Linear Regression

Due: Friday, Nov 15 at 4pm

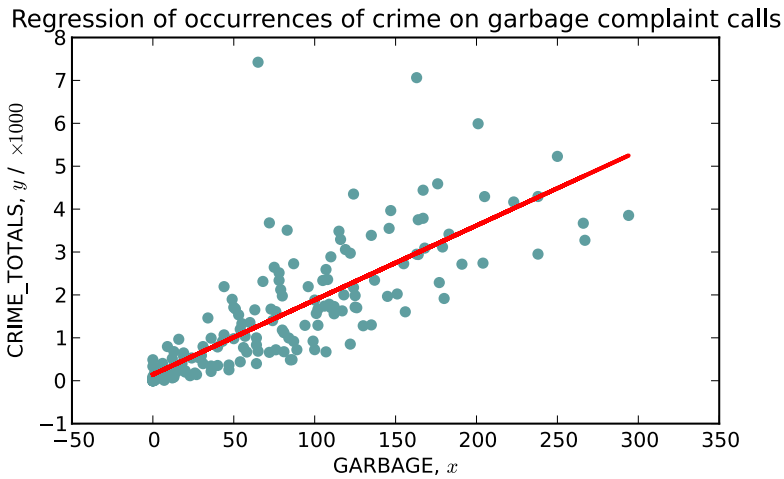
In this assignment, you will fit linear regression models and implement a few simple variable selection algorithms. The assignment will give you experience with NumPy and more practice with using classes and functions to support code reuse.

You must work alone on this assignment.

Introduction

At the heart of the assignment is a table, where each column is a variable and each row is a sample unit. As an example, in a health study, each sample unit might be a person, with variables like height, weight, sex, etc. In your analysis, you will build models that, with varying levels of accuracy, can predict the value of one of the variables as a function of the others.

Predictions are only possible if variables are related somehow. As an example, look at this plot of recorded crimes against logged complaint calls about garbage to 311.



Each point describes a sample unit, which in this example represents a geographical region of Chicago. Each region is associated with variables, such as the number of crimes or complaint calls during a fixed time frame. Given this plot, if you got the question of how many crimes you think were recorded for a region that had 150 complaint calls about garbage, you would follow the general trend and probably say something like 3000 recorded crimes. To formalize this prediction, we need a model for the data that relates a *dependent variable* (e.g., crimes) to a set of *predictor variables* (e.g., complaint calls). Our model will assume a linear dependence.

To make this precise, we will use the following notation:

N
: the total number of sample units.

K
: the total number of predictor variables. In the example above, $K = 1$.

n
: the sample unit that we are currently considering (an integer from 0 to $N - 1$).

x_{nk}
: an observation of a *predictor variable* k for sample unit n, e.g., the number of complaint calls about garbage.

y_n
: an observation of the *dependent variable* for sample unit n, e.g., the total number of crimes.

$\hat{y}_n$
: our prediction for the dependent variable for sample unit n, based on our observation of the predictor variables. This value corresponds to a point on the red line.

$\varepsilon_n = y_n - \hat{y}_n$
: The *residual* or *observed error*, that is, the difference between the actual observed value of the dependent variable, and our prediction for it. Ideally, our predictions would match the observations, so that ε_n would always be zero. In practice, there will be some discrepancy, for two reasons. For one, when we make predictions on new data, we will not have access to the observations of the dependent variable. But also, our model will assume a linear dependence between the predictor variables and the dependent variable, while in reality the relationship will not be quite linear. So, even when we do have direct access to the observations of the dependent variable, we will not have ε_n equal to zero.

Our prediction for the dependent variable will be given by a linear equation:

$$\hat{y}_n = \beta_0 + \beta_1 x_{n1} + \cdots + \beta_K x_{nK}, \tag{1}$$

where the coefficients $\beta_0, \beta_1, \ldots, \beta_K$ are real numbers. We would like to select values for these coefficients that result in small residuals ε_n.

