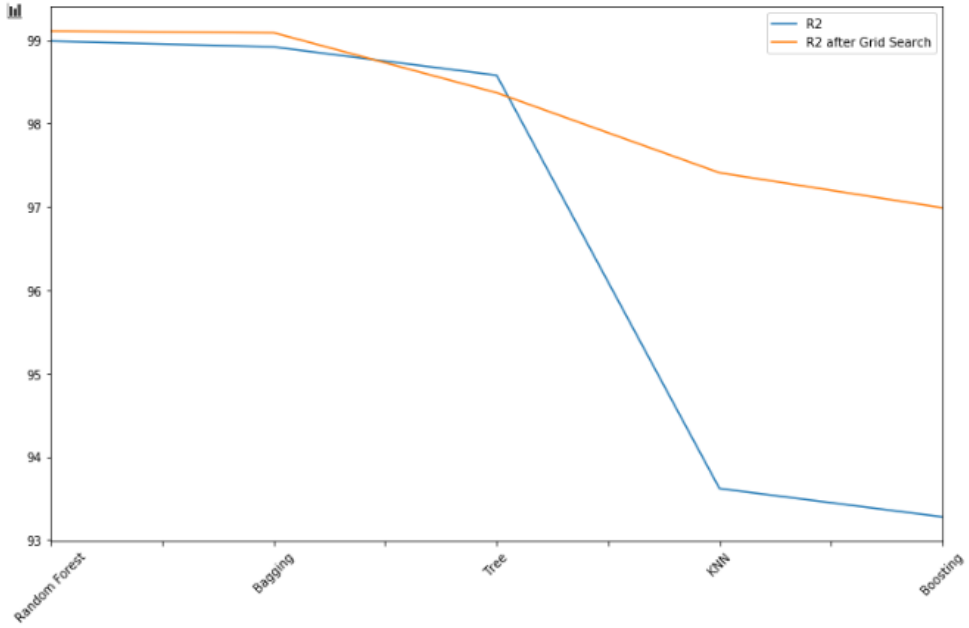
- We created the same type of score dictionary to store the result of the optimised models after grid search, then we turned them to dataframes using the following code.

```
performances_opti = pd.DataFrame.from_dict(score_dict_opti).T
MSE_opti = pd.DataFrame.from_dict(mse_dict_opti).T

score_opti = pd.DataFrame.from_dict(score_dict_opti_train).T
score_opti.columns = ["Grid best score"]
score_opti = score_opti.sort_values(by='Grid best score',ascending=False)

compare_R2 = pd.concat([performances.drop(['Boosting 0.001','Logistic Regression','SVR linear','SVR rbf']),performances_opti],axis = 1)
compare_R2.columns = ["R2","R2 after Grid Search"]
compare_R2 = compare_R2.sort_values(by='R2 after Grid Search',ascending=False)


compare_MSE = pd.concat([MSE.drop(['Boosting 0.001','Logistic Regression','SVR linear','SVR rbf']),MSE_opti],axis = 1)
compare_MSE.columns = ["MSE","MSE after Grid Search"]
compare_MSE = compare_MSE.sort_values(by='MSE after Grid Search',ascending=True)

final_dataframe = pd.concat([compare_R2,compare_MSE],axis=1)
final_dataframe = final_dataframe.sort_values(by='R2 after Grid Search',ascending=False)
```


Grid best score
98.5
98.0
97.5
97.0
96.5
96.0
95.5
Bagging
Random Forest
Boosting
Tree
KNN


R2
R2 after Grid Search
100
80
60
40
20
0
Random Forest
Bagging
Tree
KNN
Boosting


MSE
MSE after Grid Search
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
Random Forest
Bagging
Tree
KNN
Boosting


R2
R2 after Grid Search
99
98
97
96
95
94
93
Random Forest
Bagging
Tree
KNN
Boosting

- The top left graph shows the best returned score, while the bottom left graph compares the MSE before and after grid search for each model. As for the rest of the graphs they show the evolution of R^2 before and after grid search for each model (we added the line version of R^2 because it's easier to see a small-scale evolution, for Random Forest for example).
- By looking at our plots, we can see that we had some drastic improvements overall : Boosting and KNN saw a really high improvement compared to their default hyperparameters: if we look at the MSE graph, we can see how much MSE got decreased by for these models.
- However, for the regression decision tree, it seems as if we started overfitting the data since its grid search score improved but the error got higher and the R2 got lower.
- **Conclusion** :

  Random Forest seems to have a very slightly higher R2 and slightly lower MSE. However, Bagging has a better grid best score. So, since the results are close (and the methods are related after all, bagging is a special case of random forest), we will choose Bagging since it has a higher grid score (on unseen data thanks to oob_score = True), so it is less likely to overfit the data.

  Now, usually a score > 99% would raise some red flags since the model could have overfit the data and memorized the answers.

