Time Complexity Relationship Between `dropna()` Input Data Size & Execution Time

Polynomial time complexity is a fundamental concept in computer science, and understanding how it relates to the size of input data is crucial for optimizing algorithms and improving computational efficiency. In this analysis, we will explore the relationship between **the data input size *N*** and **the execution time** of a `dropna()` running program, and how this relationship can be analyzed using polynomial functions. A related discussing issue point can be found here regarding the fact of infinite loop caused in large frames.

Test Data

The data used for this analysis is generated from the following reproducible example with *N* representing for the input data size and a loop running to gain total milliseconds representing the time taken to process the data for a particular input size.

Reproducible Example

```
import pandas as pd
import numpy as np
import sys

class MyFrame(pd.DataFrame):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for col in self.columns:
            if self.dtypes[col] == "O":
                self[col] = pd.to_numeric(self[col], errors='ignore')
    @property
    def _constructor(self):
        return type(self)

def get_frame(N):
    return MyFrame(
        data=np.vstack(
            [np.where(np.random.rand(N) > 0.36, np.random.rand(N), np.nan)
        ).T,
        columns=[f"col{i}" for i in range(10)]
    )

# When N is smallish, no issue
frame = get_frame(1000)
frame.dropna(subset=["col0", "col1"])
print("1000 passed")

# Accept a data size value
if __name__ == '__main__':
    N = int(sys.argv[1])
    frame = get_frame(N)
    frame.dropna(subset=["col0", "col1"])
    print(f"{N} passed")
```

Run the loop to get the execution time

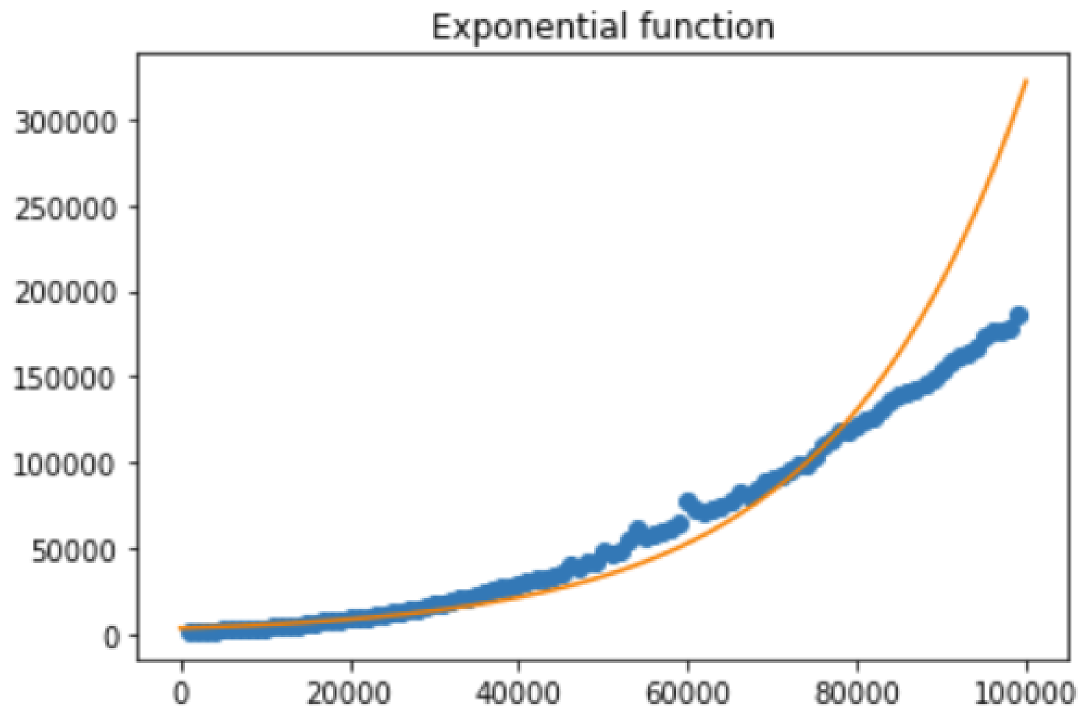
It helps to gain the execution time data for each 1000 $N \in [1,100000]$.

```
# Run the loop
$results = @()
for ($i = 1000; $i -lt 50000; $i += 1000) {
    $output = Measure-Command { py test.py $i } | Out-String
    $milliseconds = [double]($output -split 'TotalMilliseconds : ')[1]
    $results += [pscustomobject]@{N=$i; Milliseconds=$milliseconds}
}
# View the results in a table
$results | Out-GridView
```