We can rewrite this equation more concisely using vector notation. We define:

$\boldsymbol{\beta} = \begin{pmatrix} \beta_0 & \beta_1 & \beta_2 & \cdots & \beta_K \end{pmatrix}^T$
: a column vector of the regression coefficients, where β_0 is the intercept and β_k (for $1 \leq k \leq K$) is the coefficient associated with the kth predictor. This vector describes the red line in the figure above. Note that a positive value of a coefficient suggests a positive correlation with the dependent variable. The same is true for a negative value and a negative correlation.

$\mathbf{x}_n = \begin{pmatrix} 1 & x_{n1} & x_{n2} & \cdots & x_{nK} \end{pmatrix}^T$
: a column vector representation of all the predictors for a given sample unit. Note that a 1 has been prepended to the vector. This will allow us to rewrite equation (1) in vector notation without having to treat β_0 separately from the other coefficients β_k.

We can then rewrite equation (1) as:

$$\hat{y}_n = \mathbf{x}_n^T \boldsymbol{\beta}. \tag{2}$$

This equation can be written for all sample units at the same time using matrix notation. We define:

$\mathbf{y} = \begin{pmatrix} y_0 & y_1 & \cdots & y_{N-1} \end{pmatrix}^T$
: a column vector of observations of the dependent variable.

$\hat{\mathbf{y}} = \begin{pmatrix} \hat{y}_0 & \hat{y}_1 & \cdots & \hat{y}_{N-1} \end{pmatrix}^T$
: a column vector of predictions for the dependent variable.

$\boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_0 & \varepsilon_1 & \cdots & \varepsilon_{N-1} \end{pmatrix}^T$
: a column vector of the residuals (observed errors).

$\mathbf{X}$
: an $N \times (K+1)$ matrix where each row is one sample unit. The first column of this matrix is all ones.

We can then write equations (1) and (2) for all sample units at once as

$$\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta}, \tag{3}$$

And, we can express the residuals as

$$\boldsymbol{\varepsilon} = \mathbf{y} - \hat{\mathbf{y}}. \tag{4}$$

Matrix multiplication:

> Equations (2) and (3) above involve *matrix multiplication*. If you are unfamiliar with matrix multiplication, you will still be able to do this assignment. Just keep in mind that to make the calculations less messy, the matrix $\mathbf{X}$ contains not just the observations of the predictor variables, but also an initial column of all ones. The data we provide does not yet have this column of ones, so you will need to prepend it.

Model fitting

There are many possible candidates for $\boldsymbol{\beta}$, some that fit the data better than others. Finding the best value of $\boldsymbol{\beta}$ is referred to as *fitting the model*. For our purposes, the "best" value of $\boldsymbol{\beta}$ is the one that minimizes the residuals $\boldsymbol{\varepsilon} = \mathbf{y} - \hat{\mathbf{y}}$ in the least-squared sense. That is, we want the value for $\boldsymbol{\beta}$ such that the predicted values $\hat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta}$ are as close to the observed values $\mathbf{y}$ as possible (in a statistically-motivated way using maximum likelihood). We will provide a function that computes this value of $\boldsymbol{\beta}$; see "The `linear_regression` function" below.

Getting started

We have seeded your repository with a directory for this assignment. To pick it up, change to your `capp30121-aut-19-username` directory (where the string `username` should be replaced with your username) and then run the command: `git pull upstream master`. You should also run `git pull` to make sure your local copy of your repository is in sync with the server.

The `pa5` directory contains the following files:

`regression.py`:	Python file where you will write your code.
`util.py`:	Python file with several helper functions, some of which you will need to use in your code.
`output.py`:	This file is described in detail in the "Testing your code" section below.
`test_regression.py`:	Python file with the automated tests for this assignment.

The `pa5` directory also contains a `data` directory which, in turn, contains two sub-directories: `city` and `stock`.

Data

In this assignment you will write code that can be used on many different datasets. We have provided a couple of sample datasets for testing purposes.

City:	Predicting crime rate from the number of complaint calls to 311.
Stock:	Predicting a stock index from a set of stocks.

More information about each dataset can be found by clicking the links above.

For this assignment, a dataset is stored in a directory that contains two files: a CSV (comma-separated values) file called `data.csv` and a JSON file called `parameters.json`. The CSV file contains a table where each column corresponds to a variable and each row corresponds to a sample unit. The first row contains the column names. The JSON file contains a few parameters:

- `name`: the name of the dataset,
- `predictor_vars`: the column indices of the predictor variables,
- `dependent_var`: the column index of the dependent variable,
- `training_fraction`: the fraction of the data that should be used to train models, and
- `seed`: a random number generator seed.

The last two parameters will be used to help split the data into two sets: one that you will use to fit or *train* models and one that you will use to evaluate how well different models predict outcomes on new-to-the-model data. We describe this process below.

Task 0: The `DataSet` and `Model` classes

In this assignment, you will be working with datasets and models, so it will make sense for us to have a `DataSet` class and a `Model` class, which you will implement in `regression.py` (we have provided basic skeletons for those classes in that file)

This task describes the requirements for the `DataSet` and `Model` classes and, while you should give some thought to the design of these classes before moving on to the rest of the assignment, it is very likely you will have to reassess your design throughout the assignment (and you should not get discouraged if you find yourself doing this). We suggest you start by writing a first draft of the `DataSet` and `Model` classes, and then fine-tune them as you work through the rest of the assignment.

Representing Datasets

Recall that each dataset has two files: a CSV file with the data to be used for building our models and a JSON file with the parameters for the assignment. In the file `util.py`, we have provided two functions: one to read the sample data from the CSV file (`load_numpy_array`) and one to read the parameters from the JSON file (`load_json_file`). The data array returned by `load_numpy_array` does *not* include the initial column of all ones that is found in the matrix $\mathbf{X}$ described in the introduction. You can prepend a column of all ones to a two-dimensional array using the function `prepend_ones_column` provided in `util.py`.

The `DataSet` class will be very simple, and should only require a constructor that takes the name of the directory holding the dataset as its only parameter. In addition to saving the values necessary for the computation from the JSON and CSV files (dataset name, predictor variable indices, dependent variable index, and the column labels), your constructor should also split the sample data into two NumPy arrays: one, called the training data, that you will use to construct models in Tasks 1–4, and another, called the testing data, that you will use in Task 5 to evaluate how well different models predict new-to-the-model data.

Why not use all the data to fit the models? It is easy to fit a model that does a good job of predicting the dependent variable for the sample units that were used to train it, but does a terrible job of predicting the dependent variable for other data. How do we determine how well a model does with new-to-it data? We can hold out some of the data and use the model to predict the values of the dependent variable for the held-out sample units. Since we know the actual value of the dependent variable for this held-out data, we can use it to evaluate the effectiveness of the model on previously-unseen data.

We will use the train_test_split function from the Scikit-learn model selection library to make the split. This method takes a NumPy array and splits it into two arrays: one to use for training models and another to use for testing them. It has two keyword parameters that can be used to control the size of each array: `train_size` and `test_size`. We'll set `train_size` using the `training_fraction` parameter from the parameters file and we will set the `test_size` parameter to `None`. Using `None` for the `test_size` parameter indicates that the rows not chosen for the training set should be included in the test set, which is the default behavior. Note that you could omit this parameter, but the function generates an annoying warning message if you do.

The `train_test_split` method decides randomly which rows to include in which set. To support repeatability and simplify testing, it includes a `random_state` parameter that can be used to set the seed for the underlying random process. Like previous assignments, this will ensure that, from a given seed, the random process will select the same rows to be assigned to the same sets each time you run your program. We'll set this parameter using the `seed` parameter from the JSON file.

Your implementation of the `DataSet` constructor should be quite simple. Ours is less than 10 lines of code. The heavy lifting is done in the functions we provide in `util.py`, as well as the `train_test_split` method from Scikit-learn.