  Usually, we want an accuracy that is high but not very close to 100%. But we think that this is completely normal and not due to overfitting just because of the context of our dataset :

  => Generally, when it comes to transcoding, there's not really any magic behind it. Meaning, that for the same settings we will most likely always get the same / very close transcoding time. This differs from predicting the price of a house for example (Boston dataset), our dataset here seems logical.

SAVING THE MODEL

SAVING MODELS USING PICKLE

The following model is the final and best model overall, so we will pickle it using pickle.dump()

```
import pickle

final_model = BaggingRegressor(base_estimator=None, bootstrap=True, bootstrap_features=False,
                  max_features=0.925, max_samples=1.0, n_estimators=175,
                  n_jobs=None, oob_score=True, random_state=10, verbose=0,
                  warm_start=False)

model_to_pickle = final_model.fit(X_train,y_train)
```

However, testing the data we're going to input in our API needs to be scaled and encoded like we did for the training and testing datasets.
Therefore, we need to pickle the scaler and one hot encoder since they have been fitted on our training set.

BUILDING THE API

BUILDING THE API USING FLASK

- The API part will be split into two python files:
  - app.py: containing the API and its routings.
  - request.py: containing the request query.
- The manual request query should be done in three steps:

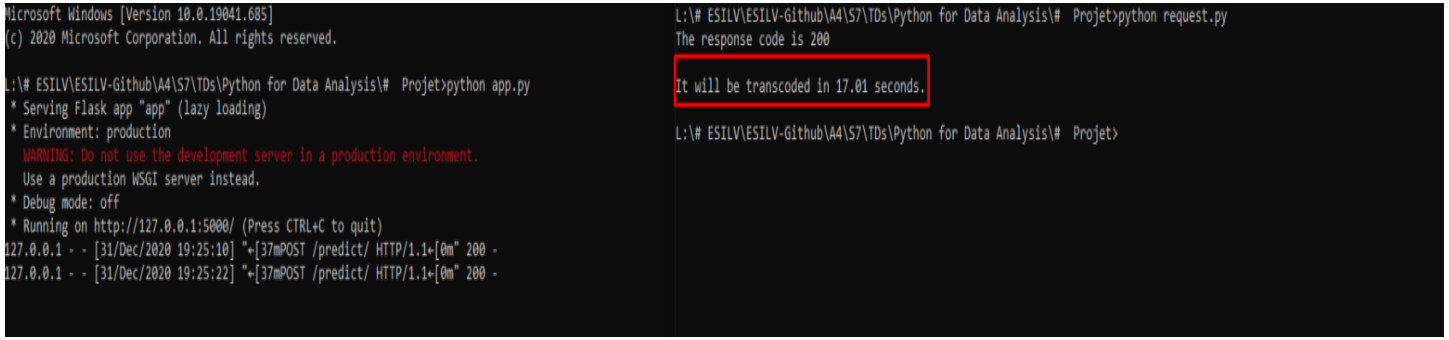
```
request_test = [180,"flv",480,360,550000,30,250,5000,150,5400,2000,85000,1000,88000,"mpeg4",56000,60,3840,2160,58528]

jsonyfied_data = json.dumps(request_test)

headers = {
        'content-type': 'application/json',
        'Accept-Charset': 'UTF-8'
        }

response = requests.post(url, data=jsonyfied_data, headers=headers)
```

The sent test data should be filled in the following order :

"duration","codec","width","height","bitrate","framerate","i","p","b","frames","i_size","p_size","b_size","size","o_codec","o_bitrate","o_framerate","o_width","o_height","umem »

The example in the image was an example of transforming a 3 min video from 480p in flv to 4K in mpeg4 while allocating 58.5 MB in umem.

```python
@app.route('/predict/', methods=['POST'])
def predict():

    data = request.get_json()

    # Scale the data

    df = pd.DataFrame(data).T

    X_test_num  = df
    X_test_num = X_test_num.drop([1,14],axis=1)

    X_test_cat = df
    X_test_cat = X_test_cat[[1,14]]

    scaled_col_test = scaler.transform(X_test_num)
    encoded_col_test = one_hot_encoder.transform(X_test_cat)

    scaled_data  = np.concatenate([scaled_col_test,encoded_col_test],axis=1)

    # Predict

    prediction = model.predict(scaled_data)

    return jsonify(result = round(float(prediction),3))
```



