Saupin Guillaume Follow

Sep 9, 2020 · 7 min read ★ · Listen

Confidence intervals for XGBoost

Building a regularized Quantile Regression objective

Gradient Boosting methods are a very powerful tool for performing accurate predictions quickly, on large datasets, for complex variables that depend non linearly on a lot of features. The underlying mathematical principles are explained in my other post:

DIY XGBoost library in less than 200 lines of python

XGBoost explained as well as gradient boosting method and HP tuning by building your own gradient boosting library for…

towardsdatascience.com

Moreover, it has been implemented in various ways: XGBoost, CatBoost, GradientBoostingRegressor, each having its own advantages, discussed here or here. Something these implementations all share is the ability to choose a given objective for training to minimize. And even more interesting is the fact that XGBoost and CatBoost offer easy support for a custom objective function.

Why do I need a custom objective?

more specific solution to achieve the expected level of precision. Using a custom objective is usually my favourite option for tuning models.

Note that you can use Hyper Parameters Tuning to help with finding the best objective. See my two papers on the subject:

Tuning XGBoost with XGBoost: Writing your own Hyper Parameters Optimization engine

towardsdatascience.com

AutoML for fast Hyperparameters tuning with SMAC

Use AutoML for Finding your path in high dimensional spaces

towardsdatascience.com

Can you provide us with an example?

Sure! Recently, I've been looking for a way to associate the prediction of one of our models with confidence intervals. As a short reminder, confidence intervals are characterised by two elements:

1. An interval [x_l, x_u]
2. The confidence level *C* ensures that C% of the time, the value that we want to predict will lie in this interval.

For instance, we can say that the 99% confidence interval of the average temperature on earth is [-80, 60].

How do you compute confidence intervals?

You'll need to train two models :

- One for the upper bound of your interval
- One for the lower bound of your interval

And guess what? You need specific metrics to achieve that: Quantile Regression objectives. Both the scikit-learn GradientBoostingRegressor and CatBoost implementations provide a way to compute these, using Quantile Regression objective functions, but both use the non-smooth standard definition of this regression :

$$\frac{\sum_{i=1}^{N}(\alpha-1(t_i \leq a_i))(t_i-a_i)w_i}{\sum_{i=1}^{N} w_i}$$

Where *t_i* is the ith true value and *a_i* is the ith predicted value. *w_i* are optional weights used to ponderate the error. And *alpha* defines the quantile.

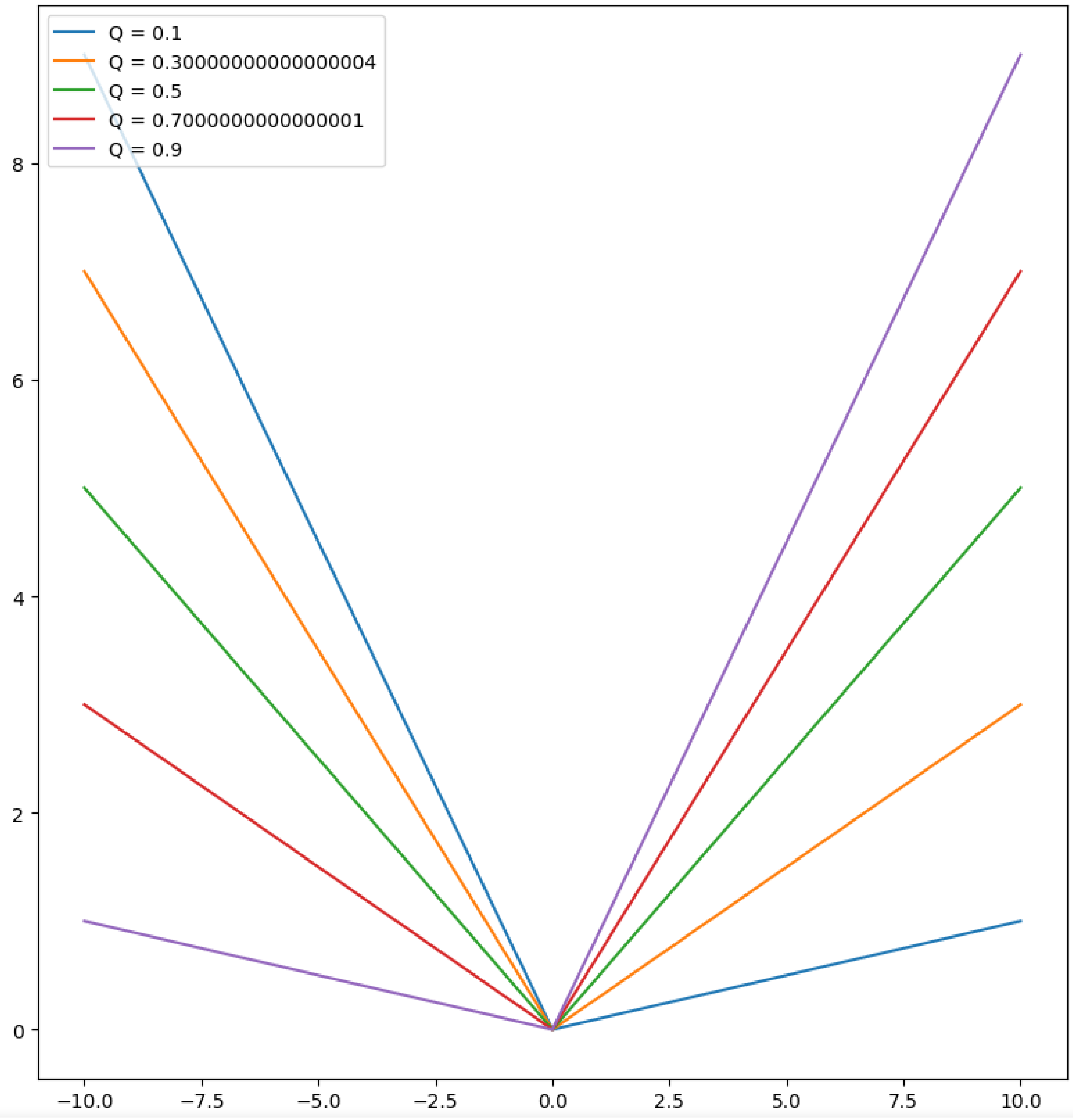
For instance, using this objective function, if you set *alpha* to 0.95, 95% of the obervations are below the predicted value. Conversely, if you set *alpha* to 0.05, only 5% of the observations are below the prediction. And 90% of real values lie between these two predictions.

Let's plot it using the following code, for the range [-10, 10] and various alphas:

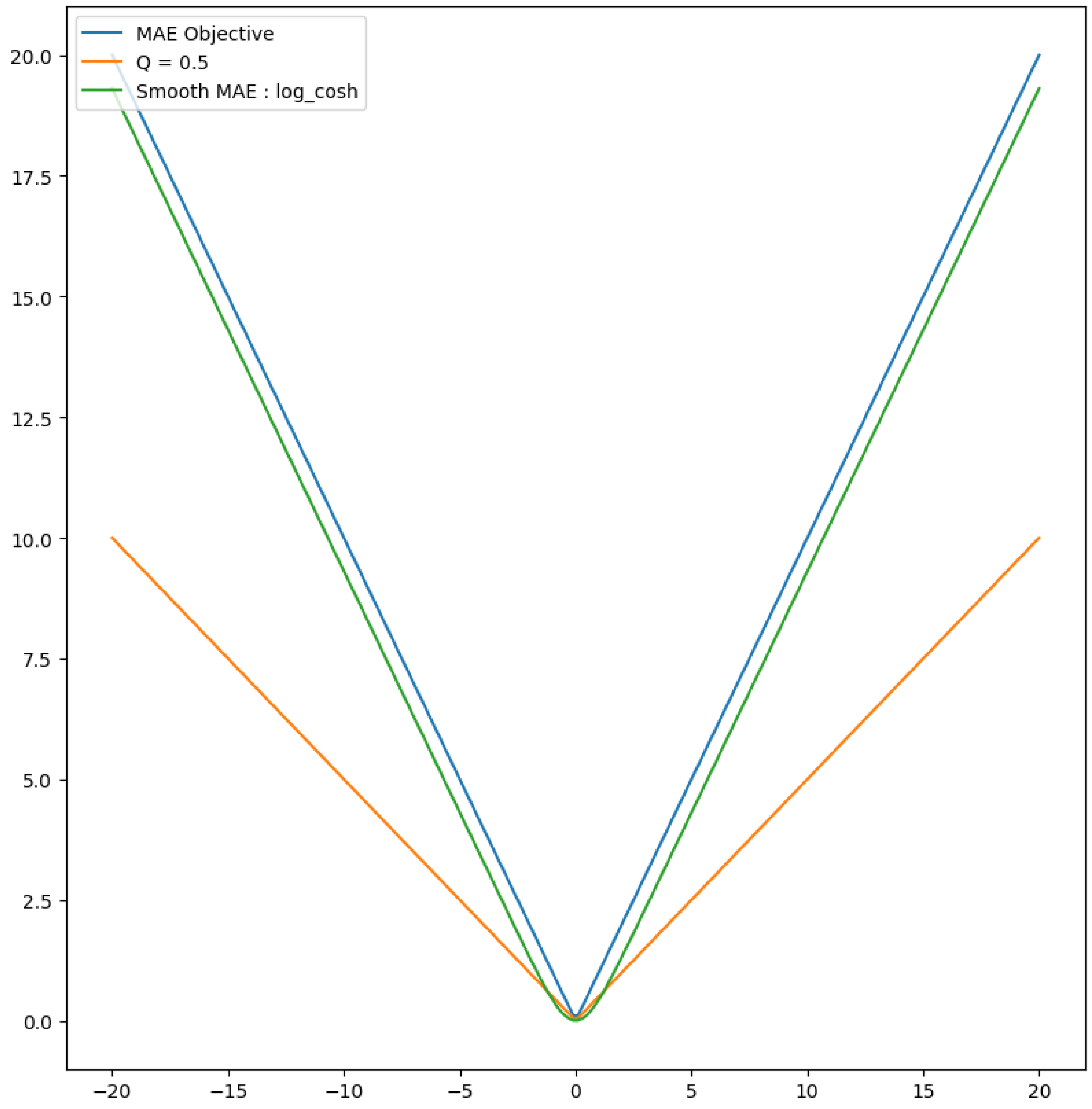
```
import matplotlib.pyplot as plt

for alpha in np.linspace(0.1, 0.9, 5):
    x = np.array(range(-10, 11))
    y = np.where(x > 0, alpha * x, (alpha - 1) * x)
    plt.plot(x, y, label=f'Q = {alpha}')
plt.legend(loc='upper left')
```

As you can see in the resulting plot below, this objective function is continuous but its derivative is not. There is a singularity in (0, 0), *i.e*. it's a C_0 function, with respect to the error, but not a C_1 function. This is an issue, as gradient boosting methods require an objective function of class C_2, i.e. that can be differentiated twice to compute the gradient and hessian matrices.



below should convince you :



The logcosh objective

$$\begin{cases} -x & ; \quad x < 0 \\ x & ; \quad x >= 0 \end{cases}$$

MAE objective formula

The figure above also shows a regularized version of the MAE, the logcosh objective. As you can see, this objective is very close to the MAE, but is smooth, i.e. its derivative is continuous and differentiable. Hence, it can be used as an objective in any gradient boosting method, and provides a reasonable rate of convergence compared to default, non-differentiable ones.

And as it is a very close approximation of the MAE, if we manage to scale and rotate it, we'll get a twice differentiable approximation of the quantile regression objective function.

You might have noticed that there is a slight offset between the curve of the MAE and the log cosh. We will explain that in detail a little further below.

The formula for the logcosh is straightforward :

$$\log(\cosh(x))$$

Formula for the logcosh objective

Rotation and scaling of the logcosh

All we need to do now is to find a way to rotate and scale this objective so that it becomes a good approximation of the quantile regression objective. Nothing complex here. As logcosh is similar to the MAE, we apply the same kind of change as for the Quantile Regression, i.e. we scale it using alpha :

$$\begin{cases} (1-\alpha) \cdot log(cosh(x)) & ; \quad x < 0 \end{cases}$$

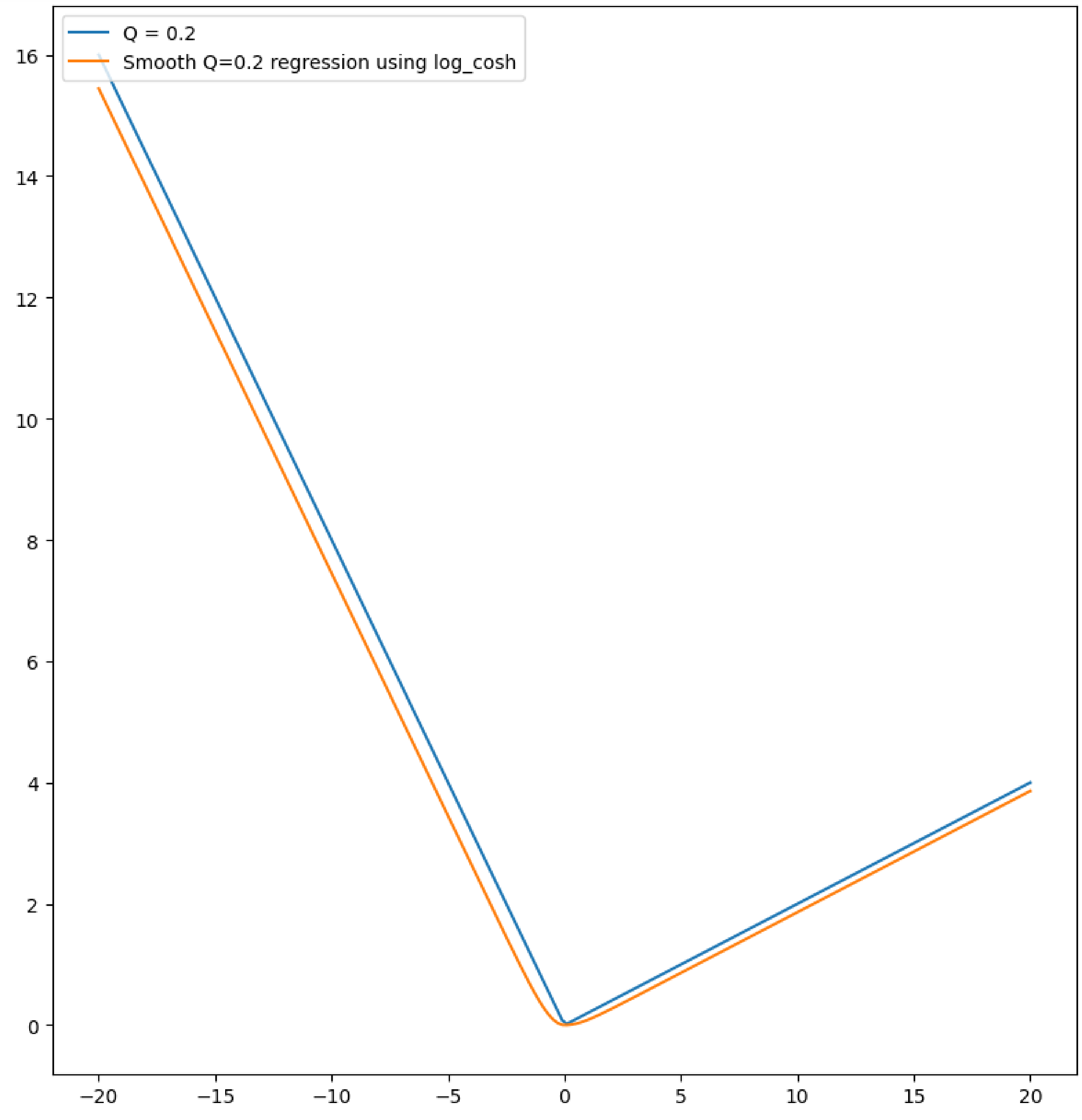
That can be done with these twelve lines of code:

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-20, 20, 200)
y = np.abs(x)
alpha = 0.2
y_Q = np.where(x > 0, alpha * x, (alpha - 1) * x)
y_logcosh = np.where(x > 0, alpha * np.log(np.cosh(x)), (1 - alpha) * np.log(np.cosh(x)))
plt.plot(x, y_Q, label=f'Q = 0.2')
plt.plot(x, y_logcosh, label=f'Smooth Q=0.2 regression using log_cosh')
plt.legend(loc='upper left')
plt.show()
```

plot_rotate_logcosh hosted with ❤ by GitHub view raw

And this works, as shown below :



But wait a minute!

You might be curious as to why combining two non-linear functions like log and cosh results in such a simple, near linear curve.

$$\cosh(x) = \frac{e^x + e^{-x}}{2}$$

cosh formula

When x is positive and large enough, *cosh* can be approximated by

$$\frac{e^x}{2}$$

Approximation of cosh when x >> 0

Conversely, when x is negative enough, *cosh* can be approximated by

$$\frac{e^{-x}}{2}$$

Approximation of cosh when x << 0

We begin to understand how combining these two formulae leads to such linear results. Indeed, as we apply the log to these approximations of cosh, we get :

$$\begin{aligned} log(cos(x)) &\approx log(\frac{e^x}{2}), \forall x, x \gg 0 \\ &\approx log(e^x) - log(2) \\ &\approx x - log(2) \end{aligned}$$

logcosh simplification for x >> 0

for x >>0. The same stands for x << 0 :

$$log(cos(x)) \approx log(\frac{e^{-x}}{2}), \forall x, x \ll 0$$

It is now clear why these two functions closely approximate the MAE. We also get as a side benefit the explanation for the slight gap between the MAE and the logcosh. It's log(2)!

Let's try it on a real example

It is now time to ensure that all the theoretical maths we perform above works in real life. We won't evaluate our method on a simple sinus, as proposed in scikit here ;) Instead, we are going to use real-world data, extracted from the TLC trip record dataset, that contains more than 1 billion taxi trips.

The code snippet below implements the idea presented above. It defines the logcosh quantile regression objective **log_cosh_quantile**, that computes its gradient and the hessian. Those are required to minimize the objective.

As stated at the beginning of this article, we need to train two models, one for the upper bound, and another one for the lower bound.

The remaining part of the code simply loads data and performs minimal data cleaning, mainly removing outliers.

```python
import pandas as pd
import numpy as np
from xgboost.sklearn import XGBRegressor
from sklearn.model_selection import ShuffleSplit
import matplotlib.pyplot as plt


# log cosh quantile is a regularized quantile loss function
def log_cosh_quantile(alpha):
    def _log_cosh_quantile(y_true, y_pred):
        err = y_pred - y_true
        err = np.where(err < 0, alpha * err, (1 - alpha) * err)
```

```
dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d %H:%M:%S')

dtf = pd.read_csv('data/taxi_data.csv',
                  parse_dates=['tpep_dropoff_datetime', 'tpep_pickup_datetime'],
                  date_parser=dateparse)

# Compute trip duration in minutes
dtf['duration'] = (dtf['tpep_dropoff_datetime'] - dtf['tpep_pickup_datetime']).apply(lambda x : x

# Do some minimal cleaning : remove outliers
dtf = dtf[(dtf['duration'] < 90) & (dtf['duration'] > 0)]

# identify useless columns and drop them
to_drop = ['tpep_dropoff_datetime',
           'tpep_pickup_datetime',
           'store_and_fwd_flag',
           'passenger_count',
           'RatecodeID',
           'store_and_fwd_flag',
           'PULocationID',
           'DOLocationID',
           'payment_type',
           'fare_amount',
           'extra',
           'mta_tax',
           'tip_amount']
dtf.drop(to_drop, axis=1, inplace=True)


# Create an object to split input dataset into train and test datasets
splitter = ShuffleSplit(n_splits=1, test_size=.25, random_state=0)

alpha = 0.95
to_predict = 'duration'

for train_index, test_index in splitter.split(dtf):
    train = dtf.iloc[train_index]
    test = dtf.iloc[test_index]

    X = train
```

Open in app Get started

```
    X_test.drop([to_predict], axis=1, inplace=True)

    # over predict
    model = XGBRegressor(objective=log_cosh_quantile(alpha),
                         n_estimators=125,
                         max_depth=5,
                         n_jobs=6,
                         learning_rate=.05)

    model.fit(X, y)
    y_upper_smooth = model.predict(X_test)

    # under predict
    model = XGBRegressor(objective=log_cosh_quantile(1-alpha),
                         n_estimators=125,
                         max_depth=5,
                         n_jobs=6,
                         learning_rate=.05)

    model.fit(X, y)
    y_lower_smooth = model.predict(X_test)
    res = pd.DataFrame({'lower_bound' : y_lower_smooth, 'true_duration': y_test, 'upper_bound': y
    res.to_csv('/tmp/duration_estimation.csv')

    index = res['upper_bound'] < 0
    print(res[res['upper_bound'] < 0])
    print(X_test[index])

