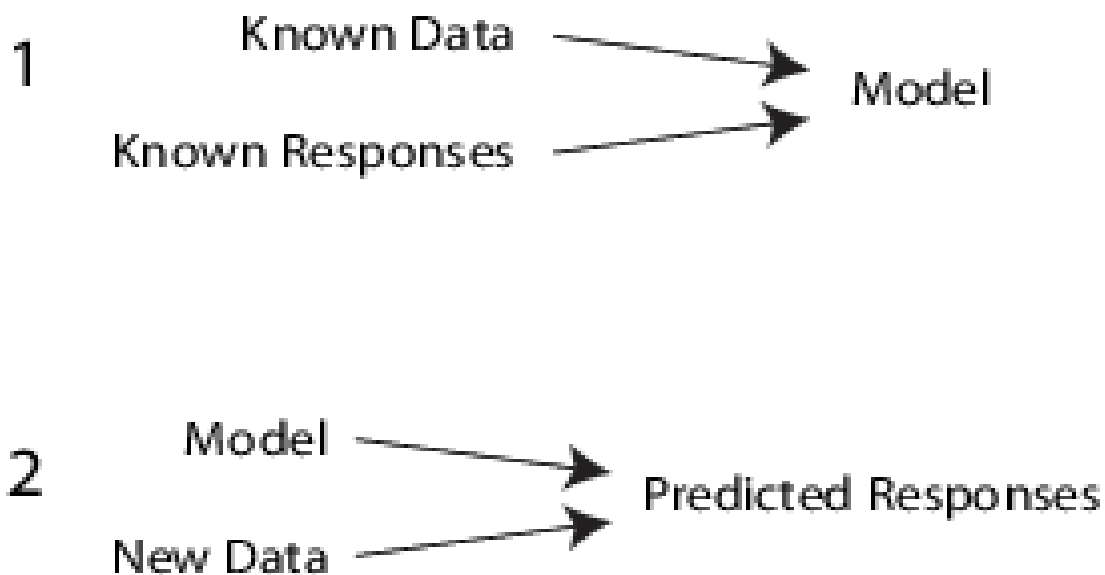


# Machine learning value estimation project

source code: [https://github.com/Sapien3/ML\\_value\\_estimation.git](https://github.com/Sapien3/ML_value_estimation.git)

**Introduction:** Using the dataset of multiple countries through index, I was requested to build a machine that's takes certain dimensions (health, education, living standards) in the form of inputs to perform on them an optimization algorithm predicting the percentage of deprivation on those countries.

This project is based heavily on pandas and scikitlearn libraries that compute some of their functionality like dataframe representation, array computing, dimensionality reduction, parametric estimation, and grid search to reform the best optimization code possible for a supervised learning machine.



**Dataset:** I downloaded the dataset from <http://kaggle.com/> , 20mb of recorded data from poor countries all over the world containing the dimensions we were speaking of.

I had a hard time with those data, it was hard to manipulate, there was a non Ascii symbols that's spyder editor(python) which cannot encoded, all numbers start with spaces, this is considered an NaN (not a number) .

## Try Again

```
>>> model.fit(X, y)
```

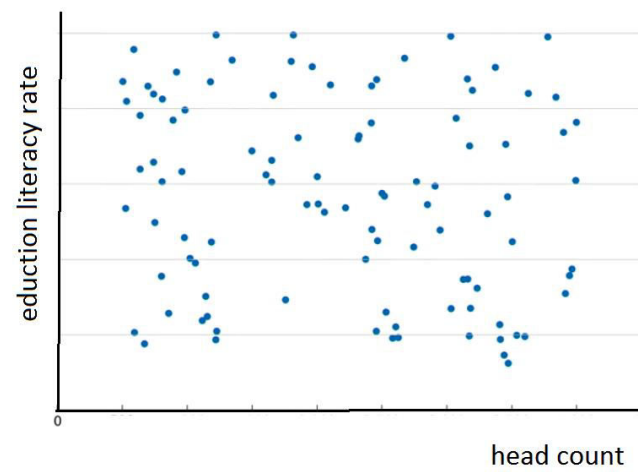
```
Traceback (most recent call last):
```

```
...
ValueError: Input contains NaN, infinity or a value
too large for dtype('float32').
```

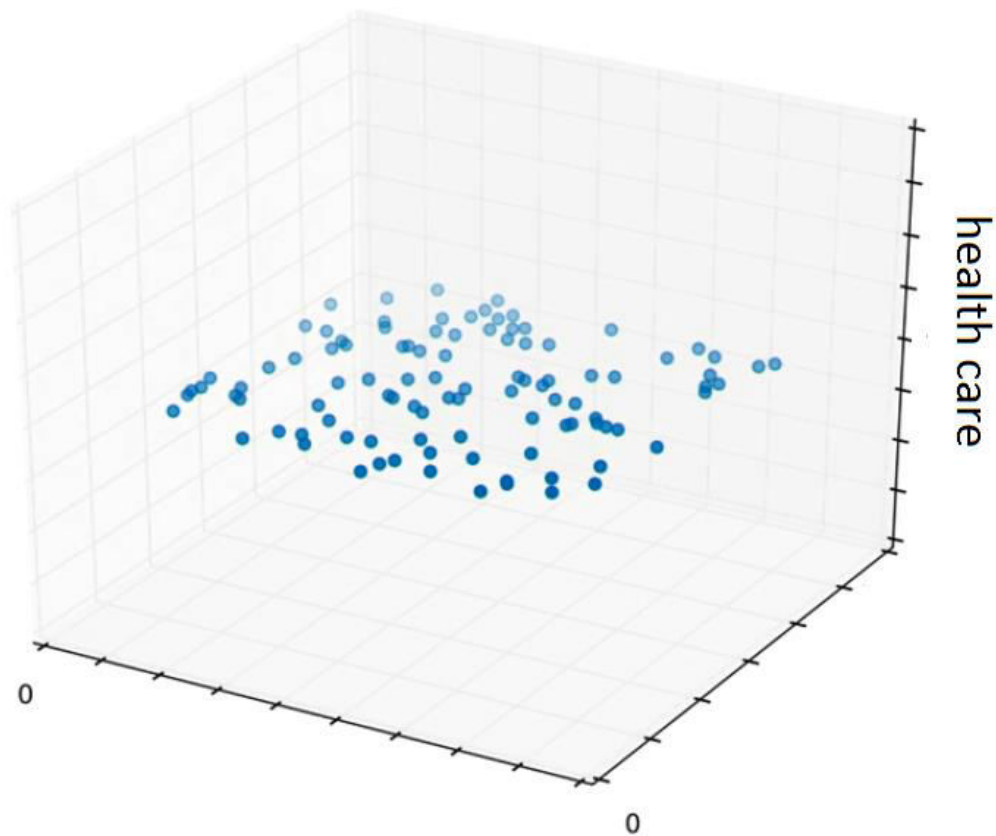


The major problem was the huge number of features relative to my low laptop capability of computing, this problem is called the curse of dimensionality, referring to various phenomena that arise when analyzing and organizing data.

Let's visualize our problem: Here we've graphed the size and value of 100 country region. With 100 data points we have pretty good coverage of all areas of the graph