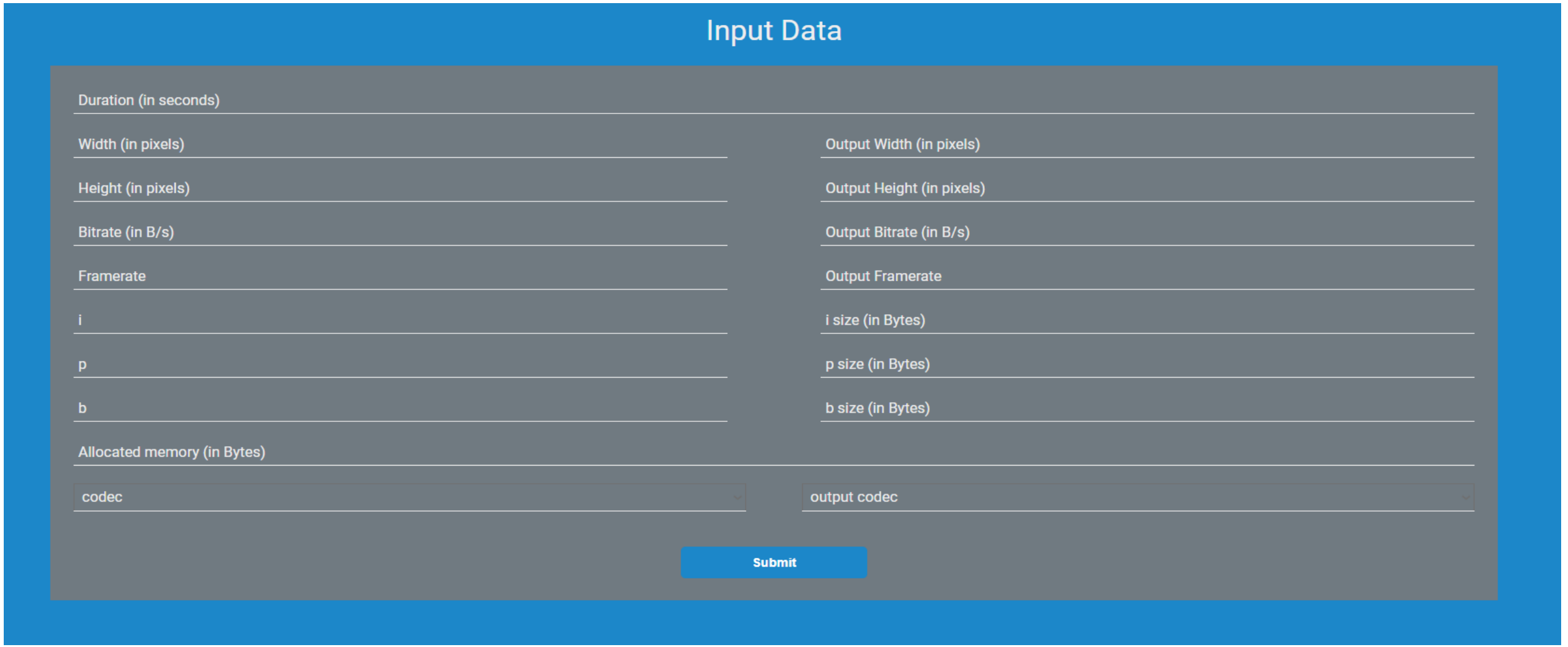
The following part in app.py handles the manual request queries, first by getting the data, and then scaling and encoding them before predicting the transcoding time, the model, the scaler and the encoder have been imported from their pickle files in the main of app.py

The console shows that after we run app.py, we can run the request.py in a separate console and it will return the prediction that have been sent to the server by request.py

Similarly to app.py this code will handle the prediction using the features that have been filled out in the online form, it will start by retrieving the data and returning the predicted of the transcoded time.

```
@app.route('/predict-site/',methods=['POST'])
def predictSite():

    duration = float(request.form['duration'])
    width = int(request.form['width'])
    o_width = int(request.form['o_width'])
    height = int(request.form['height'])
    o_height = int(request.form['o_height'])
    bitrate = int(request.form['bitrate'])
    o_bitrate = int(request.form['o_bitrate'])
    framerate = float(request.form['framerate'])
    o_framerate = float(request.form['o_framerate'])

    i = int(request.form['i'])
    p = int(request.form['p'])
    b = int(request.form['b'])

    i_size = int(request.form['i_size'])
    p_size = int(request.form['p_size'])
    b_size = int(request.form['b_size'])

    frames = i + p + b
    size = i_size + p_size + b_size

    umem = float(request.form['umem'])

    codec = request.form['codec']
    o_codec = request.form['o_codec']

    data = [duration,codec,width,height,bitrate,framerate,i,p,b,frames,i_size,p_size,b_size,size,o_codec,o_bitrate,o_framerate,o_width,o_height,umem]

    # Scale the data

    df = pd.DataFrame(data).T

    X_test_num  = df
    X_test_num = X_test_num.drop([1,14],axis=1)

    X_test_cat = df
    X_test_cat = X_test_cat[[1,14]]

    scaled_col_test = scaler.transform(X_test_num)
    encoded_col_test = one_hot_encoder.transform(X_test_cat)

    scaled_data  = np.concatenate([scaled_col_test,encoded_col_test],axis=1)

    # Predict

    prediction = model.predict(scaled_data)

    #return jsonify(result = round(float(prediction),3))
    return render_template("result.html",utime=round(float(prediction),3))
```

We took the html code of the following site: https://www.w3docs.com/tools/editor/5797, and we changed so it can fit our needs, since the goal of this project is python usage and not html.



And here is the result after testing another example: