Python for data analysis

Gomes Teixeira Filipe
DIA2

Dataset Analyse

Dataset

The dataset used during this project is :

Online Video Characteristics and Transcoding Time Dataset (https://archive.ics.uci.edu/ml/datasets/Online+Video+Characteristics+and+Transcoding+Time+Dataset)

The dataset is composed of 2 dataset :

- **Youtubes_videos.tsv** -> this dataset give us an insight on the data
- **Transcoding_mesurment.tsv** -> give us the data in order to train our models

Inside of the dataset

In the first dataset, which give us information on the youtube videos we have this elements :

- id : the id of video on youtube
- duration : the duration of the video (seconds)
- bitrate : the number of Kbits (audio + video) conveyed per seconds
- bitrate(video) : the number of Kbits (only video) conveyed per seconds
- height : the height of the video in pixels
- width : the width of the video in pixels
- frame rate : the number of image shown during one second
- frame rate(est.) : it's an estimation of the frame rate of the video
- codec : the computer program used for the compression and the decompression of a video in order to speed up the upload on the web
- category : the category (ex: Music, Gaming, ...) of the video on youtube
- url : the link of the video

In the second dataset who possess the transcoding characteristics of the youtube videos we have :

- id : the id of video on youtube
- framerate : the number of image shown during one second
- bitrate : the number of Kbits (audio + video) conveyed per seconds
- o_codec : output codec used for transcoding
- o_bitrate : output bitrate used for transcoding
- o_framerate : output framerate used for transcoding
- o_width : output width in pixel used for transcoding
- o_height : output height used in pixel for transcoding
- umem : total codec allocated memory for transcoding
- duration : the duration of the video (seconds)
- width : the width of the video in pixels
- height : the height of the video in pixels
- frames : the number of frames of the video.
- i_size : the total size in byte of i-frames in the video
- p_size : the total size in byte of p-frames in the video
- b_size : the total size in byte of b-frames in the video
- size : the total size in byte of the video (i_size + p_size + b_size)
- utime : total transcoding time for transcoding
- codec : the computer program used for the compression and the decompression of a video in order to speed up the upload on the web
- i : i correspond to the i-frames. The i-frames are frames examined independently from the others frames, you can usually find them interspersed with b-frames and p-frames in a compressed video.
- p : p correspond to the p-frames. The p-frames are frames who follow the i-frames. They only contain the data that have changed from the preceding i-frame.
- b : b correspond to the b-frames. The b-frames are frames who contains only the data that have changed from the preceding frame or are different from the data of the next frame.

Description of the datasets

When we analyse the insight dataset, we see that in average:

- The videos last 145 seconds
- The total bitrate is at 459 Kbits with 349 Kbits of video and 110 Kbits of audio
- 480*360 pixels, this correspond to an 4:3 aspect ratio resolution of 360p
- The videos posses 30 fps

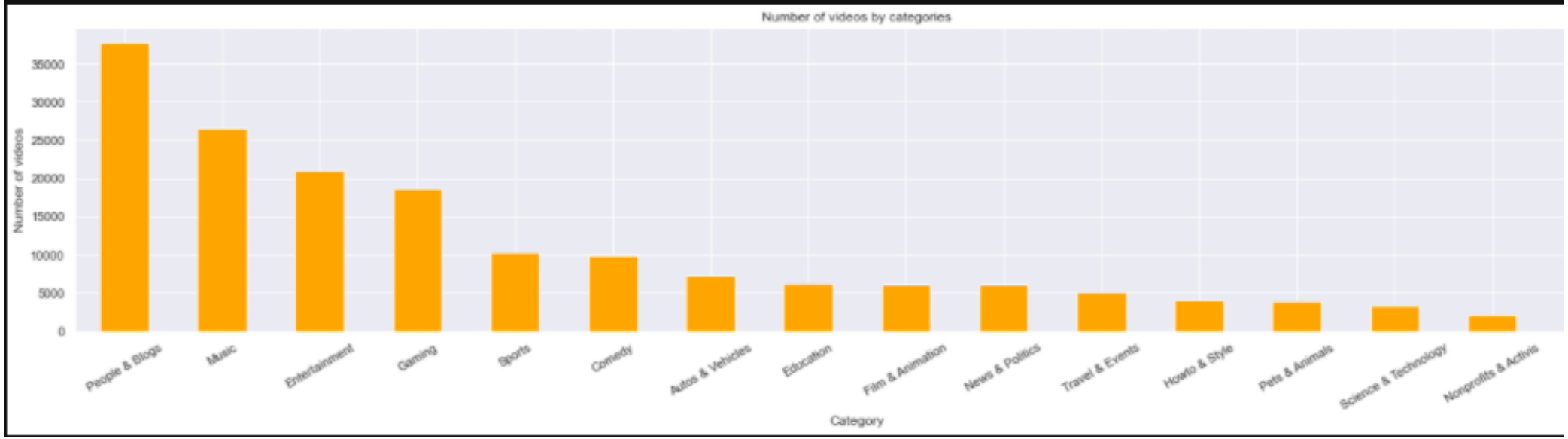
In the analyze of the transcoding dataset:

- The videos have in average 5628 frames composed of 80 i-frames, 5515 p-frames and 0 b-frames.
- When we compare the bitrate with the o_bitrate we observe that there is no dependency between them
- In average, the time in order to transcode a video is 4 seconds meanwhile the mean time is 10 seconds. This allow us to understand that the video have in average a good transcoding time