    max_length = 150
    fig = plt.figure()
    plt.plot(list(y_test[:max_length]), 'gx', label=u'real value')
    plt.plot(y_upper_smooth[:max_length], 'y_', label=u'Q up')
    plt.plot(y_lower_smooth[:max_length], 'b_', label=u'Q low')
    index = np.array(range(0, len(y_upper_smooth[:max_length])))
    plt.fill(np.concatenate([index, index[::-1]]),
             np.concatenate([y_upper_smooth[:max_length], y_lower_smooth[:max_length][::-1]]),
             alpha=.5, fc='b', ec='None', label='90% prediction interval')
    plt.xlabel('$index$')
    plt.ylabel('$duration$')
    plt.legend(loc='upper left')
```

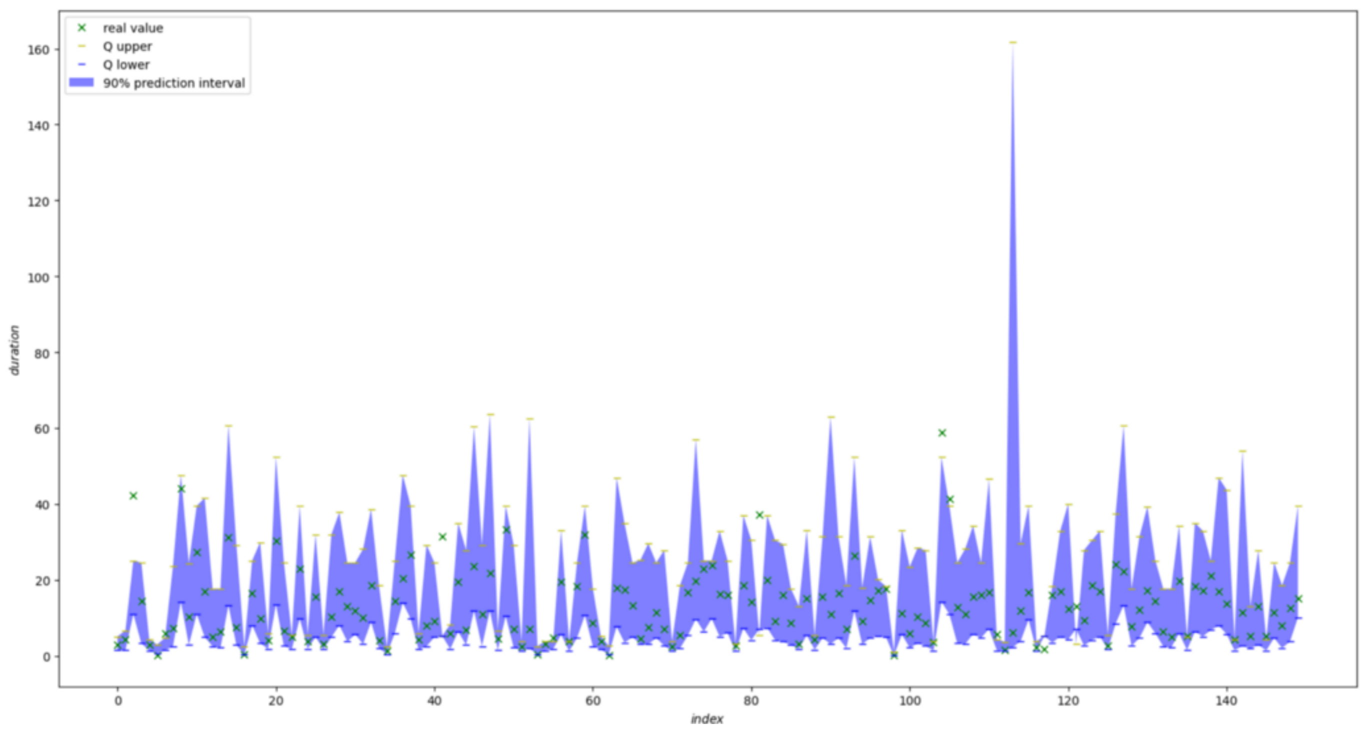
```
print(f'pref = {count} / {total}')
```

taxi_quantile_regression hosted with ❤ by GitHub view raw

In this code, we have chosen to compute the 90% confidence interval. Hence we use **alpha=0.95** for the upper bound, and **alpha=0.05** for the lower bound.

Hyperparameter tuning has been done manually, using fairly standard values. It could certainly be improved, but the results are good enough to illustrate this paper.

The last lines of the script are dedicated to the plotting of the first 150 predictions of the randomly build test set with their confidence interval:



Note that we have also included at the end of the script a counter to evaluate the number of real values whose confidence interval is correct. On our test set, 22 238 over 24 889

Conclusion

With simple maths, we have been able to define a smooth quantile regression objective function, that can be plugged into any machine learning algorithm based on objective optimisation.

Using these regularized functions, we have been able to predict reliable confidence intervals for our prediction.

This method has the advantage over the one presented here of being parameters-less. Hyperparameter tuning is already a demanding step in optimizing ML models, we don't need to increase the size of the configuration space with another parameter ;)

Thanks to Elisabeth Guegan, Elliot Gunn, and Nicolas David