Representing models

Implementing a `Model` class will allow us to write functions that take `Model` objects as parameters, and which can return `Model` objects as well, instead of having to pass around lists or dictionaries with the information for a given model.

Each model will be constructed from a dataset and a subset of the dataset's predictor variables. We encourage you to think carefully about what attributes and methods to include in your class.

Take into account that, for the tests to work, your `Model` class must have at least the following public attributes (don't worry if you don't understand what these mean right now; they will become clearer in subsequent tasks):

- `R2`: The value of R^2 for the model
- `adj_R2`: The value of the Adjusted R^2 for the model
- `dep_var`: The index of the dependent variable.
- `pred_vars`: A list of integers containing the indices of the predictor variables.
- `beta`: A NumPy array with the model's $\boldsymbol{\beta}$

As we will describe later, it can also be very helpful to include a `__repr__` method.

A note on array dimensions:

The parameter of the function `prepend_ones_column` (as well as the first parameter to the functions `linear_regression` and `apply_beta` described below) is required to be a two-dimensional array, even when that array only has one column. NumPy makes a distinction between one-dimensional and two-dimensional arrays, even if they contain the same information. For instance, if we want to extract the second column of `A` as a one-dimensional array, we do the following:

```
>>> A
array([[5. , 2. ],
       [3. , 2. ],
       [6. , 2.1],
       [7. , 3. ]])
>>> A[:, 1]
array([2. , 2. , 2.1, 3. ])
>>> A[:, 1].shape
(4,)
```

The resulting shape will not be accepted by `prepend_ones_column`. To retrieve a 2D column subset of `A`, you can use a list of integers as the index. This mechanism keeps `A` two-dimensional, even if the list of indices has only one value:

```
>>> A[:, [1]]
array([[2. ],
       [2. ],
       [2.1],
       [3. ]])
>>> A[:, [1]].shape
(4, 1)
```

In general, you can specify a slice containing a specific subset of columns as a list. For example, let `Z` be:

```
>>> Z = np.array([[1, 2, 3, 4], [11, 12, 13, 14], [21, 22, 23, 24]])
>>> Z
array([[ 1,  2,  3,  4],
       [11, 12, 13, 14],
       [21, 22, 23, 24]])
```

Evaluating the expression `Z[:, [0, 2, 3]]` will yield a new 2D array with columns 0, 2, and 3 from the array `Z`:

```
>>> Z[:, [0, 2, 3]]
array([[ 1,  3,  4],
       [11, 13, 14],
       [21, 23, 24]])
```

or more generally, we can specify any expression for the slice that yields a list:

```
>>> l = [0, 2, 3]
>>> Z[:, l]
array([[ 1,  3,  4],
       [11, 13, 14],
       [21, 23, 24]])
```

The `linear_regression` function

We are interested in the value of $\boldsymbol{\beta}$ that best fits the data. To simplify your job, we have provided code for fitting a linear model. The function `linear_regression` in `util.py` finds the best value of $\boldsymbol{\beta}$ as described in the introduction. This function accepts as input a two-dimensional NumPy array of floats `X` containing the observations of the predictor variables ($\mathbf{X}$) and a one-dimensional NumPy array of floats `y` containing the observations of the dependent variable ($\mathbf{y}$). It returns $\boldsymbol{\beta}$ as a one-dimensional NumPy array `beta`.

We have also provided a function, `apply_beta(beta, X)` for making predictions using the model. It takes `beta`, the array generated by our linear regression function, and `X` as above, applies the linear function described by `beta` to each row in `X`, and returns a one-dimensional NumPy array of the resulting values (that is, it returns $\mathbf{X}\boldsymbol{\beta}$).

