Prédiction du temps de transcodage des vidéos

Python for Data Analysis

Marwan Larroussi et Anatole Lapelerie, DIA 3

Sommaire

Introduction au transcodage

En vidéo, le transcodage est le fait de changer le format de codage d'un média pour le comprimer ou l'encapsuler dans un fichier, ou pour transporter un signal analogique ou numérique. Bien souvent, la transformation comporte des pertes d'information.

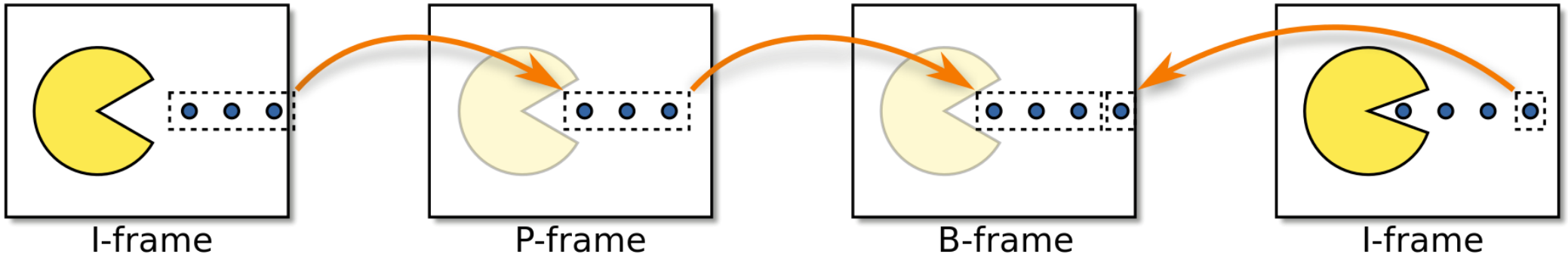
Une heure de film de DVD au format PAL (25 images de 720×557 px par seconde) contient un total de 104 Go d’information d’image ! La compression est donc essentielle pour stocker le fichier dans un DVD de 9 Go de capacité.

Les supports de transcodage utilisent un codec (*coder – encoder*), un langage particulier pour compresser la vidéo.

Il existe plusieurs techniques pour compresser une vidéo :

- Le sous échantillonnage de la chrominance
- La compression spatiale
- La compression temporelle

C'est lors de l'utilisation de cette dernière technique que sont créées les *i-frames*, *p-frames* et *b-frames*.



Il existe de même plusieurs codecs, ayant des performances, des taux de compression et une rapidité de décompression tous différents. On citera les codecs suivants : H.264, H.265, VP9, FLV1, MPEG-4.

A propos du dataset

Les données utilisées pour ce projet sont issues de deux datasets :

- `youtube_videos.tsv`, qui contient des métadonnées de vidéos YouTube ;
- `transcoding_mesurment.tsv`, qui contient des données issues de transcodages entre différents codecs à partir des vidéos du précédent fichier et menés sur une machine Intel i7-3720QM CPU grâce au logiciel FFmpeg 4.

Ces données sont tirées du *Machine Learning Repository* de l'UCI et sont regroupées sous la référence *Online Video Characteristics and Transcoding Time Dataset Data Set*.

Le dataset `youtube_videos.tsv` contient les paramètres suivants :

- **id**, **url** : informations identifiant la vidéo ;
- **category** : catégorie de la vidéo ;
- **codec** : codec de la vidéo ;
- **duration** : durée de la vidéo ;
- **bitrate**, **bitrate(video)** : débit en bit/s et débit de la vidéo en bit/s ;
- **height**, **weight** : résolution de l'image (hauteur et largeur) ;
- **framerate**, **framerate(est.)** : débit d'image par seconde.

En plus des données présentées pour le fichier précédent, le dataset `transcoding_mesurment.tsv` contient les paramètres suivants :

- **i**, **p**, **b**, : nombre total de *i-frames*, *p-frames* et *b-frames* ;
- **frames** : nombre total d'image dans la vidéo ;
- **i_size**, **p_size**, **b_size** : taille totale des *i-frames*, *p-frames* et *b-frames* ;
- **size** : taille totale du fichier vidéo;
- **o_bitrate**, **o_framerate**, **o_width**, **o_height** : bitrate, framerate, largeur et hauteur de sortie de transcodage ;
- **utime**, **umem** : temps total et mémoire totale utilisée pour le transcodage.

On affiche les statistiques de répartition des paramètres numériques des datasets YouTube et de transcodage.

youtube_videos.tsv

	count	mean	std	min	25%	50%	75%	max
duration	168286.0	271.654184	552.881871	1.0	55.00	145.00	289.00	25845.00
bitrate	168286.0	730.621490	919.154730	0.0	289.00	459.00	826.00	22421.00
bitrate(video)	168286.0	624.363025	860.955654	0.0	231.00	349.00	640.00	22229.00
height	168286.0	561.018706	359.071569	100.0	320.00	480.00	640.00	2592.00
width	168286.0	368.399701	201.274180	88.0	240.00	360.00	480.00	1944.00
frame rate	168286.0	24.564592	7.396615	0.0	23.98	29.92	29.97	59.94
frame rate(est.)	168286.0	19.884441	11.435070	0.0	12.00	25.00	29.97	30.02

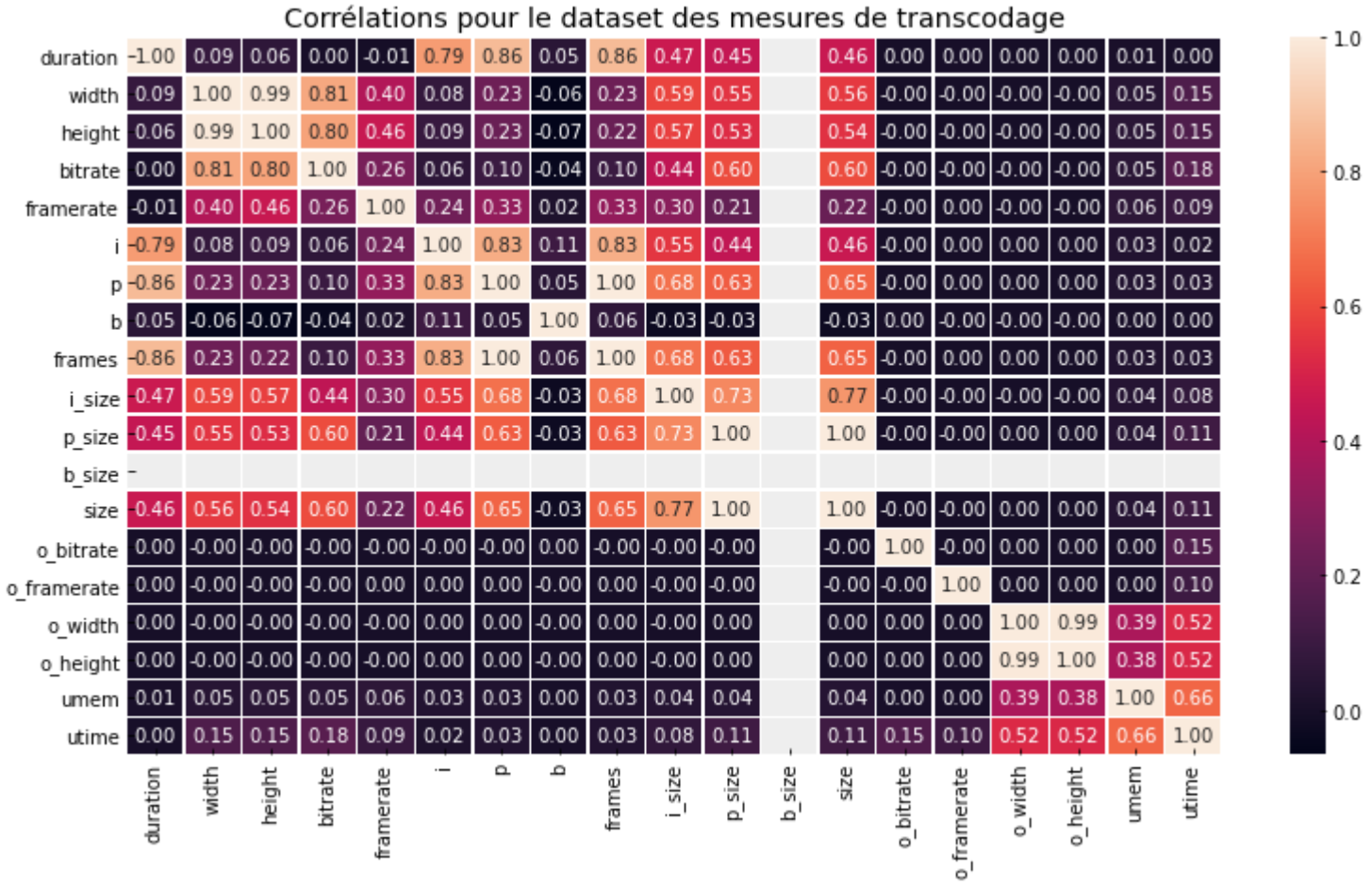
transcoding_mesurment.tsv

	count	mean	std	min	25%	50%	75%	max
duration	68784.0	2.864139e+02	2.872576e+02	31.080000	106.765	2.391417e+02	3.793200e+02	2.584409e+04
width	68784.0	6.249342e+02	4.631691e+02	176.000000	320.000	4.800000e+02	6.400000e+02	1.920000e+03
height	68784.0	4.125722e+02	2.406155e+02	144.000000	240.000	3.600000e+02	4.800000e+02	1.080000e+03
bitrate	68784.0	6.937015e+05	1.095628e+06	8384.000000	134334.000	2.911500e+05	6.529670e+05	7.628466e+06
framerate	68784.0	2.324132e+01	7.224848e+00	5.705752	15.000	2.502174e+01	2.900000e+01	4.800000e+01
i	68784.0	1.008683e+02	8.476479e+01	7.000000	39.000	8.000000e+01	1.380000e+02	5.170000e+03
p	68784.0	6.531692e+03	6.075872e+03	175.000000	2374.000	5.515000e+03	9.155000e+03	3.049590e+05
b	68784.0	9.147854e+00	9.251618e+01	0.000000	0.000	0.000000e+00	0.000000e+00	9.407000e+03
frames	68784.0	6.641708e+03	6.153342e+03	192.000000	2417.000	5.628000e+03	9.232000e+03	3.101290e+05
i_size	68784.0	2.838987e+06	4.325137e+06	11648.000000	393395.000	9.458650e+05	3.392479e+06	9.082855e+07
p_size	68784.0	2.218057e+07	5.097306e+07	33845.000000	1851539.000	6.166260e+06	1.515506e+07	7.689970e+08
b_size	68784.0	0.000000e+00	0.000000e+00	0.000000	0.000	0.000000e+00	0.000000e+00	0.000000e+00
size	68784.0	2.502294e+07	5.414402e+07	191879.000000	2258222.000	7.881069e+06	1.977335e+07	8.067111e+08
o_bitrate	68784.0	1.395036e+06	1.749352e+06	56000.000000	109000.000	5.390000e+05	3.000000e+06	5.000000e+06
o_framerate	68784.0	2.119086e+01	6.668703e+00	12.000000	15.000	2.400000e+01	2.500000e+01	2.997000e+01
o_width	68784.0	8.023364e+02	6.099598e+02	176.000000	320.000	4.800000e+02	1.280000e+03	1.920000e+03
o_height	68784.0	5.038255e+02	3.159704e+02	144.000000	240.000	3.600000e+02	7.200000e+02	1.080000e+03
umem	68784.0	2.282247e+05	9.743088e+04	22508.000000	216820.000	2.194800e+05	2.196560e+05	7.118240e+05
utime	68784.0	9.996355e+00	1.610743e+01	0.184000	2.096	4.408000e+00	1.043300e+01	2.245740e+02

On remarque que, comme les vidéos du dataset de transcodage sont tirées du dataset des vidéos YouTube, il est possible de réaliser une jointure de ces deux tables en une seule à partir de l'identifiant de la vidéo.

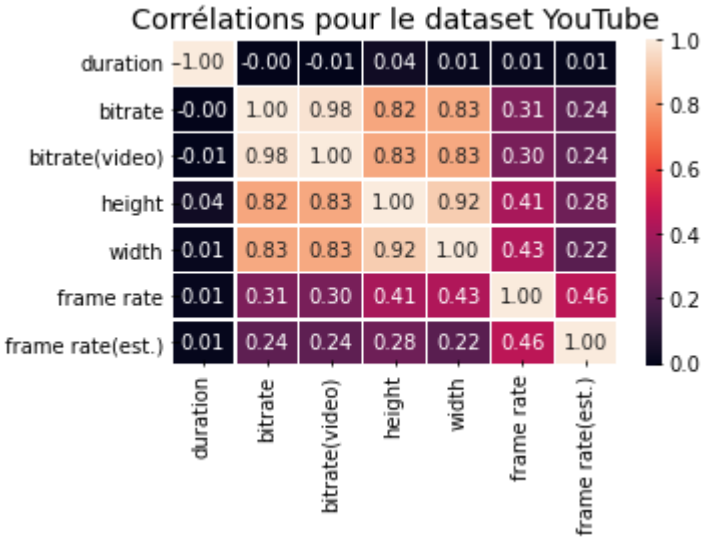
Durant tout le projet, on utilise Pandas pour traiter les données. On choisira donc ici des DataFrame pour manipuler nos tables.

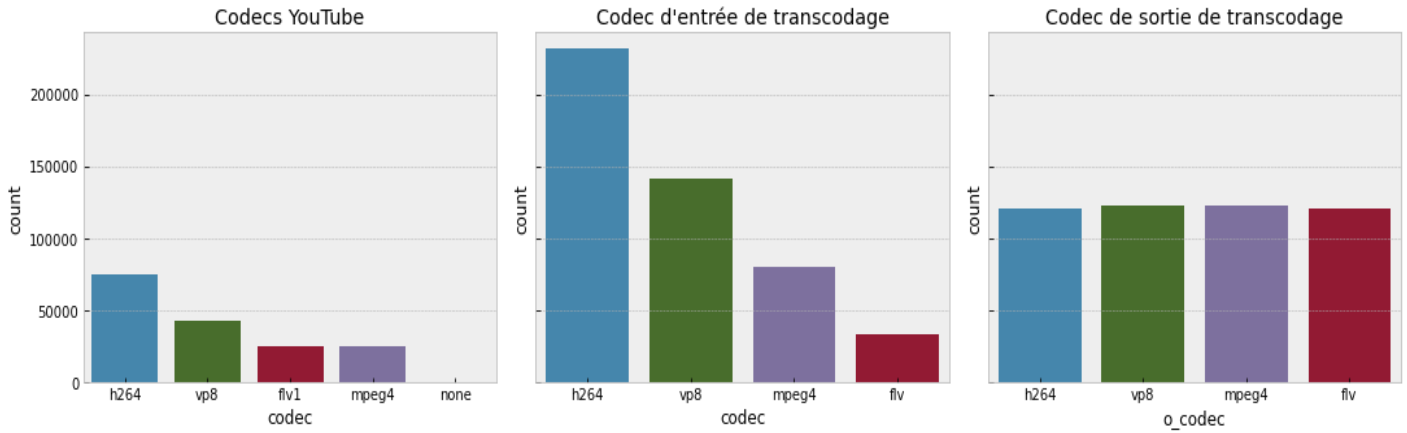
Data Vizualisation

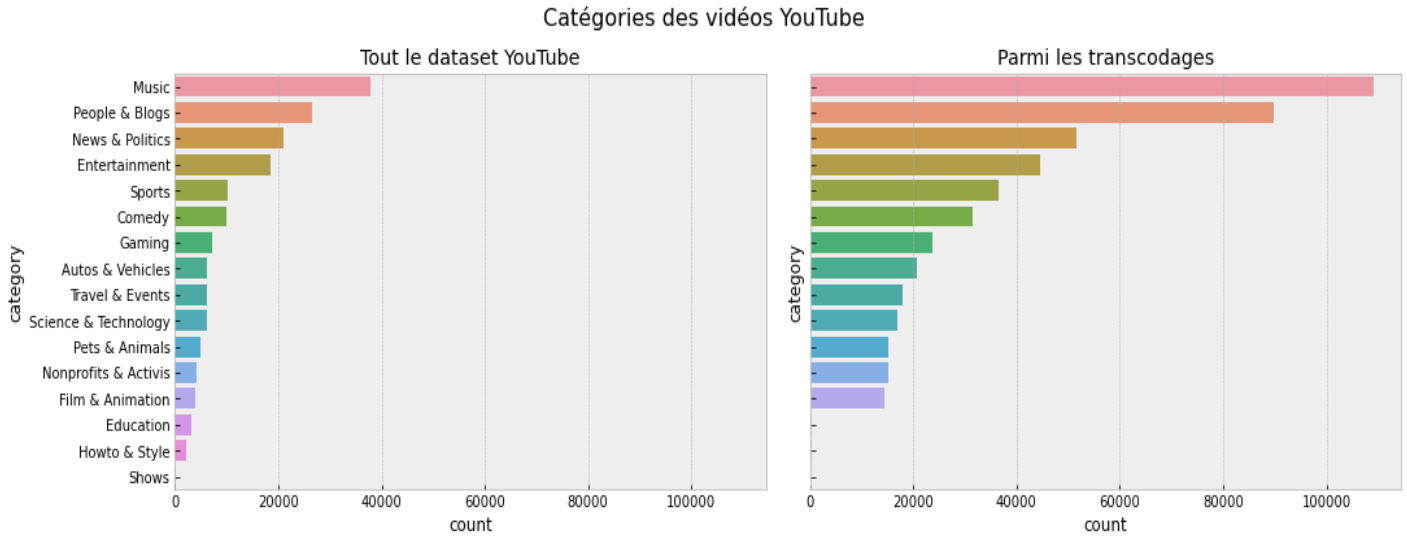


On affiche les tables de corrélations pour les deux datasets afin d'avoir une rapide idée du comportement des valeurs numériques.

A première vue, **umem** et **utime** sont assez indépendantes des autres valeurs.