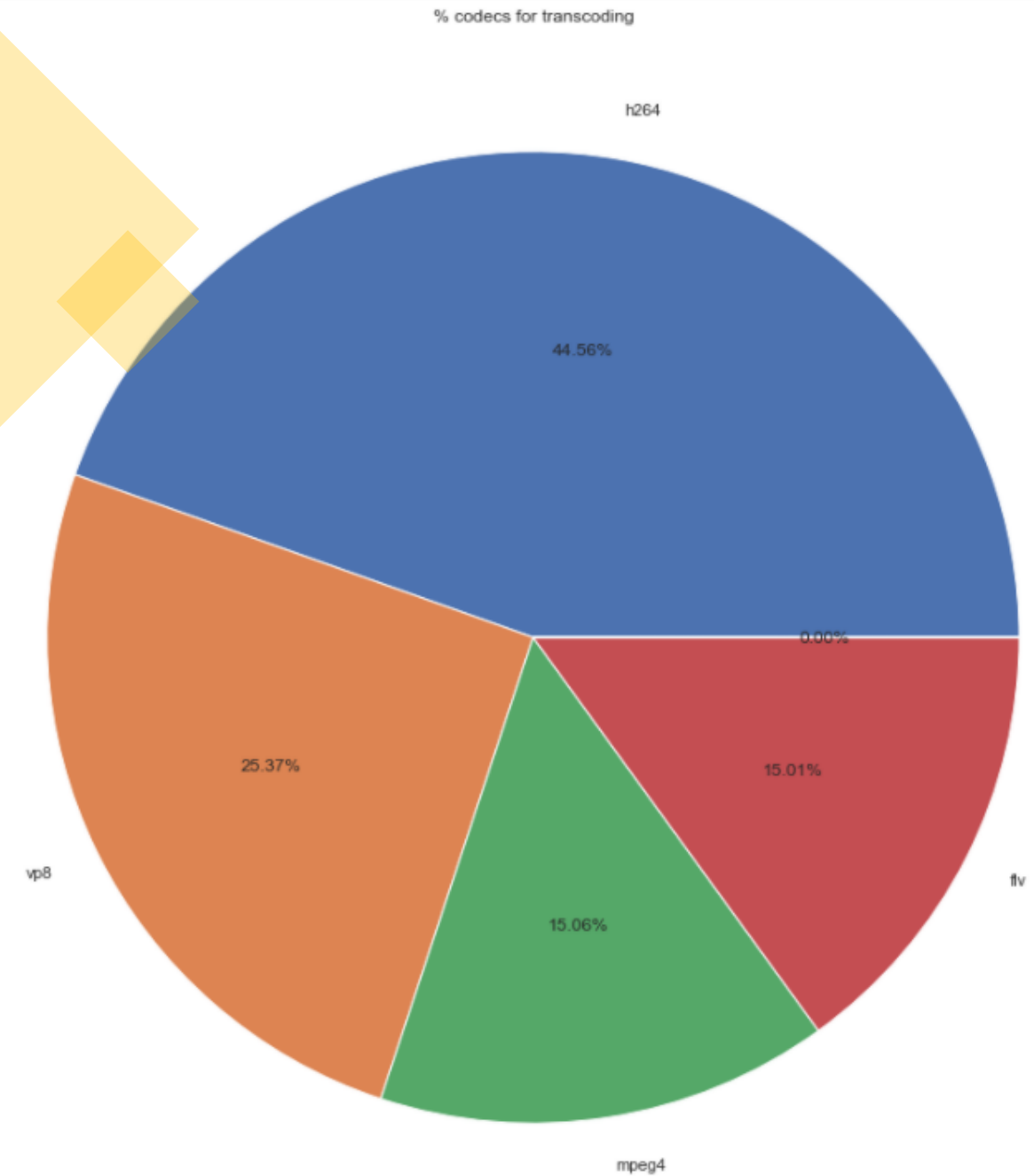
Graph analysis of the data

Display of the number of video for each category of video

```
insight["category"].value_counts().plot(kind='bar', figsize = (25,5),rot = 30, color = 'orange')
plt.xlabel("Category")
plt.ylabel('Number of videos')
plt.title("Number of videos by categories")
plt.show()
```



```
(insight["codec"].value_counts(), labels = ["h264", "vp8", "mpeg4", "flv", None], autopct='%1.2f%%')
le("% codecs for transcoding")
w()
```



When I was analyzing the dataset, I determined 2 type of categorical data : **category** and **codec**.

Since **codec** was the most interesting in order to determine the transcoding time of the videos, I did some graph on it in order to understand is influence.

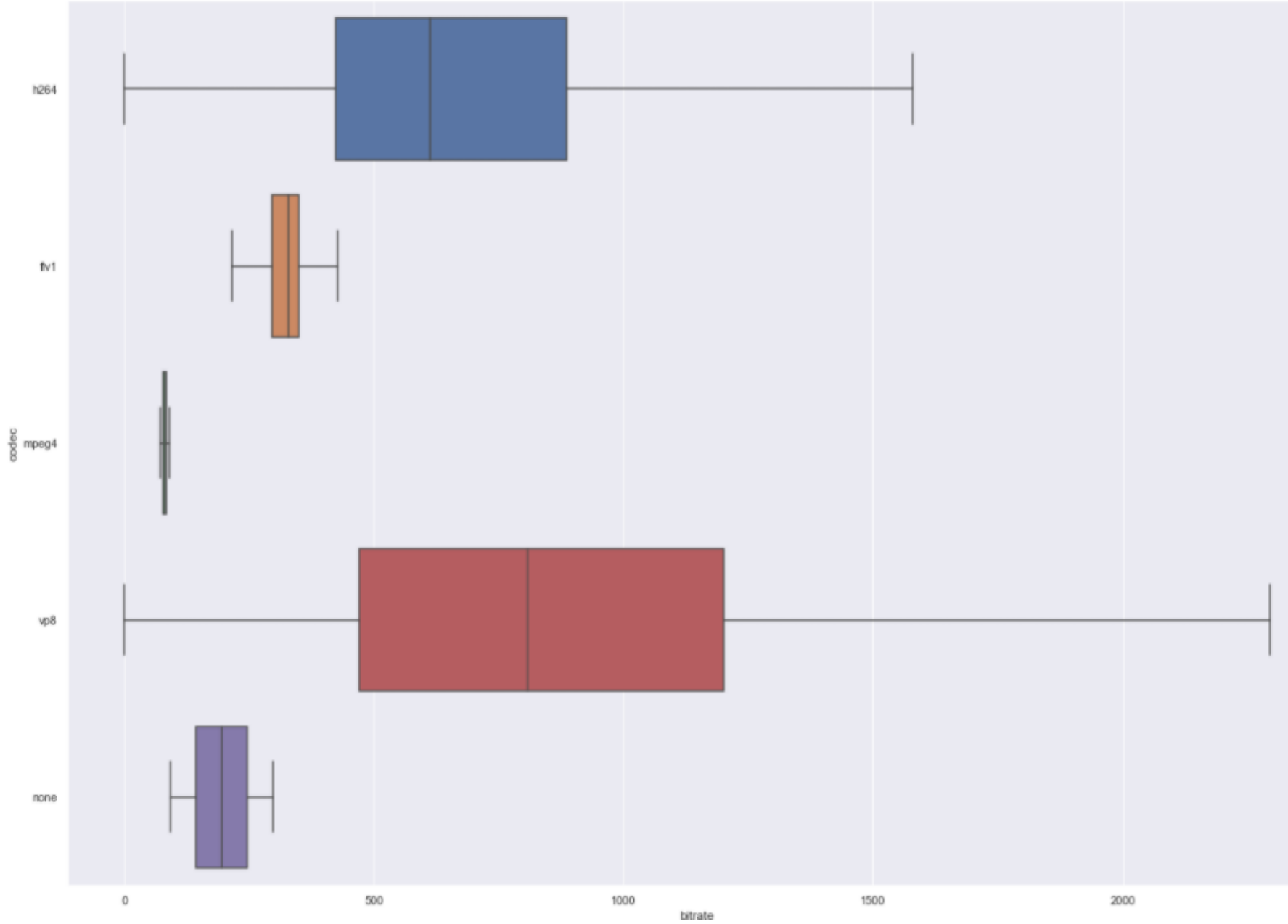
The pie graph at the left show % of use for each codecs. We observe that the most used codecs is the h264.