Here are sample calls to `linear_regression` and `apply_beta`:

```
In [1]: import numpy as np

In [2]: from util import *

In [3]: predictors = np.array([[5, 2], [3, 2], [6, 2.1], [7, 3]]) # predictors

In [4]: X = prepend_ones_column(predictors)

In [5]: X
Out[5]:
array([[1. , 5. , 2. ],
       [1. , 3. , 2. ],
       [1. , 6. , 2.1],
       [1. , 7. , 3. ]])

In [6]: y = np.array([5, 2, 6, 6]) # observations of dependent variable

In [7]: beta = linear_regression(X, y)

In [8]: beta # yhat_n = 1.2 + 1.4 * x_n1 - 1.7 * x_n2
Out[8]: array([ 1.20104895,  1.41083916, -1.6958042 ])

In [9]: apply_beta(beta, X)
Out[9]: array([4.86363636, 2.04195804, 6.1048951 , 5.98951049])
```

Prepending a column of ones:

In the above example, we have two predictor variables, stored in the array `predictors`. However, to be able to call `linear_regression` and `apply_beta`, we need to prepend a column of all ones to the array containing the predictor variables.

The mathematical reason for this is described in the introduction (it is related to the β_0 intercept), although it is possible to complete the assignment without fully understanding the reason. Regardless, you have to be aware that the data returned by `load_numpy_array` does *not* include that columns of ones.

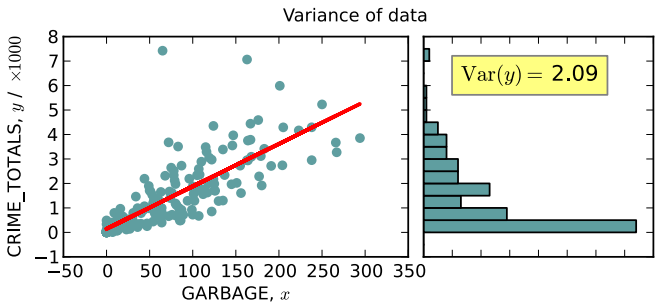
This means that, if you load a dataset with `load_numpy_array`, and then extract certain columns to create a model with those variables, those columns by themselves will not be the $\mathbf{X}$ array. You

need to use the `prepend_ones_column` function (as shown above) to obtain the `X` array that you can use to call `linear_regression` and `apply_beta`.

Task 1: Model evaluation using the Coefficient of Determination (R^2)

Blindly regressing on all available predictors is bad practice and can lead to *over-fit* models. Over-fitting happens when your model picks up on random structure in the training data that does not represent the underlying phenomenon. This situation is made even worse if some of your predictors are highly correlated (see multicollinearity), since it can ruin the interpretation of your coefficients. As an example, consider a study that looks at the effects of smoking and drinking on developing lung cancer. Smoking and drinking are highly correlated, so the regression can arbitrarily pick up either as the explanation for high prevalence of lung cancer. As a result, drinking could be assigned a high coefficient while smoking a low one, leading to the questionable conclusion that drinking causes lung cancer while smoking does not.

This example illustrates the importance of variable selection. In order to compare two different models (subsets of predictors) against each other, we need a measurement of their goodness of fit, *i.e.*, how well they explain the data. A popular choice is the coefficient of determination, R^2, which first considers the variance of the dependent variable, $\mathrm{Var}(\mathbf{y})$:

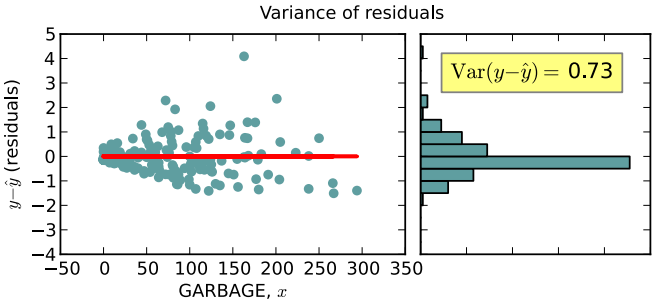


In this example we revisit the regression of the total number of crimes, y_n, on the number of calls about garbage to 311, x_n. To the right, a histogram over the y_n is shown and their variance is calculated. The variance is a measure of how spread out the data is. It is calculated as follows:

$$\mathrm{Var}(\mathbf{y}) = \frac{1}{N}\sum_{n=1}^{N}(y_n - \bar{y})^2, \tag{5}$$

where $\bar{y}$ denotes the mean of all of the y_n.