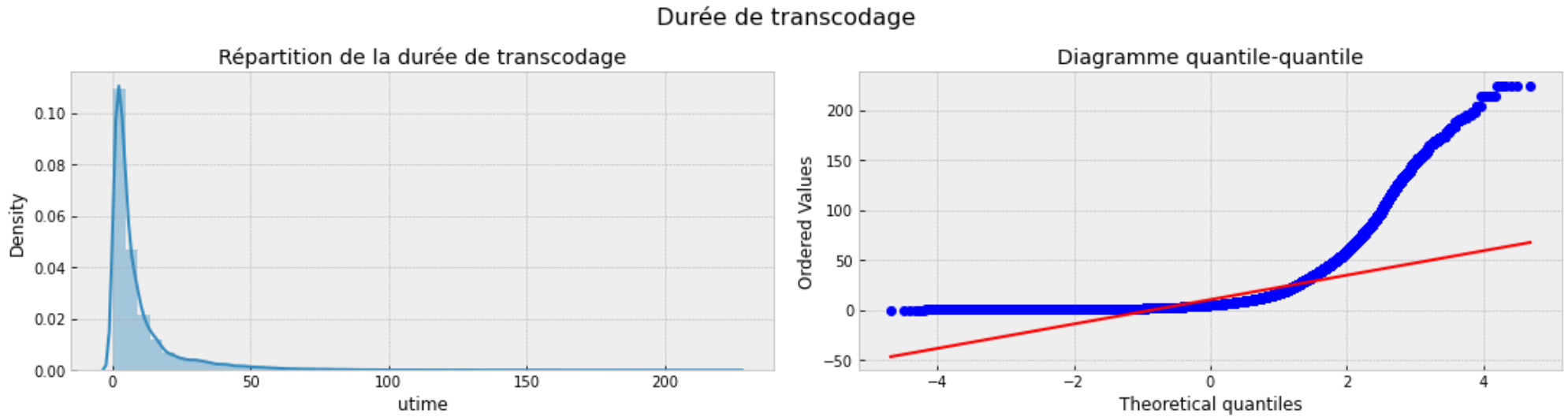


On vérifie également la répartition des valeurs non numériques présentes dans **codec** et o_**codec** d'une part et **category** d'autre part.

Dans le dataset YouTube, si la répartition de ces valeurs était relativement inégale, il a été décider pour réaliser le dataset de transcodage d'avoir un meilleur équilibre des classes de codecs et de catégorie de vidéo.

Cible n°1 : le temps de transcodage **utime**

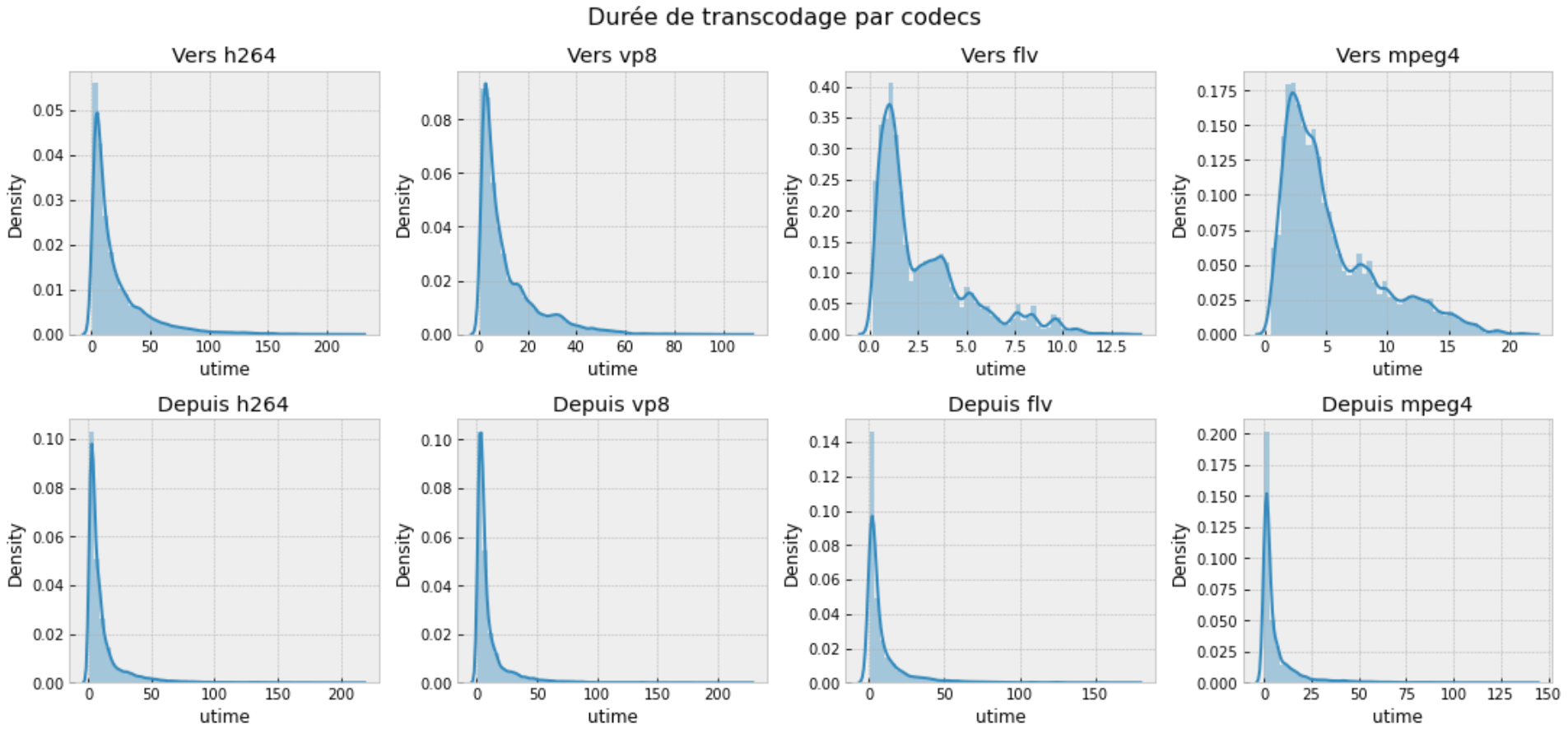
Pour poursuivre la visualisation des données, on décide de se concentrer d'abord sur le paramètre **utime**, première potentielle cible à prédire.



La répartition de **utime** n'est pas normalisée et suit vraisemblablement une loi exponentielle ou de Poisson.

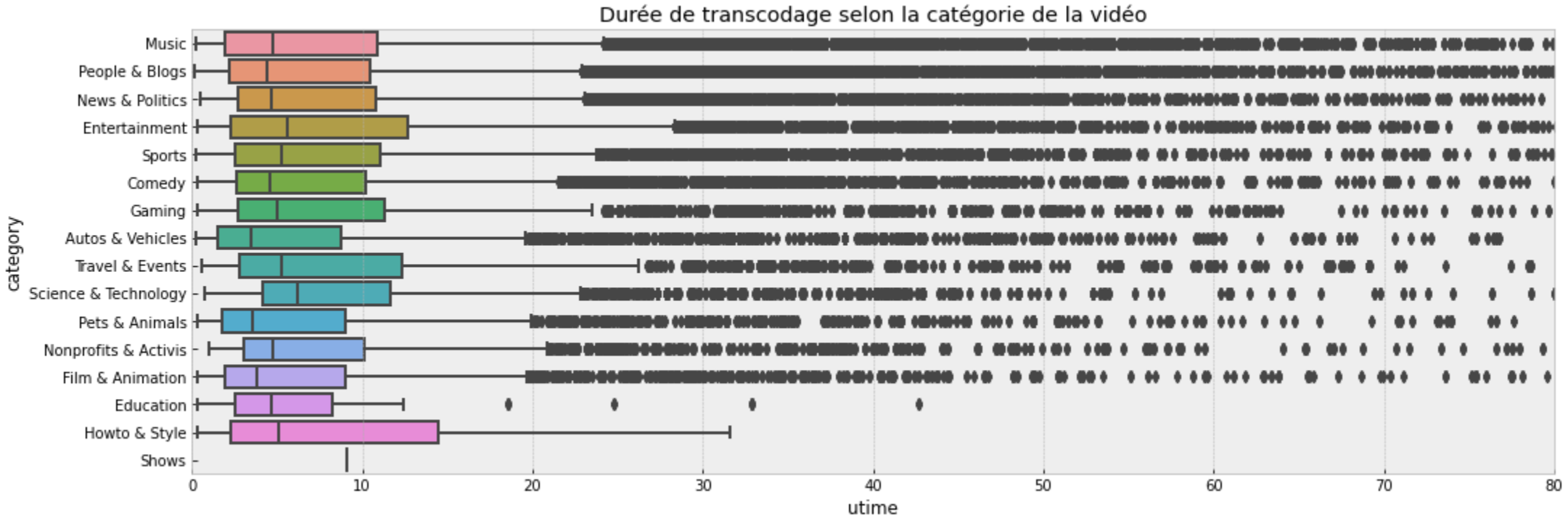
Regardons cette répartition en fonction des codecs d'entrée **codec** et de sortie **o_codec**.

Cette répartition est toujours asymétrique et très concentrée en « ses sommets ».