Now that we seen the usage for each codec, what is interesting to analyse is the bitrate of the video depending of their codec.

Has we see in the graph at the right, the bitrage change depending of the codec used for the transcoding.

So we possess a relation between them.

```
d_category = insight[["codec", "bitrate"]]
sns.set_theme(style="whitegrid")
sns.set(rc={'figure.figsize':(22, 15)})
ax = sns.boxplot(x="bitrate", y="codec", data=d_category, showfliers = False)
```



After having analysed the codec, I choose to realise a pairplot in order to see if some feature are dependant between them.

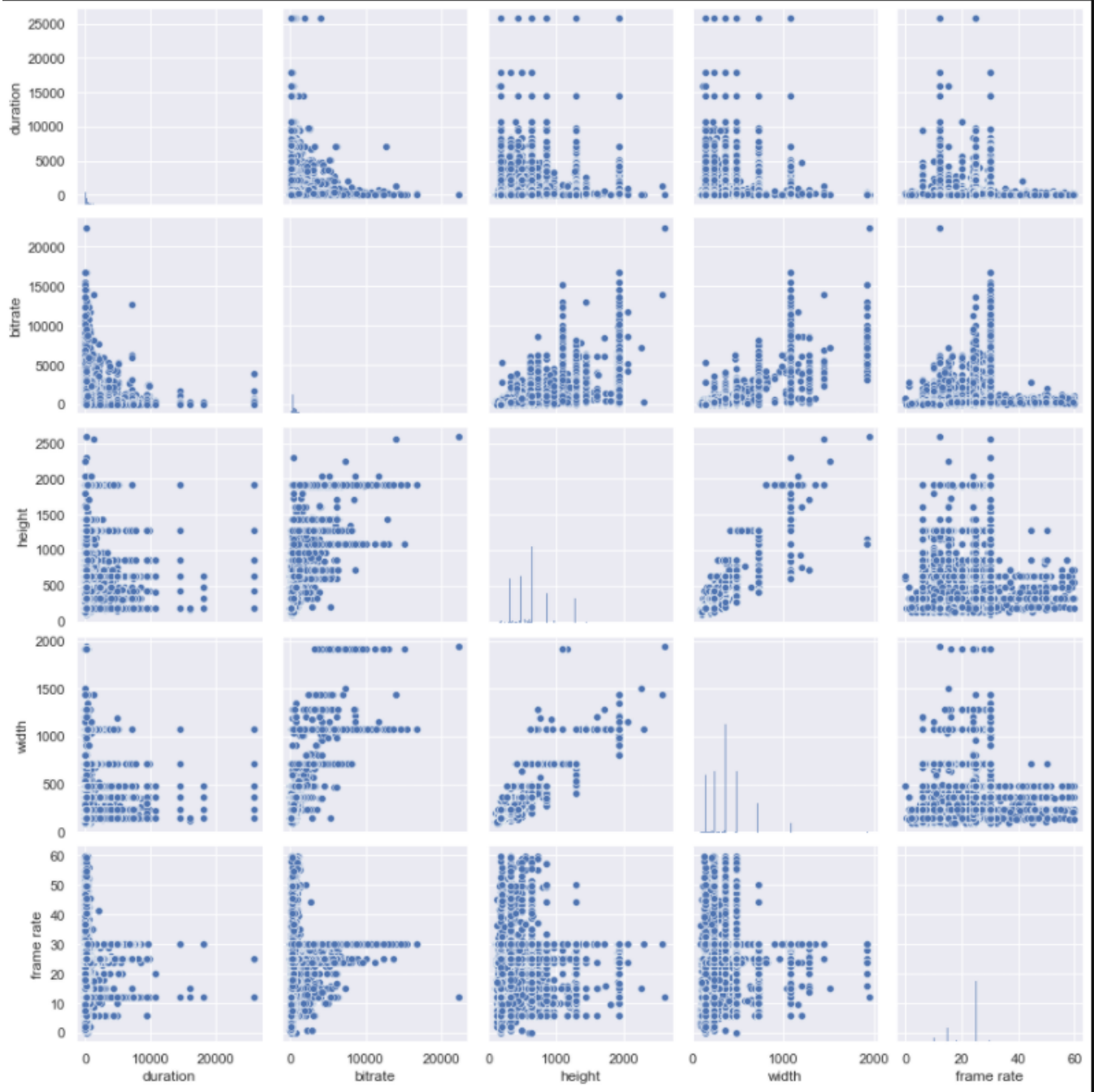
We observe a good dependence between height and width.