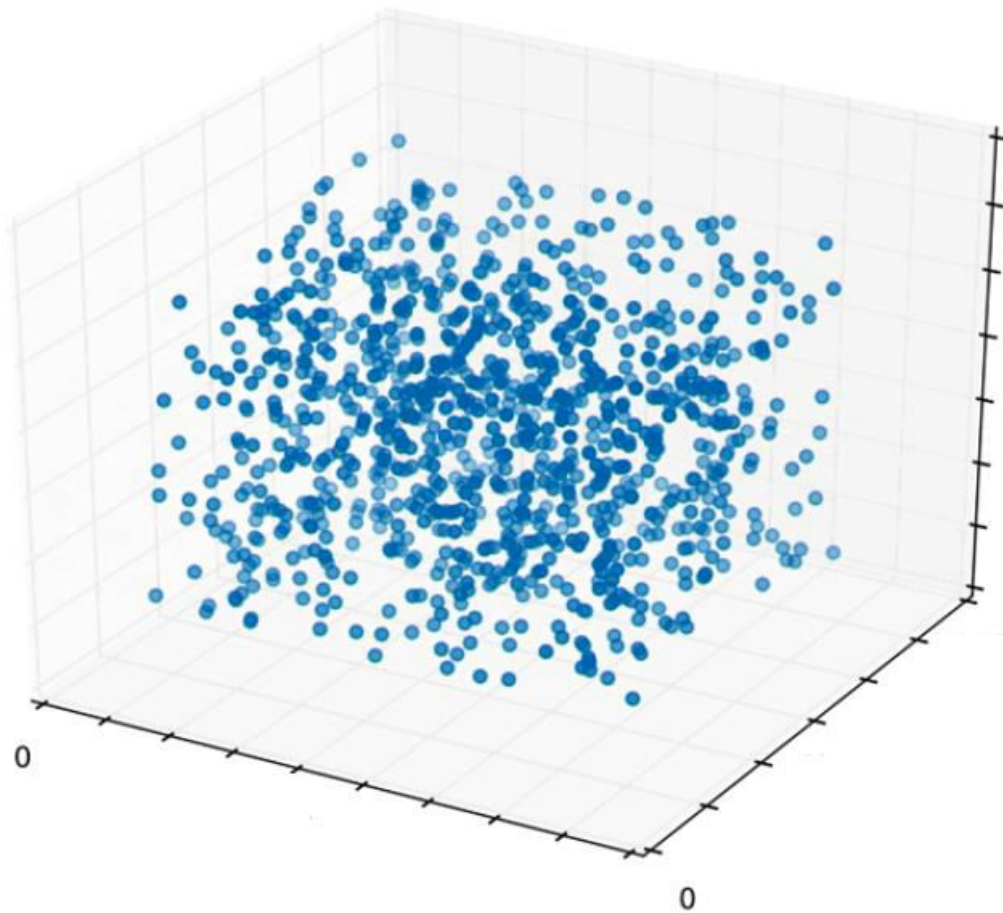


Now, let's add another dimension to our data set. Let's say that our deprivation predictor uses education literacy rate , health care ,and headcount.



We still have the same 100 data points as before but now that we've added another feature, or dimension, those 100 data points only cover a small part of the total possible area in the graph. To get good coverage of all possible areas on this graph, now we need 10 times as much data as with our one dimensional region index predictor.

In other words, as we add more features to our machine learning system, we need an exponential amount of rows of data to cover that increased space. When we start getting the hundreds and thousands of features the amount of data and the time required to train the system can be prohibitive.

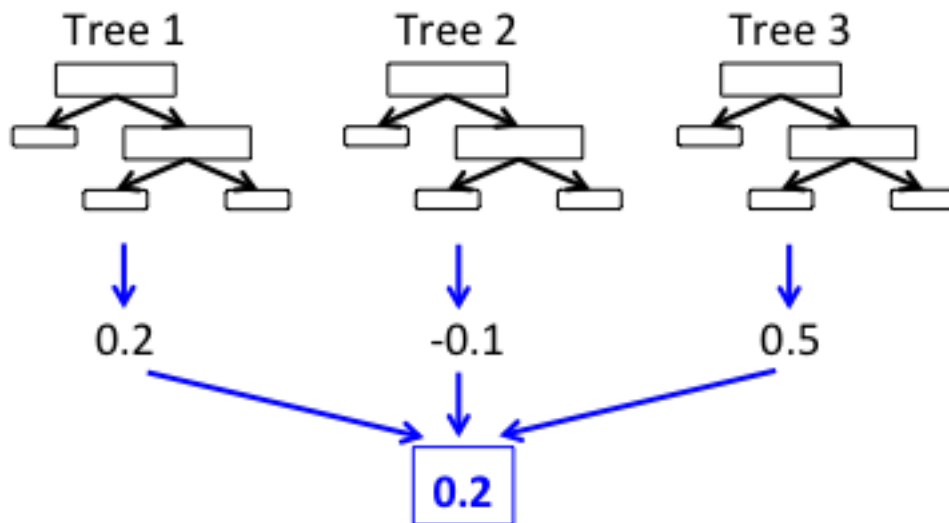


**My approach:** the main reason of importing scikitlearn library is to perform their boosting trees for regression and classification built methods.