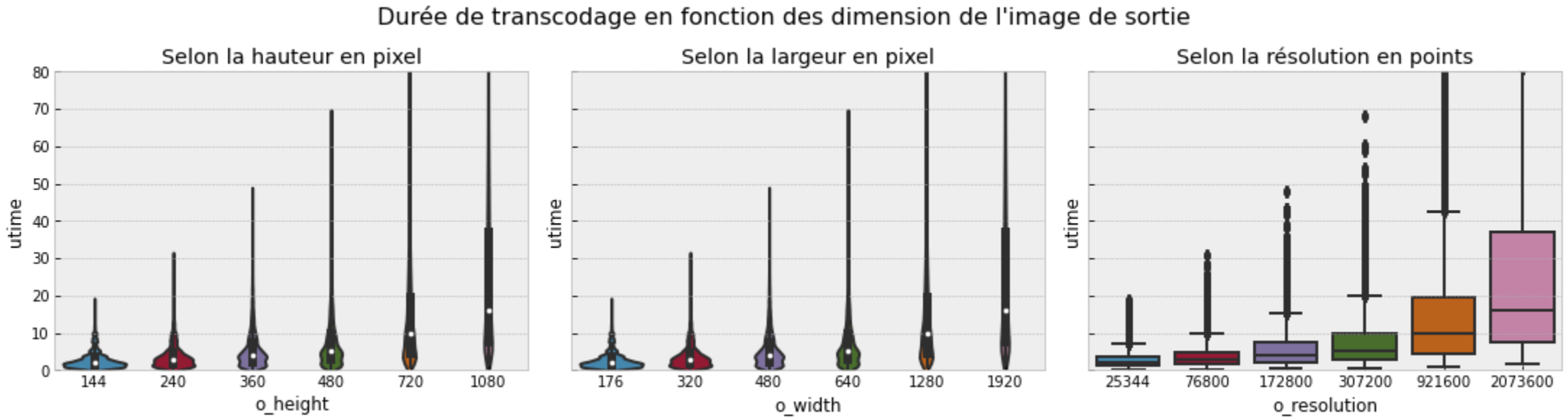


Regardons l’incidence de la catégorie de la vidéo **category** sur la durée de transcodage **utime**.

Peu d’informations ressortent. On peut peut-être remarquer que les catégories "Autos & Events" et "Pets & Animals" demandent souvent moins de temps de transcodage, mais une explication poussée à ces résultats paraît hasardeuse.



Regardons l'incidence de la résolution de la vidéo en sortie de transcodage (hauteur **o_height** et largeur **o_width**) sur la durée de transcodage **utime**.



Les durées de transcodage sont très étalées, mais on voit bien que plus la résolution de sortie est importante, plus le transcodage nécessite de temps. On rappelle les associations hauteur – largeur suivantes :

144 px × 176 px = 25 344 pts

240 px × 320 px = 76 800 pts

360 px × 450 px = 172 800 pts

480 px × 640 px = 307 200 pts

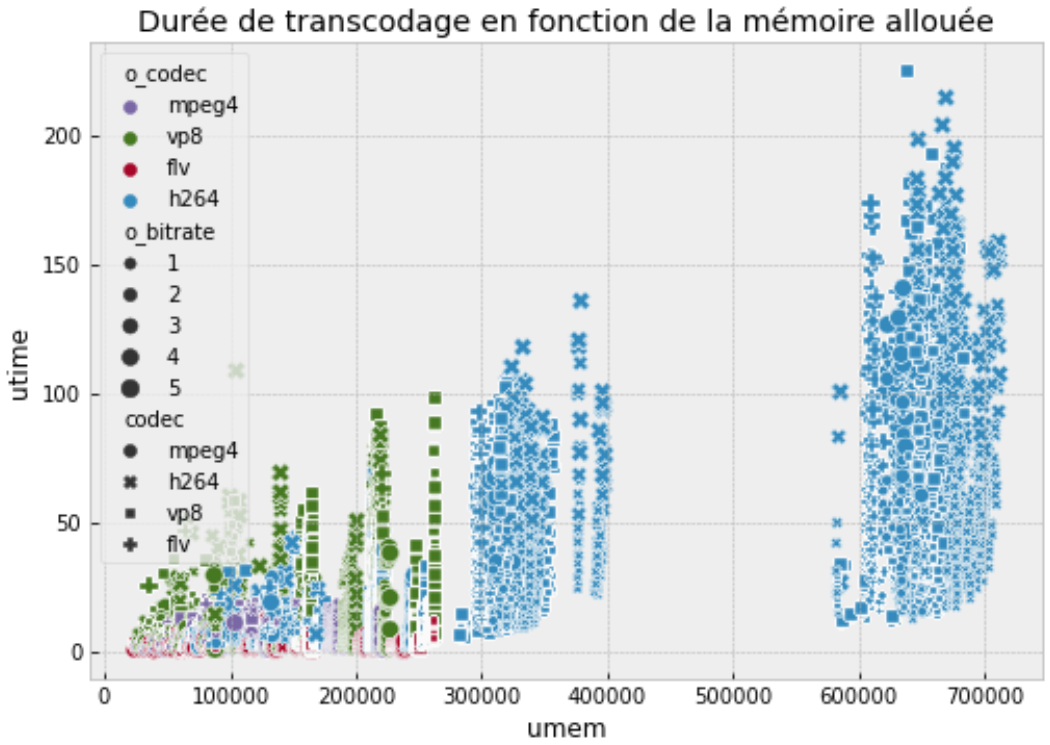
720 px × 1280 px = 921 600 pts

1080 px × 1920 px = 2 073 600 pts

Regardons l'incidence de la mémoire allouée pour le transcodage **umem** sur la durée de transcodage **utime**.

Le marqueur sur le graphique représente le codec d'entrée **codec**, la couleur représente le codec de sortie o_**codec** et la taille du marqueur représente le bitrate de sortie o_**bitrate**.

On remarque que plus **utime** est grand, plus **o_bitrate** est important.



Différents regroupements apparaissent indépendamment du codec de sortie **o_codec**.

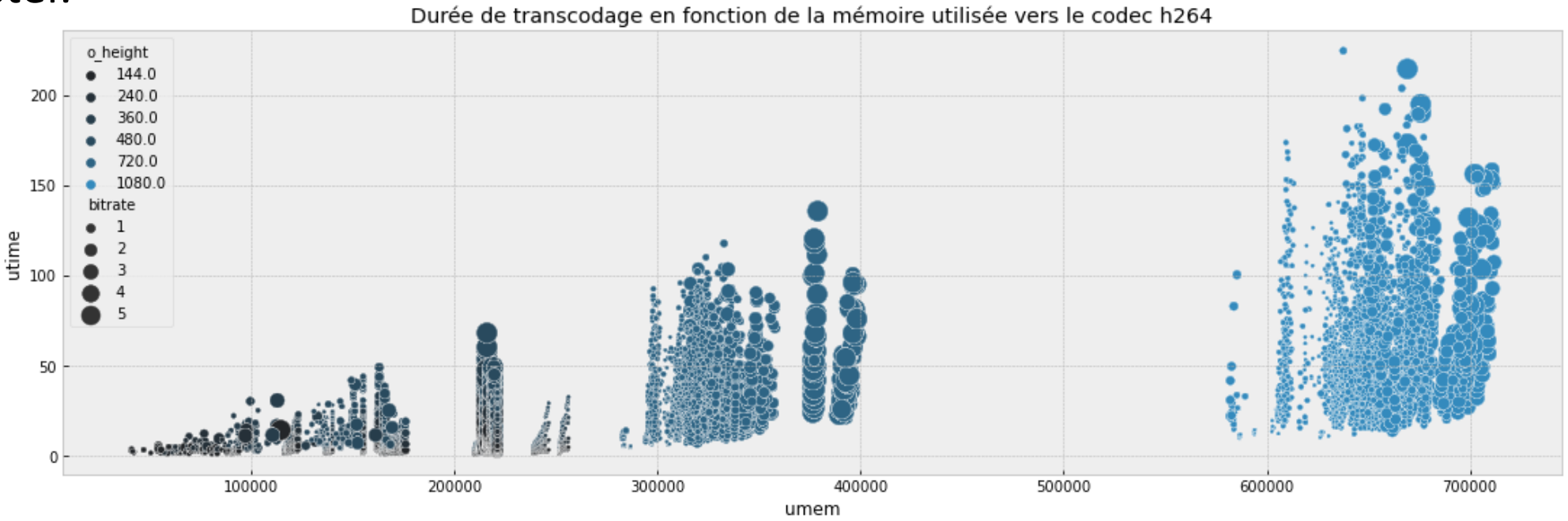
En bleu, les vidéos transcodées vers le codec h264 : on observe au moins trois clusters qui correspondent à cette couleur.

Il semble qu'il en va de même pour les autres codecs de sortie.

On étudie plus particulièrement **utime** en fonction de **umem** pour les transcodages vers le codec h264 uniquement (couleur bleue).

Pour identifier l'origine des clusters cités précédemment, on introduit la résolution de sortie de transcodage (uniquement **o_height**), caractérisée par une tonalité particulière de bleu, ainsi que le bitrate d'entrée de la vidéo.