There is also a little dependence between bitrate and duration.

We can observe the increase of the bitrate when the height/width of the video raise.

```
sns.pairplot(insight[['duration', 'bitrate', 'height', 'width', 'frame rate']])
```

```
<seaborn.axisgrid.PairGrid at 0x263c8adc880>
```



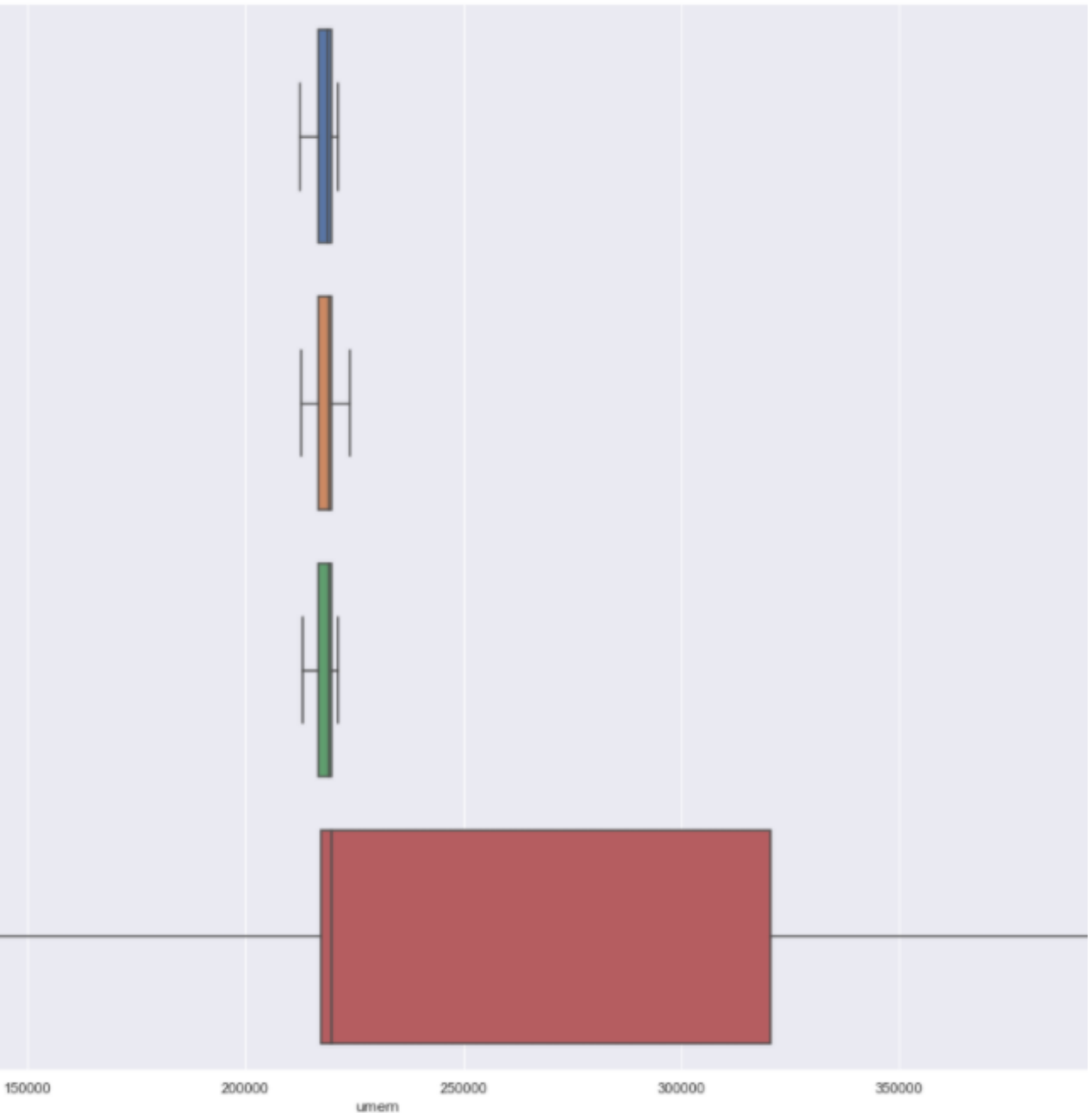
```
result = transcoding.corr()
result = result[['utime']].iloc[:len(result[['utime']])-1]
result = result[result['utime'] > 0.01].sort_values(by="utime")
result
```

	utime
i	0.018489
frames	0.033115
p	0.033201
i_size	0.064711
framerate	0.079336
size	0.097096
p_size	0.097644
o_framerate	0.104043
height	0.128479
width	0.129861
bitrate	0.155200
o_bitrate	0.155479
o_height	0.519649
o_width	0.523388
umem	0.663301

Next to that, I choose to look at the linear correlation between the features and utime, which give me the result that you can observe at the left.

We observe a good linear correlation of utime with o_bitrate, o_height, o_width and umem.