The general idea is to compute a sequence of (very) simple trees, where each successive tree is built for the prediction residuals of the preceding tree. This method will build binary trees, partition the data into two samples at each split node. Now suppose that you were to limit the complexities of the trees to 3 nodes only: a root node and two child nodes, a single split. Thus, at each step of the boosting (boosting trees algorithm), a simple (best) partitioning of the data is determined, and the deviations of the observed values from the respective means (residuals for each partition) are computed. The next 3-node tree will then be fitted to those residuals, to find another partition that

will further reduce the residual (error) variance for the data, given the preceding sequence of trees

### Ensemble Model: example for regression



As a parametric estimation model, I choose the "Ls" function to compute loss, instead of some good loss-function like "huber", the main reason is the powerful and efficient math representation:

For a scalar parameter  $\theta$ , a decision function whose output  $\hat{\theta}$  is an estimate of  $\theta$ , and a quadratic loss function

$$L(\theta, \hat{\theta}) = (\theta - \hat{\theta})^2,$$

the risk function becomes the [mean squared error](#) of the estimate,

$$R(\theta, \hat{\theta}) = \mathbb{E}_{\theta}(\theta - \hat{\theta})^2.$$

In [density estimation](#), the unknown parameter is [probability density](#) itself. The loss function is typically chosen to be a [norm](#) in an appropriate [function space](#). For example, for  $L^2$  norm,

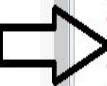
$$L(f, \hat{f}) = \|f - \hat{f}\|_2^2,$$

the risk function becomes the [mean integrated squared error](#)

$$R(f, \hat{f}) = \mathbb{E} \|f - \hat{f}\|^2.$$

**Improvement on our existing module:** I tried to minimize the features as best as I could, thus instead of using a discrete features, MPI index from Oxford University offer representation for the accumulation of some of these features in one, known as MPI. Then, instead of using 6 features we end up using only 3, and one of them for output which is the deprivation ration. To be most certain of the decision of my parameters, I coded a grid search program that's cost me 2 hour of computing some of my tested parameters, the result are:

```
33 'max_depth': [4, 6],
34 'min_samples_leaf': [3, 5, 9, 17],
35 'learning_rate': [0.1, 0.05, 0.02, 0.01],
36 'max_features': [1.0, 0.3, 0.1],
37 'loss': ['ls', 'lad', 'huber']
38 }
39
40 # Define the grid search we want to run. Run it with four cpus in parallel.
41 gs_cv = GridSearchCV(model, param_grid, n_jobs=4)
42
43 # Run the grid search - on only the training data!
44 gs_cv.fit(X_train, y_train)
45
46 # Print the parameters that gave us the best result!
47 print(gs_cv.best_params_)
48
49 # After running a ....long.... time, the output will be something like
50 # {'loss': 'huber', 'learning_rate': 0.1, 'min_samples_leaf': 9, 'n_estimators': 3000, 'max_features': 0.1, 'max_depth': 4}
51
52 # That is the combination that worked best!
53
54 # Find the error rate on the training set using the best parameters
55 mse = mean_absolute_error(y_train, gs_cv.predict(X_train))
56 print("Training Set Mean Absolute Error: %.4f" % mse)
57
58 # Find the error rate on the test set using the best parameters
59 mse = mean_absolute_error(y_test, gs_cv.predict(X_test))
60 print("Test Set Mean Absolute Error: %.4f" % mse)
61
62
```



```
trained_Intensity of deprivation Regional_classifier_model.pkl 5.5
view_data.py 477 b

IPython console
Console 1/A
y, test_size=0.3, random_state=0)
...:
...: # Fit regression model
...: model = ensemble.GradientBoostingRegressor(
...:     n_estimators=3000,
...:     learning_rate=0.1,
...:     max_depth=4,
...:     min_samples_leaf=5,
...:     max_features=1,
...:     loss='ls',
...:     random_state=0
...: )
...: model.fit(X_train, y_train)
...:
```

Finally I had an improvement of 16% of my initial model with an astonishing accuracy of 99.3%.

Check the code!