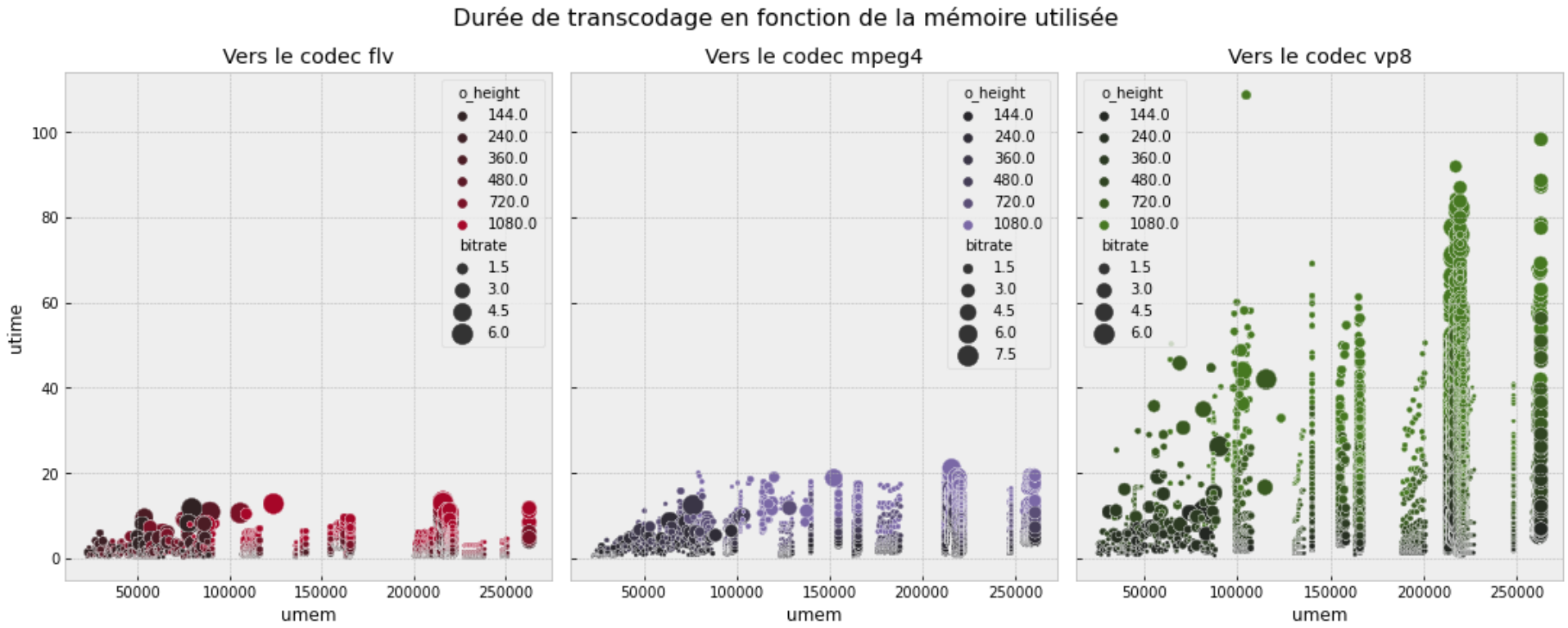
Chaque résolution a son cluster : la résolution de sortie est donc la principale raison de différences pour **umem**. De plus, dans chaque cluster, plus **bitrate** est important, plus **umem** est grand et donc plus la vidéo se place « à droite » de son cluster.



On s’intéresse à ce phénomène pour les trois autres codecs de sortie.

Ici, les vidéos se regroupent par colonnes où le bitrate est équivalent. On sait en réalité que ces colonnes sont des colonnes où le codec d’entrée **codec** est également le même (voir le premier graphique comparant **utime** à **umem**).

Dans ces colonnes, **utime** dépend principalement de la résolution de sortie (à la différence des transcodages vers le h264).



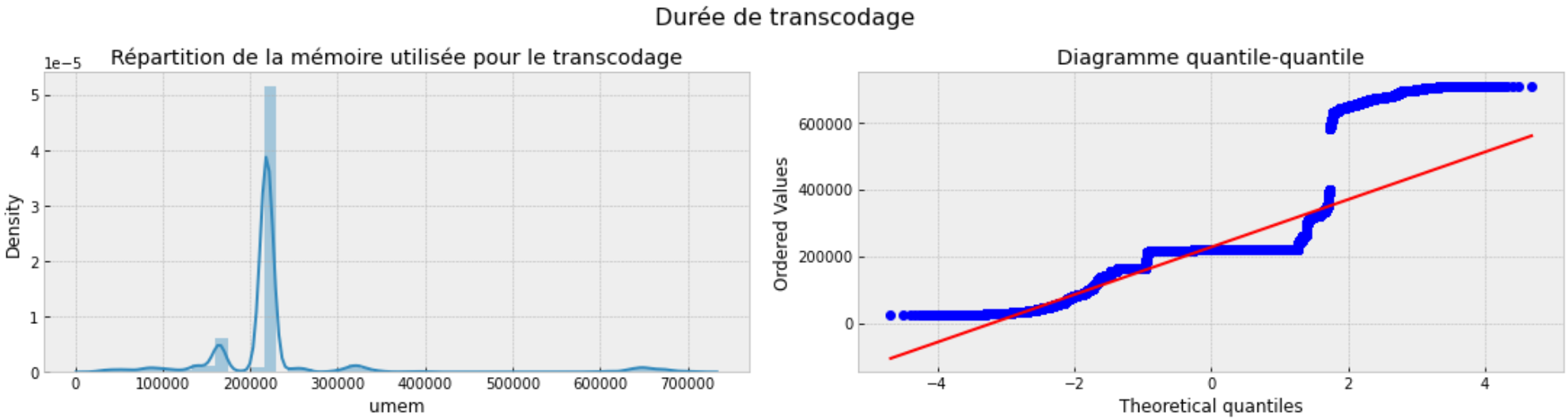
En résumé, pour les transcodages vers le codec h264 :
- **utime** augmente avec **o_bitrate** ;
- **umem** est relativement équivalent pour les vidéos de même résolution de sortie ;
- **umem** augmente légèrement avec **o_bitrate**.

Pour les transcodages vers les autres codecs :
- **utime** augmente avec la résolution de sortie ;
- **umem** augmente plutôt avec **bitrate**.

On décide de s'intéresser, pour la suite de la visualisation des données, à la mémoire allouée **umem** et à ses rapports avec le dataset.

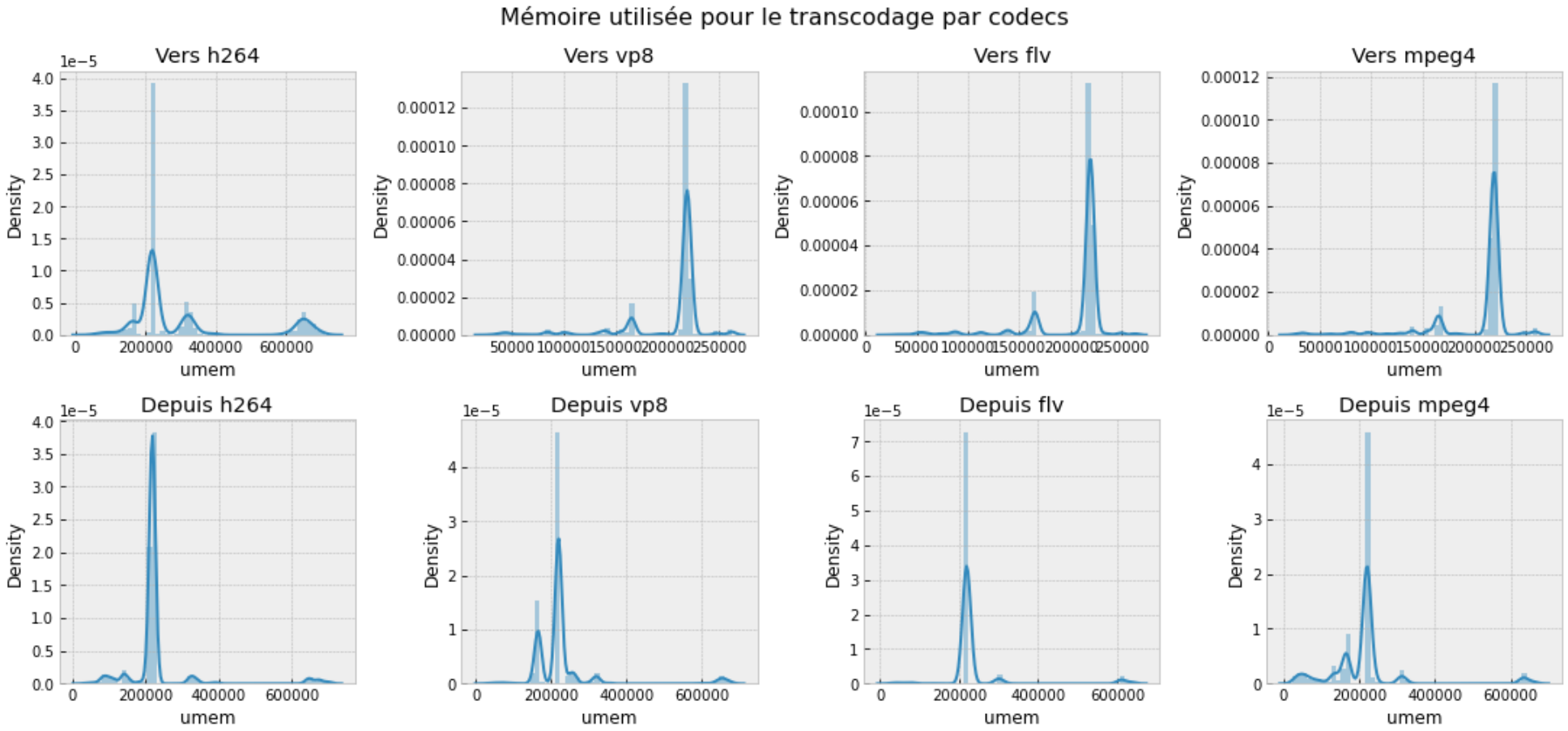
Cible n°2 : la mémoire allouée **umem**

Comme pour **utime**, étudions d'abord la répartition des valeurs en fonction du paramètre **umem**, deuxième potentielle cible à prédire.