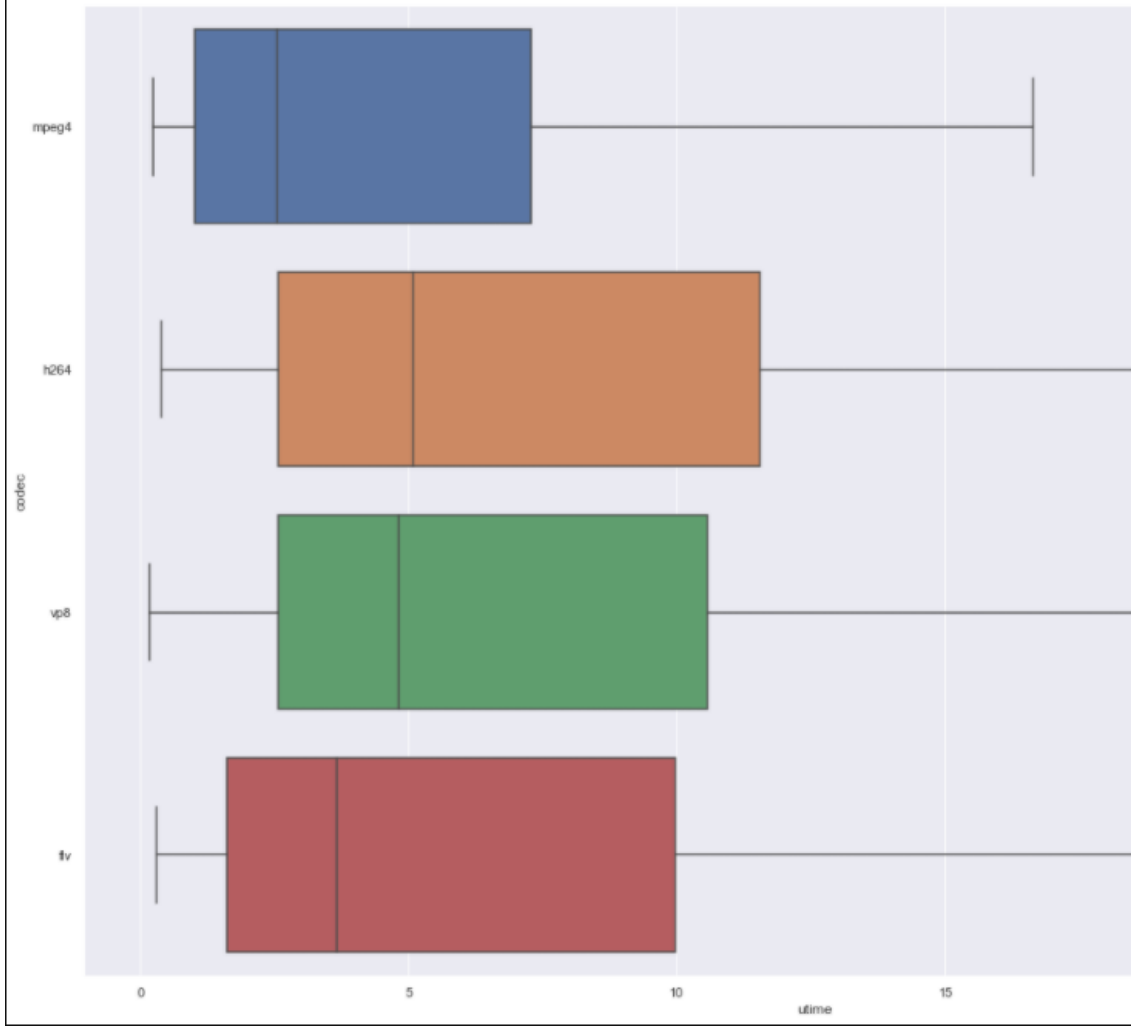
```
_category, showfliers = False)
```



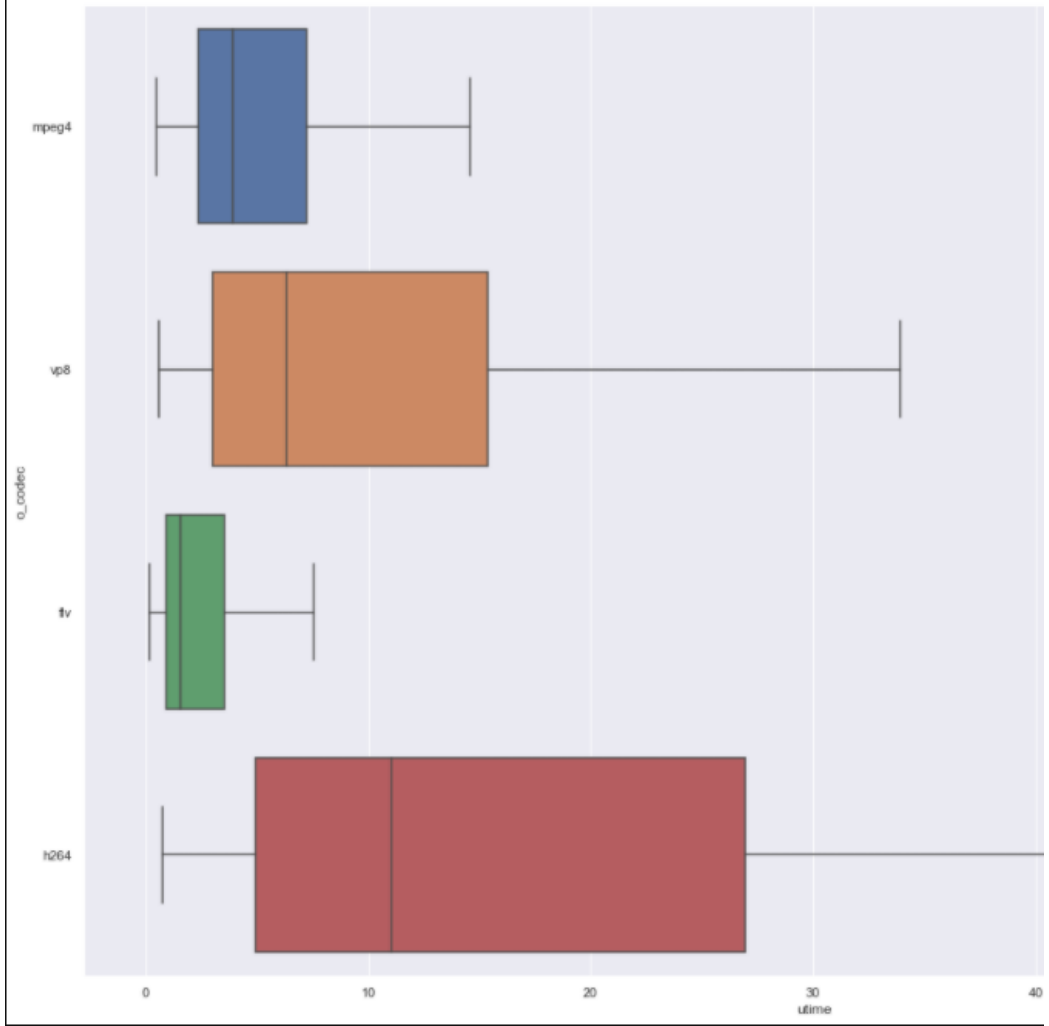
Since umem had the best linear correlation with utime, I wanted to compare the umem used by the o_codec. It’s appeared that the h264 codec was the only codec taking a lot of umem.