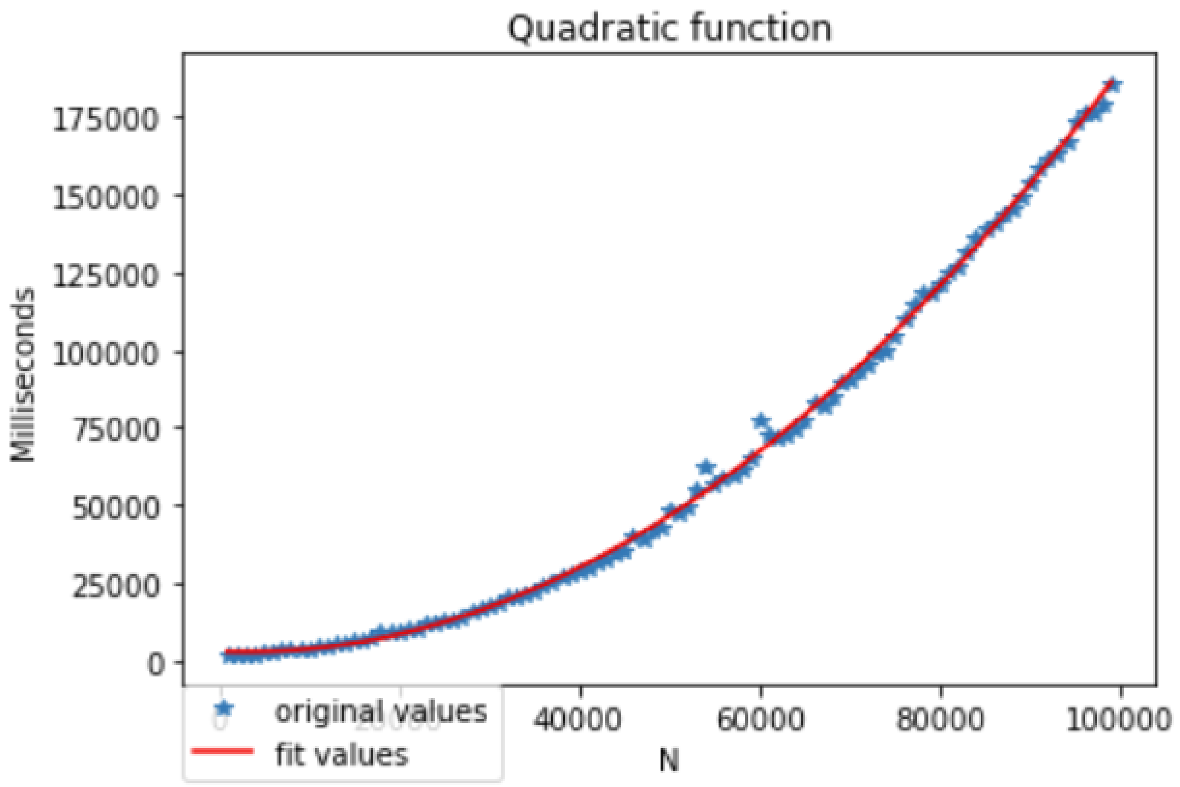
Curve Plotting & Polynomial Relationship