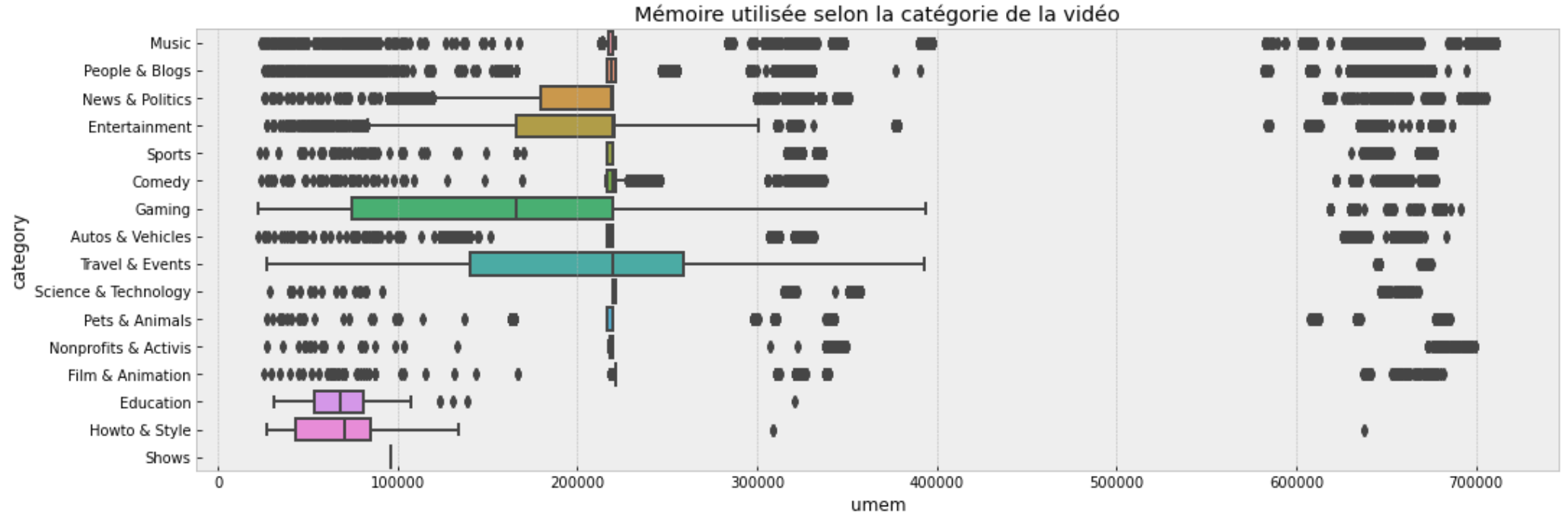
Regardons cette répartition en fonction des codecs d'entrée **codec** et de sortie **o_codec**.

Cette répartition est toujours asymétrique mais moins concentrée en certaines valeurs que pour **utime**.



Regardons l'incidence de la catégorie de la vidéo **category** sur la mémoire allouée **umem**.

On réobserve les mêmes groupes de répartition des valeurs de **umem**, principalement en fonction de la résolution de l'image de sortie. On en déduit qu'il sera moins aisé de réaliser une prédiction de la mémoire « automatiquement » allouée au transcodeur et qu'il sera plus intéressant d'ajouter **umem** aux entrées pour se concentrer sur la prédiction de la durée de transcodage **utime**. **umem** deviendrait pour notre étude une mémoire « manuellement » allouée à la machine.



Machine Learning

Comme nous avons décidé d'utiliser **umem** en tant que paramètre d'entrée, en supposant que l'on connait la mémoire allouée au transcodage, nous concentrons le travail de prédiction sur le temps de transcodage **utime**.

1. Traitement des données
   a. Ajout des données non numériques
   b. Séparation des données
   c. Normalisation
   d. Visualisation des corrélations pour le dataset d'entraînement
2. Choix du modèle
   a. Modèles avec Scikit-Learn
   b. Modèles avec Keras

1. Traitement des données

Pour inclure les paramètres non numériques aux *features* qui serviront à prédire utime, on ajoute autant de nouvelles colonnes booléennes qu'il y a de valeurs différentes pour **codec**, **o_codec** et **category**. Ces colonnes valent 0 ou 1 en fonction de ce que valent les paramètres non numériques. On passe ainsi d'un dataset de 25 paramètres à un dataset de 42 paramètres.

	duration	width	height	bitrate	framerate	i	p	b	frames	i_size	...	Pets & Animals	Sports	Music	Comedy	Nonprofits & Activis	Film & Animation	Science & Technology	Howto & Style	Education	Shows
0	130.35667	176	144	54590	12.0	27	1537	0	1564	64483	...	0	0	0	0	0	0	0	0	0	0
1	130.35667	176	144	54590	12.0	27	1537	0	1564	64483	...	0	0	0	0	0	0	0	0	0	0
2	130.35667	176	144	54590	12.0	27	1537	0	1564	64483	...	0	0	0	0	0	0	0	0	0	0
3	130.35667	176	144	54590	12.0	27	1537	0	1564	64483	...	0	0	0	0	0	0	0	0	0	0
4	130.35667	176	144	54590	12.0	27	1537	0	1564	64483	...	0	0	0	0	0	0	0	0	0	0

5 rows × 42 columns

On affiche ici les cinq premières lignes du DataFrame correspondant.

On poursuit la préparation des données en les séparant en deux datasets : un premier dataset d'entraînement, qui servira à améliorer les modèles et un second dataset de test, qui servira à tester les modèles.

Si besoin, un troisième dataset de validation pourra être utilisé, créé à partir des données d'entraînement.

Cette séparation des données inclut également l'isolement du paramètre cible.

```
from sklearn.model_selection import train_test_split
train_features, test_features, train_labels, test_labels = train_test_split(features, labels)
```

	duration	width	height	bitrate	framerate	i	p	b	frames	i_size	...
163271	64.93300	320	240	236793	15.000000	39	936	0	975	318954	...
446594	311.10000	176	144	56416	12.000000	113	3620	0	3733	127552	...
5079	130.35667	176	144	54590	12.000000	27	1537	0	1564	64483	...
250883	379.32000	480	360	181960	25.000000	195	9289	0	9484	3392479	...
334292	385.05200	1280	720	441862	23.979221	77	9155	0	9232	16244376	...

	utime
163271	0.688
446594	1.244
5079	0.356
250883	2.560
334292	1.840

Les données étant inégalement réparties, il convient ensuite de normaliser les valeurs des différents paramètres.