Since the codec is one of the most important feature, I tried to compare the influence of the codec on utime. It’s appear that depending of the codecs used for the transcoding, the utime could vary of a dozen of seconds.

```
d_category = transcoding[["codec", "utime"]]
sns.set_theme(style="whitegrid")
sns.set(rc={'figure.figsize':(22, 15)})
ax = sns.boxplot(x="utime", y="codec", data=d_category, showfliers = False)
```




```
d_category = transcoding[["o_codec", "utime"]]
sns.set_theme(style="whitegrid")
sns.set(rc={'figure.figsize':(22, 15)})
ax = sns.boxplot(x="utime", y="o_codec", data=d_category, showfliers = False)
```



Machine Learning

Modification of the data

In order to create the models with efficacity I did some modifications :

- I first deleted the useless features such has the id of the video, since the id of each video is unique, it would be useless.

- After that I encoded the values of the categorical features. Otherwise, I couldn’t use them with some models.

Preparation of the datasets

- In order to train with efficacity the models, I have set apart the utime then I splited the transcoding dataset into 3 datasets :

- 1 - train dataset : this will be the training dataset used for the fit of the models

- 2 - validation dataset : this dataset is the dataset used in order to test the prediction of the trained model and determine a score for each of them

- 3 - test dataset : this final dataset is a dataset put apart until the end. The utility of this dataset is to test the best model and obtain a score who will prove the precision of the model and if the model isn’t overfitted to the train dataset

Bayesian Ridge Regression

```
model_brr = BayesianRidge().fit(X_train, y_train)
score_brr = model_brr.score(X_valid, y_valid)
d_score = d_score.append({'Model': 'Bayesian Ridge Regression', 'Score': score_brr}, ignore_index=True)
score_brr
```

```
0.5978119641241004
```

Linear Regression

```
model_lr = LinearRegression().fit(X_train, y_train)
score_lr = model_lr.score(X_valid, y_valid)
d_score = d_score.append({'Model': 'Linear Regression', 'Score': score_lr}, ignore_index=True)
score_lr
```

```
0.5978045856677494
```

Decision Tree Regressor

```
model_dtr = DecisionTreeRegressor().fit(X_train, y_train)
score_dtr = model_dtr.score(X_valid, y_valid)
d_score = d_score.append({'Model': 'Decision Tree Regressor', 'Score': score_dtr}, ignore_index=True)
score_dtr
```

```
0.9617122664490906
```

Boosting

```
model_b = GradientBoostingRegressor(random_state=20).fit(X_train, y_train)
score_b = model_b.score(X_valid, y_valid)
d_score = d_score.append({'Model': 'Boosting', 'Score': score_b}, ignore_index=True)
score_b
```

```
0.9289097007021189
```

K Neighbors Regressor

```
model_knr = KNeighborsRegressor().fit(X_train, y_train)
score_knr = model_knr.score(X_valid, y_valid)
d_score = d_score.append({'Model': 'K Neighbors Regressor', 'Score': score_knr}, ignore_index=True)
score_knr
```

```
0.7075704266096103
```

Random Forest Regressor

```
model_rfr = RandomForestRegressor(max_depth=2, random_state=0).fit(X_train, y_train)
score_rfr = model_rfr.score(X_valid, y_valid)
d_score = d_score.append({'Model': 'Random Forest Regressor', 'Score': score_rfr}, ignore_index=True)
score_rfr
```