Basic Analysis

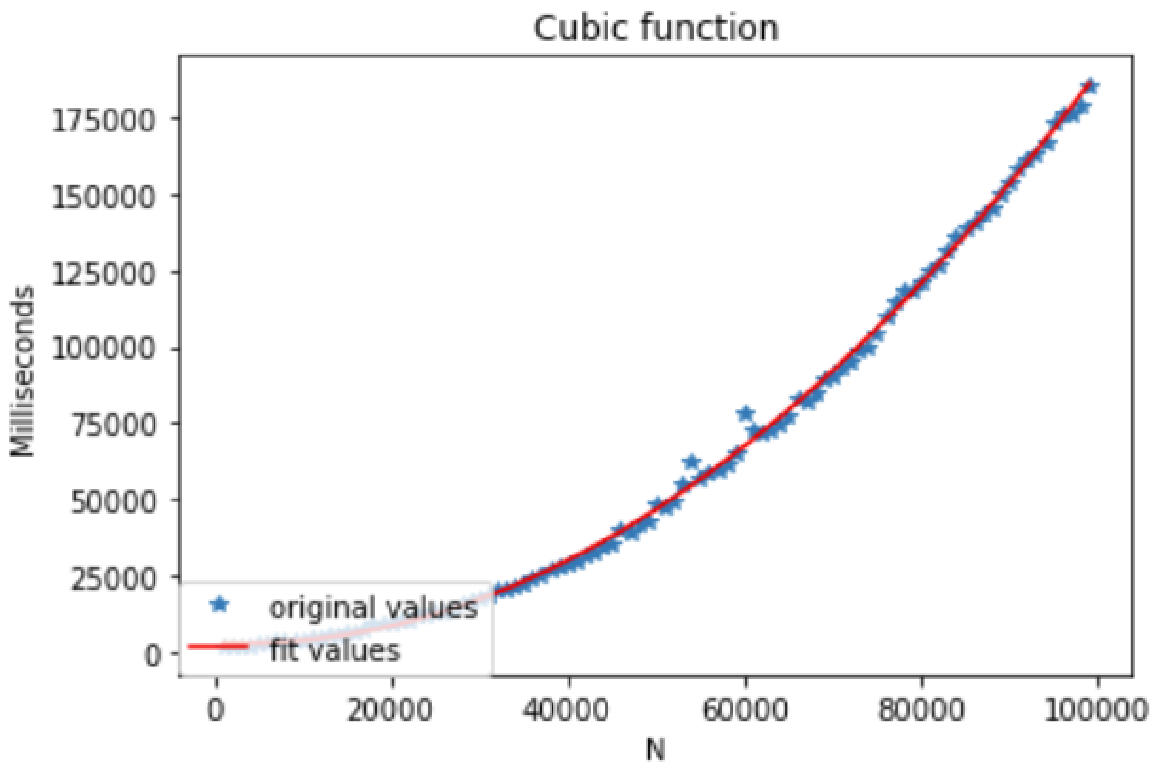
Initially, the exponential function is implemented trying to fit the curve, while the result turns to be not quite optimistic.



Then, a quadratic function is applied, which is represented by the equation y = ax^2 + bx + c. The function is fitted to the data using the `np.polyfit()` function, which returns the coefficients a, b, and c that best fit the data. These coefficients are then used to create a polynomial object using the `np.poly1d()` function, which can be evaluated at any point to obtain the predicted execution time for that input size. The resulting curve shows that the execution time increases rapidly with input size, with a clear quadratic relationship between *N* and execution time. This means that the algorithm has a polynomial time complexity of O(N^2), indicating that the execution time increases quadratically with input size.



Similarly, a cubic function also generated, which is represented by the equation y = ax^3 + bx^2 + cx + d. The function is fitted to the data using the same process as before, and the resulting curve shows that the execution time increases even more rapidly with input size than the quadratic function. This indicates that the algorithm has a polynomial time complexity of O(N^3), meaning that the execution time increases cubicly with input size.