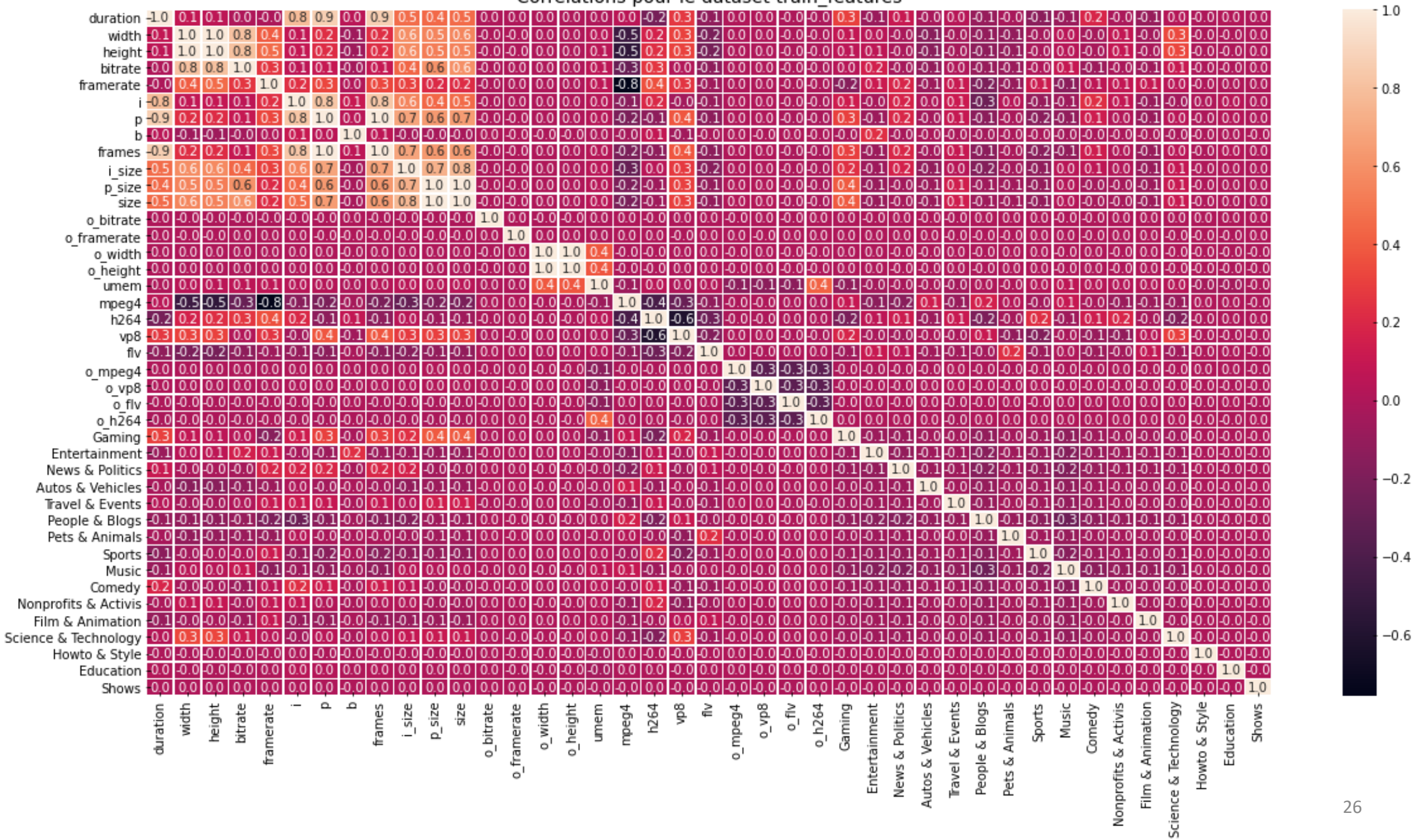
On créée avec Keras un calque de normalisation basé sur les valeurs du dataset d'entraînement `train_features`.

```
normalizer = preprocessing.Normalization()
Normalizer.adapt(np.array(train_features))
```

```
Avant normalisation : [[     64.93     320.       240.    236793.        15.        39.
      936.         0.       975.    318954.   1603007.   1921961.
   820000.        29.97     480.       360.    217080.         0.
        0.         0.         1.         0.         0.         1.
        0.         0.         0.         0.         0.         0.
        0.         0.         0.         1.         0.         0.
        0.         0.         0.         0.         0.  ]]

Après normalisation : [[-0.76 -0.77 -0.83 -0.5  -1.22 -0.71 -0.89 -0.08 -0.89 -0.63 -0.45 -0.48
  -0.33  1.32 -0.53 -0.46 -0.11 -0.44 -0.95 -0.64  3.68 -0.58 -0.58  1.74
  -0.57 -0.23 -0.32 -0.34 -0.21 -0.2  -0.48 -0.18 -0.28  1.86 -0.26 -0.18
  -0.18 -0.19 -0.02 -0.02 -0.  ]]
```

Avant de tester différents modèles sur ces données, on visualise une dernière fois la table des corrélations des paramètres sur les données d'entraînement.


Corrélations pour le dataset train_features
duration
width
height
bitrate
framerate
i
p
b
frames
i_size
p_size
size
o_bitrate
o_framerate
o_width
o_height
umem
mpeg4
h264
vp8
flv
o_mpeg4
o_vp8
o_flv
o_h264
Gaming
Entertainment
News & Politics
Autos & Vehicles
Travel & Events
People & Blogs
Pets & Animals
Sports
Music
Comedy
Nonprofits & Activis
Film & Animation
Science & Technology
Howto & Style
Education
Shows
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

2. Choix du modèle

Nous allons tester différents modèles avec différents paramètres afin de choisir le modèle le plus adapté à la prédiction de **utime**.

Le modèle choisi sera celui présentant le meilleur R^2.

Nous utiliserons des modèles de régression proposés par Scikit-Learn, puis par Keras (avec l'aide de TensorFlow) :

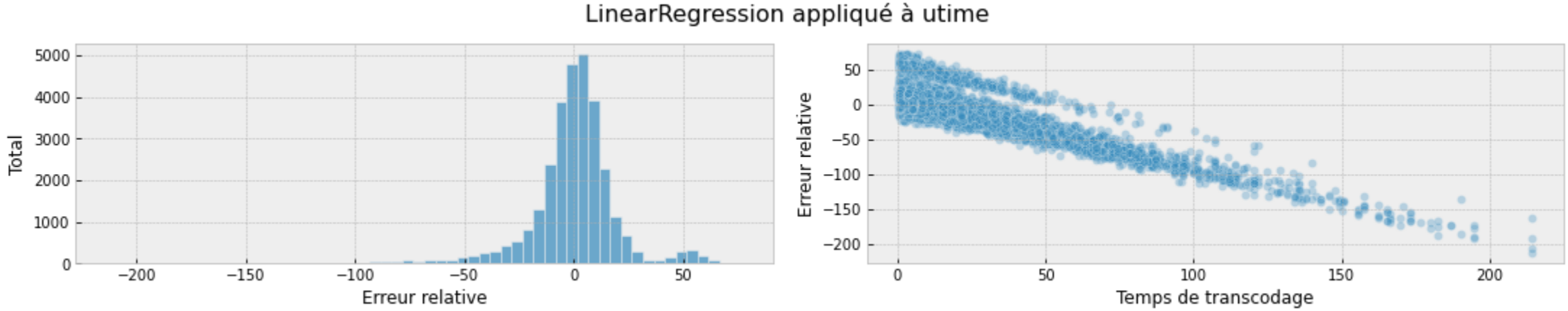
- Régression linéaire simple
- Régression Lasso
- Régression avec arbre de décision
- Régression KNN
- Régression avec réseau de neurones DNN

Régression linéaire avec **LinearRegression** de Scikit-Learn

On normalise d'abord les données avec un `scaler` MinMax de Scikit-Learn.

On utilise ensuite les paramètres par défaut de `LinearRegression`.

```
models[model_name] = LinearRegression()
models[model_name].fit(train_features_scaled, train_labels['utime'])
```

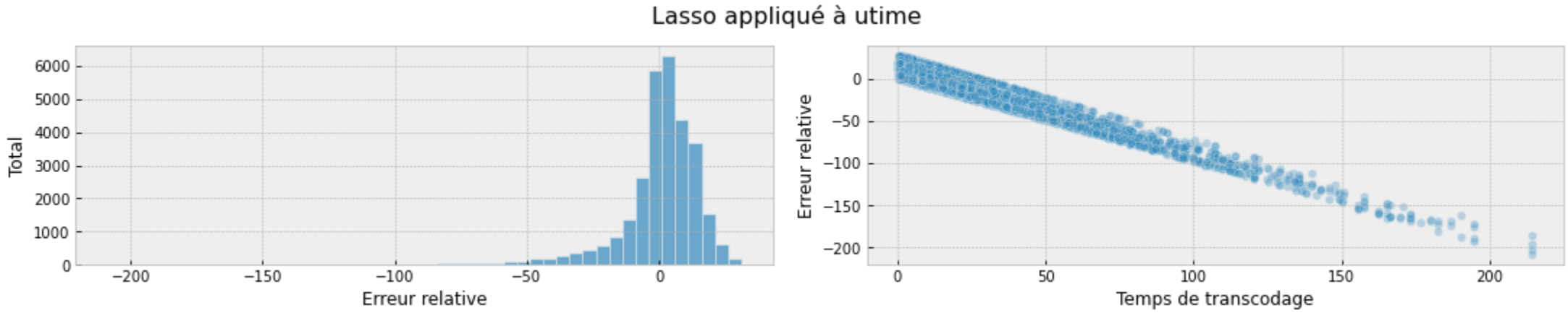


Régression Lasso avec **Lasso** de Scikit-Learn

On normalise d'abord les données avec un `scaler` MinMax de Scikit-Learn.

On utilise ensuite les paramètres par défaut de `Lasso`.

```
models[model_name] = Lasso()
models[model_name].fit(train_features_scaled, train_labels['utime'])
```

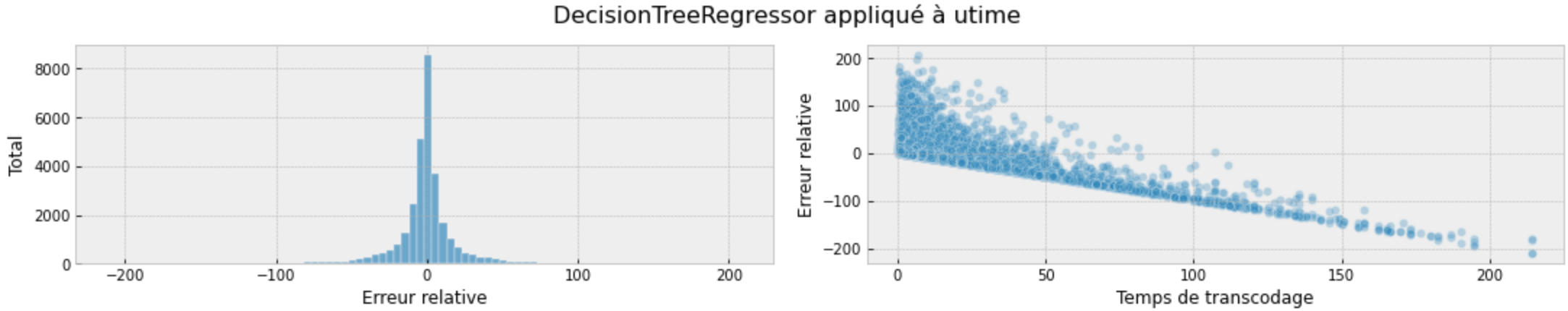


Régression avec arbre de décision avec **DecisionTreeRegressor** de Scikit-Learn

On ne normalise pas les données ici.