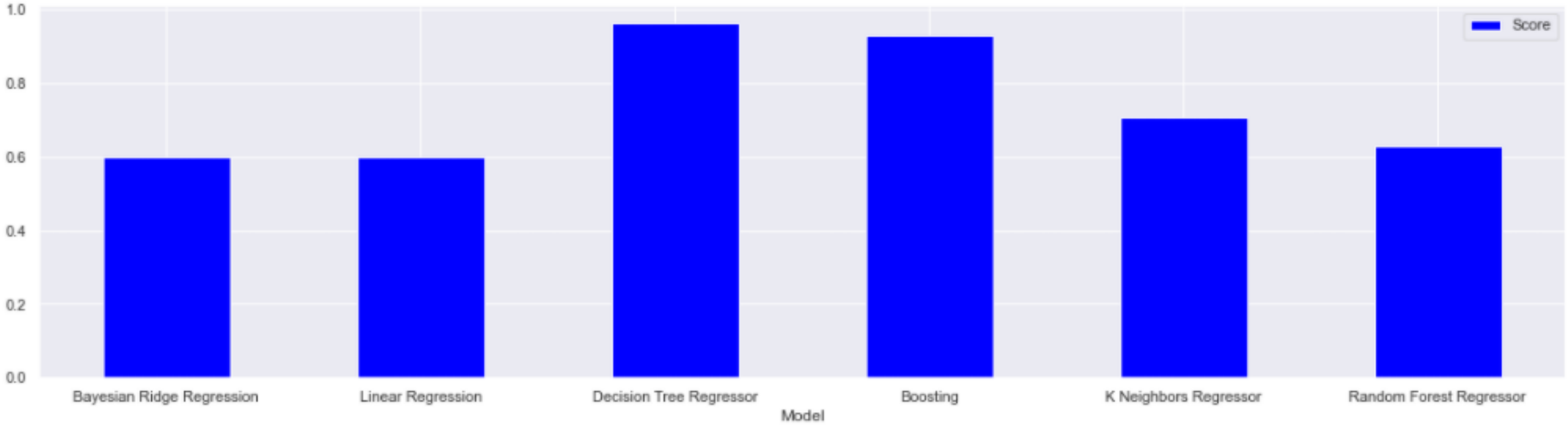
```
0.6304438266639001
```

Result of the different models

	Model	Score
0	Bayesian Ridge Regression	0.597812
1	Linear Regression	0.597805
2	Decision Tree Regressor	0.961712
3	Boosting	0.928910
4	K Neighbors Regressor	0.707570
5	Random Forest Regressor	0.630444

```
d_score.plot(kind='bar',x="Model", y="Score", figsize = (20,5), rot=0, color = 'blue')
```

```
<AxesSubplot:xlabel='Model'>
```



Optimization of the hyperparameters

Now that we possess our best model (The Decision Tree Regression), we will optimize his hyperparameters in order to improve our accuracy.

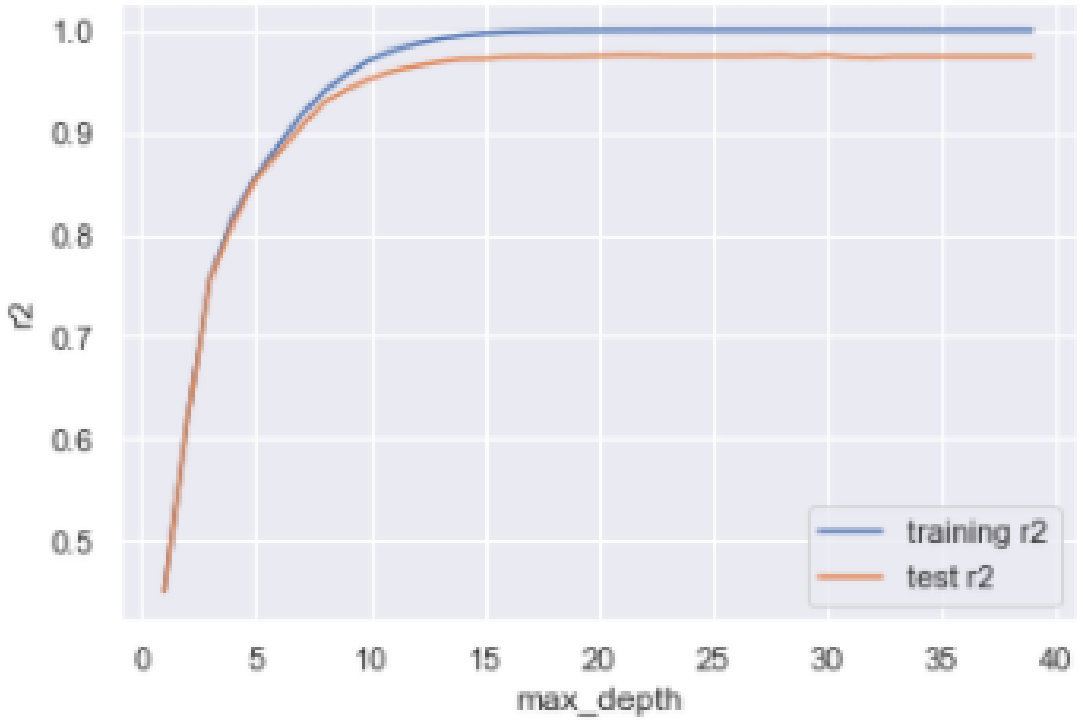
In order to realize this optimization, we will analyze a gap of the best values for the hyperparameters.

Once we possess the gap, we will do a cross validation with all the parameters between their gap in order to get the best combination of value for our model.

```
parameters = {'max_depth': range(1, 40)}

model_dtr_depth = DecisionTreeRegressor(criterion = "mse", random_state = 100)

tree_depth = GridSearchCV(model_dtr_depth, parameters, cv=n_folds, return_train_score=True, scoring="r2")
tree_depth.fit(X, y)
scores_depth = pd.DataFrame(tree_depth.cv_results_)
```

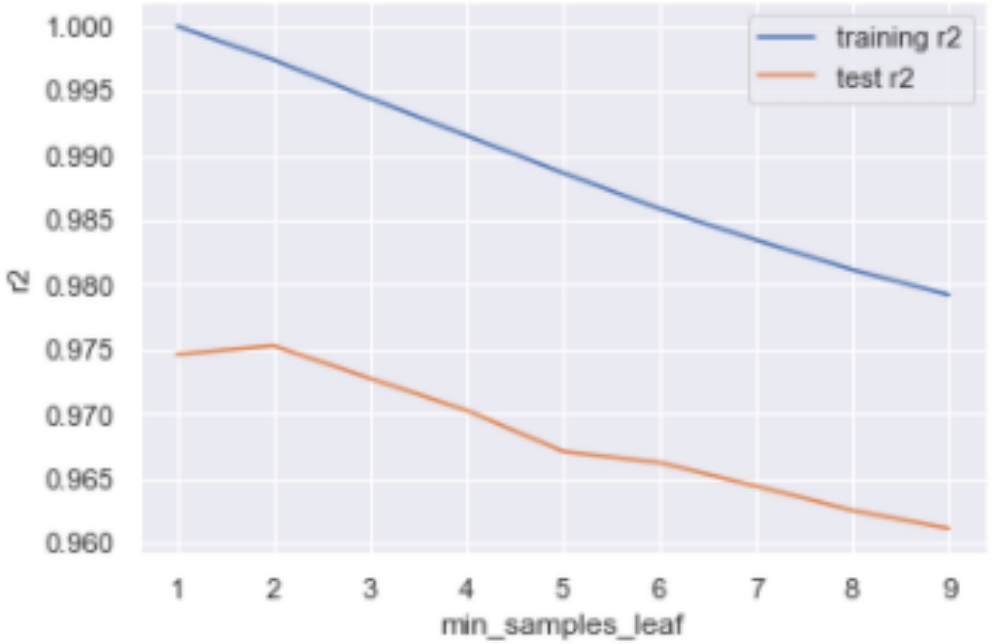



```
plt.figure()
plt.plot("param_max_depth", "mean_train_score", label="training r2", data=scores_depth)
plt.plot("param_max_depth", "mean_test_score", label="test r2", data=scores_depth)
plt.xlabel("max_depth")
plt.ylabel("r2")
plt.legend()
plt.show()
```

```
parameters = {'min_samples_leaf': range(1, 10, 1)}

model_dtr_sl = DecisionTreeRegressor(criterion = "mse", random_state = 100)

tree_sl = GridSearchCV(model_dtr_sl, parameters, cv=n_folds, return_train_score=True, scoring="r2")
tree_sl.fit(X, y)
scores_sl = pd.DataFrame(tree_sl.cv_results_)
```