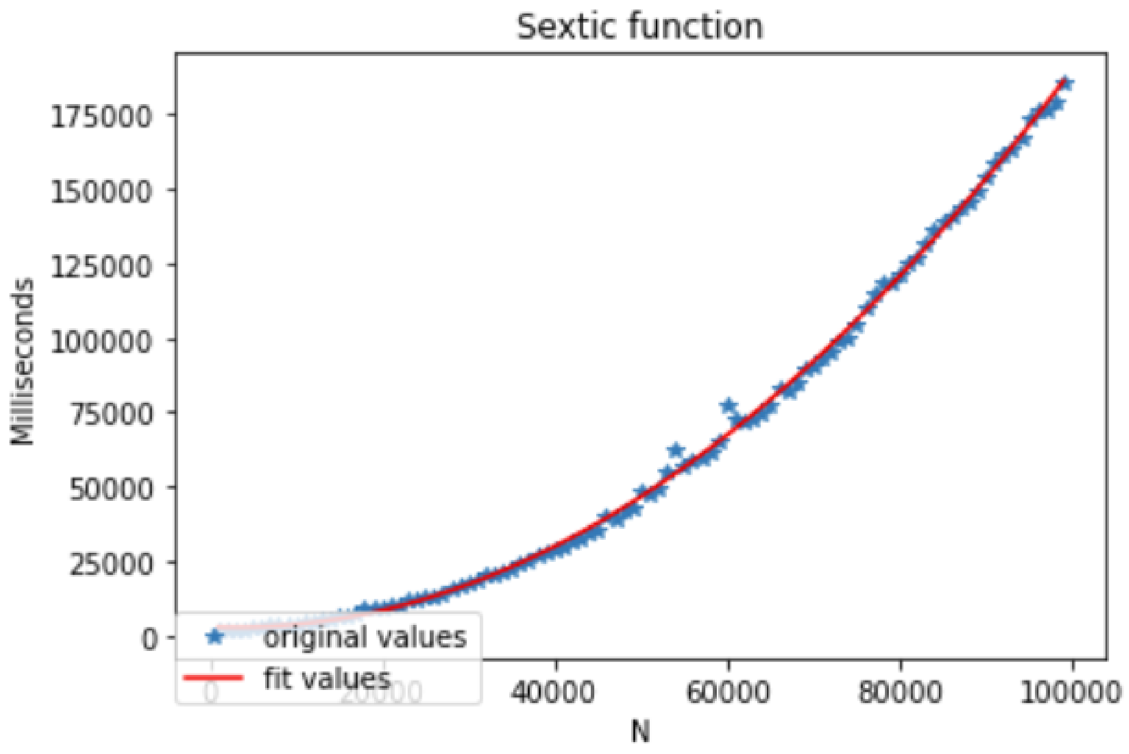
The significance of these findings lies in their implications for algorithm design and optimization. If an algorithm has a polynomial time complexity of O(N^2) or higher, it may not be suitable for processing large datasets or real-time applications where speed is crucial. In such cases, it may be necessary to redesign the algorithm to reduce its time complexity or implement parallel processing techniques to improve efficiency. Furthermore, an interesting point here in this analysis is that both quadratic function and cubic function fit well with the original curve, and the next step for digging more in this phenomenon should be the evaluation and optimization process.