On utilise ensuite les paramètres par défaut de `DecisionTreeRegressor`.

```
models[model_name] = DecisionTreeRegressor()
models[model_name].fit(train_features, train_labels['utime'])
```

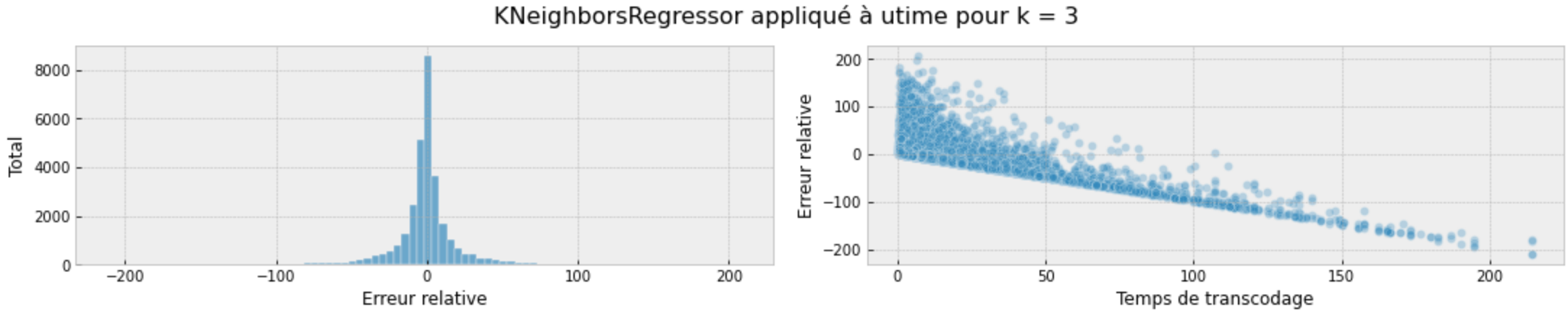


Régression KNN avec **KNeighborsRegressor** de Scikit-Learn

On normalise d'abord les données avec un `scaler` MinMax de Scikit-Learn.

On teste pour différents K le `DecisionTreeRegressor`.

```
models[model_name] = KNeighborsRegressor(n_neighbors = k)
models[model_name].fit(train_features_scaled, train_labels['utime'])
```



Régression avec réseau de neurones DNN avec **Sequential** de Keras

On normalise d'abord les données avec un `normalizer` de Keras.

On établit 2 couches de 64 neurones et on détermine les paramètres d'entraînement suivant :

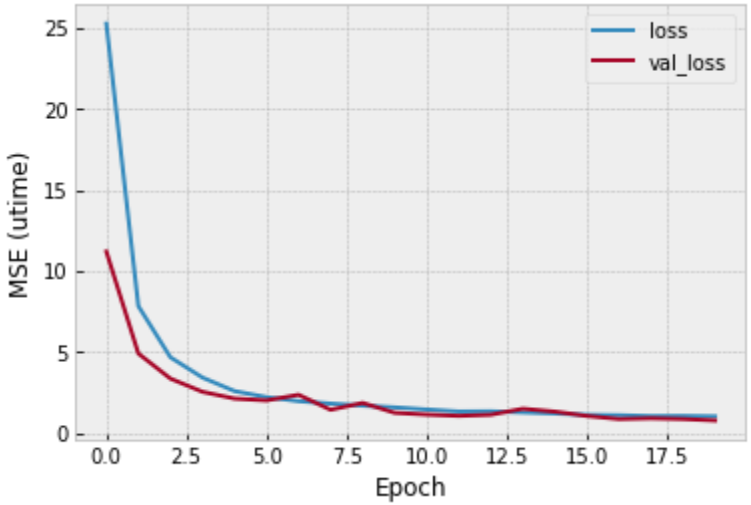
- Erreur : *Mean Squared Error*
- Optimiseur : Adam(0.001)

```
models[model_name] = keras.Sequential([
      normalizer,
      layers.Dense(64, activation='relu'),
      layers.Dense(64, activation='relu'),
      layers.Dense(1)
  ])

models[model_name].compile(loss='mean_squared_error',
    optimizer=tf.keras.optimizers.Adam(0.001))
```

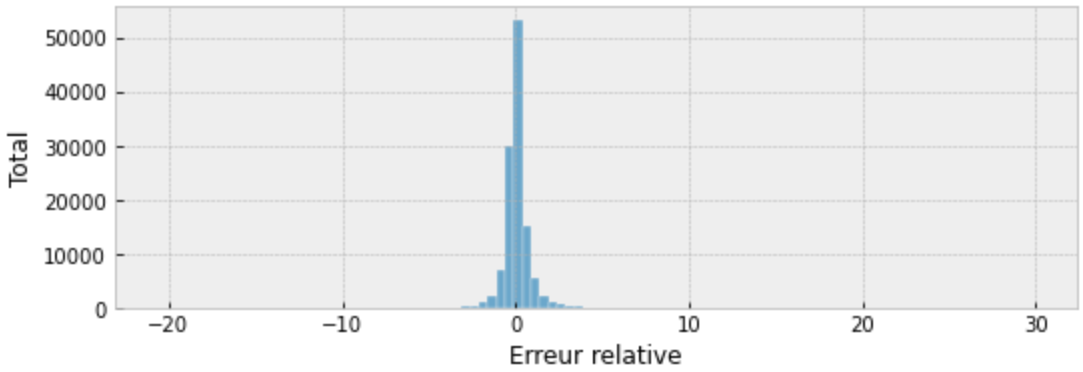
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
normalization (Normalization (None, 41)                83
_________________________________________________________________
dense (Dense)                (None, 64)                2688
_________________________________________________________________
dense_1 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 6,996
Trainable params: 6,913
Non-trainable params: 83
_________________________________________________________________
```

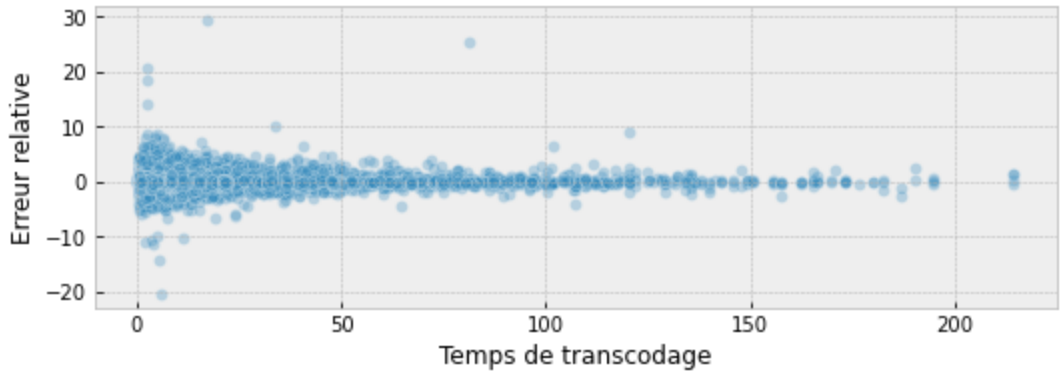
Le modèle n’est pas surentraîné : `loss` et `val_loss` sont équivalent et évoluent de la même manière lors des dernières époques.




```
dnn_history = models[model_name].fit(
    train_features, train_labels['utime'],
    validation_split=0.2,
    verbose=0, epochs=20)
```

DNN appliqué à utime





Choix du modèle

	Modèle	R^2	MSE
2	DecisionTreeRegressor	0.999947	0.014012
4	KNN 3	0.999122	0.233302
6	DNN ([64, 64, 1], MSE, Adam 0.001)	0.996828	0.842587
1	KNN 5	0.995807	1.113766
5	KNN 11	0.979867	5.347772
0	LinearRegression	0.661217	89.987608
3	Lasso	0.385506	163.222155

Selon toute vraisemblance, les calculs du R^2 et du MSE pour le `DecisionTreeRegressor` et les `KNeighborsRegressor` sont biaisés. Nous n'avons pas trouvé d'explication à ce phénomène.

Nous choisissons donc le modèle DNN réalisé avec Keras, que nous enregistrons.

```
models[choice].save('saves') # La normalisation est inclue dans le modèle avec Keras
```

Rendu final

On créée une API avec l’aide de Flask.

Une première page comporte un formulaire pour compléter toutes les valeurs d’entrée du modèle. Une deuxième page affiche le résultat de la régression.