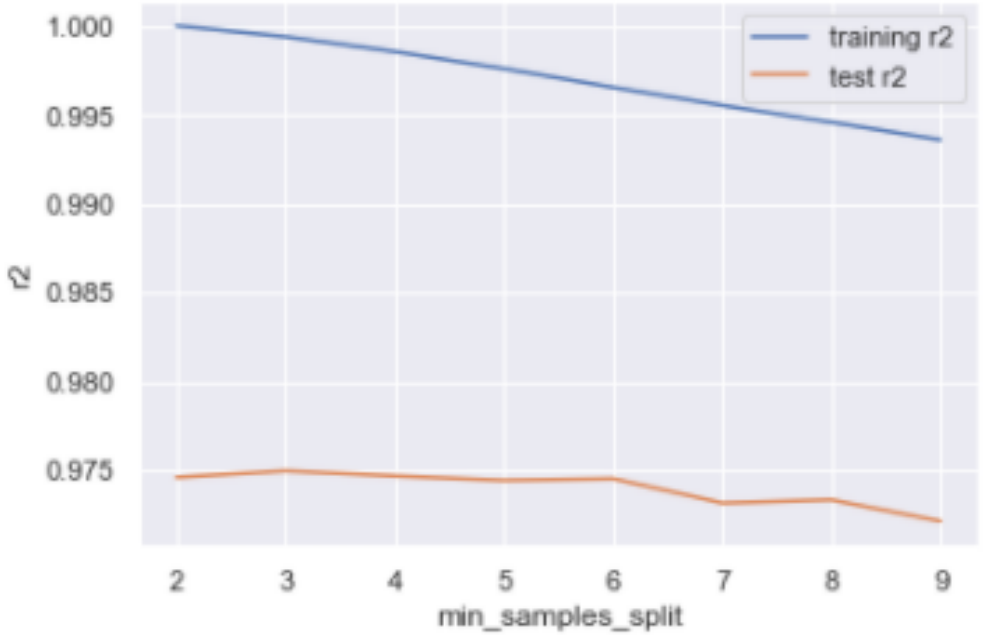



```
plt.figure()
plt.plot("param_min_samples_leaf", "mean_train_score", label="training r2", data=scores_sl)
plt.plot("param_min_samples_leaf", "mean_test_score", label="test r2", data=scores_sl)
plt.xlabel("min_samples_leaf")
plt.ylabel("r2")
plt.legend()
plt.show()
```

```
parameters = {'min_samples_split': range(2, 10, 1)}

model_dtr_ss = DecisionTreeRegressor(criterion = "mse", random_state = 100)

tree_ss = GridSearchCV(model_dtr_ss, parameters, cv=n_folds, return_train_score=True, scoring="r2")
tree_ss.fit(X, y)
scores_ss = pd.DataFrame(tree_ss.cv_results_)
```




```
plt.figure()
plt.plot("param_min_samples_split", "mean_train_score", label="training r2", data=scores_ss)
plt.plot("param_min_samples_split", "mean_test_score", label="test r2", data=scores_ss)
plt.xlabel("min_samples_split")
plt.ylabel("r2")
plt.legend()
plt.show()
```

```
param_grid = {
    'max_depth': range(25, 35, 1),
    'min_samples_leaf': range(1, 3, 1),
    'min_samples_split': range(2, 4, 1),
    'criterion': ["mse", "mae"]
}

model_dtr = DecisionTreeRegressor()
grid_cv_dtr = GridSearchCV(model_dtr, param_grid, cv=3, verbose=2)
grid_cv_dtr.fit(X, y)
```

```
GridSearchCV(cv=3, estimator=DecisionTreeRegressor(),
             param_grid={'criterion': ['mse', 'mae'],
                         'max_depth': range(25, 35),
                         'min_samples_leaf': range(1, 3),
                         'min_samples_split': range(2, 4)})
```

```
print(f"R-Squared : {grid_cv_dtr.best_score_}")
print(f"Best Hyperparameters : \n{grid_cv_dtr.best_params_}")
```

```
R-Squared : 0.9709938398498 35
Best Hyperparameters :
{'criterion': 'mse', 'max_depth': 26, 'min_samples_leaf': 1, 'min_samples_split': 2}
```

Now that we posses the best hyperparameters, we will compare the model with the best hyperparameters and the model without the hyperparameters

```
model_dtr_best_hp = DecisionTreeRegressor(criterion='mse', max_depth=26, min_samples_leaf=1, min_samples_split=2).fit(X_train, y_train)
model_dtr = DecisionTreeRegressor().fit(X_train, y_train)
score_final_test_best_hp = model_dtr_best_hp.score(X_test, y_test)
score_final_test = model_dtr.score(X_test, y_test)
print(f"best hyperparameters : {score_final_test_best_hp}\nclassic : {score_final_test}")
```

```
best hyperparameters : 0.9671445698582063
classic : 0.9491373784953243
```

At the end we won 0.018 of precision which is a lot