Evaluation & Optimization

When applying a polynomial analysis, the basic idea is to find a function curve that approximates the real curve infinitely. Such function can be represented as $y(x, W) = w_0 + w_1x + w_2x^2 + \ldots + w_kx^k = \sum_{i=1}^{k} w_ix^i$, where k is the function degree and &w_{0},...,w_{k}& are polynomial coefficients that being recorded as W. In this section, we will evaluate the fitted polynomial using the mean squared error as the error function and find the best fitted function degree. Meanwhile, the &R^2& will be involved and the regularization will be used simultaneously to improve overfitting.

```
from sklearn.linear_model import Ridge
from sklearn.metrics import r2_score
from sklearn.preprocessing import PolynomialFeatures
import polyfit
class PolynomialFitting:
    def fitting(x_list,y_list,degree=2):
        if not isinstance(x_list, list):
            raise Exception(print("X axis error"))
        if not isinstance(y_list, list):
            raise Exception(print("Y axis error"))
        if not isinstance(degree, int):
            raise Exception(print("Degree error"))
        try:
            x_array = np.array(x_list)
            y_array = np.array(y_list)
        except:
            raise Exception(print("axis convert error"))
        try:
            w = np.polyfit(x_array, y_array, degree)
        except:
            raise Exception("Polynomial degree too high")
        f = PolynomialFitting.get_fx(w)
        r2 = PolynomialFitting.get_r2(y_array, x_array, f)

        return w, r2, f

    def fitting_with_lambda(x_list,y_list,degree=2,lambda_=0.001):
        if not isinstance(x_list, list):
            raise Exception(print("X axis error"))
        if not isinstance(y_list, list):
            raise Exception(print("Y axis error"))
        if not isinstance(degree, int):
            raise Exception(print("Degree error"))
        if not isinstance(lambda_, float):
            raise Exception(print("lambda error"))
        if lambda_<=0.0:
            raise Exception(print("invalid lambda"))
        try:
            x_array = np.array([x_list])
            y_array = np.array([y_list])
        except:
            raise Exception(print("axis convert error"))
        x_array = x_array.T
        y_array = y_array.T

        poly = PolynomialFeatures(degree=degree)
        x_list_poly = poly.fit_transform(x_array)

        lr = Ridge(alpha=(lambda_/2))
        lr.fit(x_list_poly,y_array)
        w=lr.coef_[0]
        w_1 = w.tolist()
        w_1.reverse()
        w=np.array(w_1)
        f = PolynomialFitting.get_fx(w)
        r2 = PolynomialFitting.get_r2(y_list, x_list, f)

        return w, r2, f

    def print_polynomial(w_list):
        fx="y = "
        for i in range(0,len(w_list)):
            param = w_list[i]
            order = len(w_list)-1-i
            if order:
                fx += "{} * x ^ {} + ".format(param, order)
            else:
                fx += "{}".format(param)
        return fx

    def get_fx(w_list):
        f = np.poly1d(w_list)
        return f

    def get_r2(y_ture, x_ture, f):
        r2 = r2_score(y_ture, f(x_ture))
        return r2

    def get_best_fitting(x_list, y_list):
        degree =1
        best_degree =1
        w_r, r2_r, f_r = PolynomialFitting.fitting(x_list,y_list,degree)
        while True:
            try:
                w,r2,f=PolynomialFitting.fitting(x_list,y_list,degree)
                print("Degree:{}\tIndex:{}".format(degree, r2))

                if r2<=0 or r2>1:
                    break
                if w[0]==0:
                    degree +=1
                    continue
                if r2> r2_r:
                    w_r =w
                    r2_r = r2
                    f_r = f
                    best_degree = degree
                degree += 1
            except:
                break
        print("Result")
        return w_r,r2_r,f_r,best_degree

    def get_best_fitting_with(x_real, y_real):
        best_degree =2
        w_r, r2_r, f_r = PolynomialFitting.fitting_with_lambda(x_real, y_re
        lambda_r =np.exp(0)

        for degree in range(2,10):
            for i in range (0, -19, -1):
                lambda_=np.exp(i)
                w, r2,f = PolynomialFitting.fitting_with_lambda(x_real, y_r
                print("Degree:{},lambda: {}".format(degree, lambda_))

                if r2> r2_r:
                    w_r =w
                    r2_r = r2
                    f_r = f
                    best_degree = degree
                    lambda_r =lambda_
        return w_r,r2_r,f_r,best_degree, lambda_r
```

As a result, it finds that the best fitting function degree should be 6 with R^2 as 0.9989.



Conclusion

In conclusion, the analysis of the polynomial time complexity relationship between input size and execution time provides valuable insights into the performance of algorithms and the design of efficient computational systems, which gives us a idea for the reason why the large frame may cause breakdown when using `dropna()` command. By understanding this relationship and using tools such as polynomial regression to analyze data, developers and researchers may be capible for further optimizing algorithms and improving computational efficiency in a wide range of applications.