We now subtract the red line from each point. These new points are the residuals and represent the deviations from the predicted values under our model. If the variance of the residuals, $\mathrm{Var}(\mathbf{y} - \hat{\mathbf{y}})$, is zero, it means that all residuals had to be zero and thus all sample units lie perfectly on the fitted line. On the other hand, if the variance of the residuals is the same as the variance of $\mathbf{y}$, it means that the model did not explain anything away and we can consider the model useless. These two extremes represent R^2 values of 1 and 0, respectively. All R^2 values in between these two extremes will be cases where the variance was reduced some, but not entirely. Our example is one such case:



As we have hinted, the coefficient of determination, R^2, measures how much the variance was reduced by subtracting the model. More specifically, it calculates this reduction as a percentage of the original variance:

$$R^2 = \frac{\mathrm{Var}(\mathbf{y}) - \mathrm{Var}(\mathbf{y} - \hat{\mathbf{y}})}{\mathrm{Var}(\mathbf{y})} = 1 - \frac{\mathrm{Var}(\mathbf{y} - \hat{\mathbf{y}})}{\mathrm{Var}(\mathbf{y})}. \tag{6}$$

which finally becomes:

$$R^2 = 1 - \frac{\sum_{n=1}^{N}(y_n - \hat{y}_n)^2}{\sum_{n=1}^{N}(y_n - \bar{y})^2}. \tag{7}$$

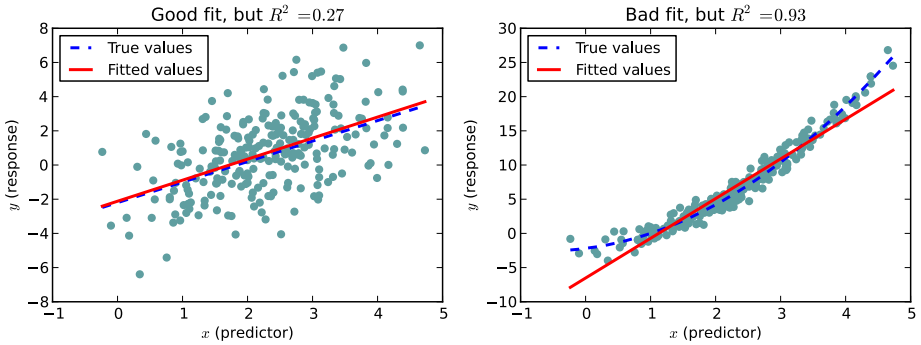
In equation (7) we omitted the normalization constants N since they cancel each other out. We also did not subtract the mean when calculating the variance of the residuals, since it can be shown mathematically that if the model has been fit by least squares, the sum of the residuals is always zero.

There are two properties of R^2 that are good to keep in mind when checking the correctness of your calculations:

1. $0 \leq R^2 \leq 1$.
2. If model A contains a superset of predictors of model B, then the R^2 value of model A is greater than or equal to the R^2 value of model B.

These properties are only true if R^2 is computed using the same data that was used to train the model parameters. If you calculate R^2 on a held-out testing set, the R^2 value could decrease with more predictors (due to over-fitting) and R^2 is no longer guaranteed to be greater than or equal to zero. Furthermore, equations (6) and (7) would no longer be equivalent (and the intuition behind R^2 would need to be reconsidered, but we omit the details here). You should always use equation (7) to compute R^2.

A good model should explain as much of the variance as possible, so we will be favoring higher R^2 values. However, since using all predictors gives the highest R^2, we must balance this goal with a desire to use few predictors. In general it is important to be cautious and not blindly interpret R^2; take a look at these two generated examples:



This data was generated by selecting points on the dashed blue function, and adding random noise. To the left, we see that the fitted model is close to the one we used to generate the data. However, the variance of the residuals is still high, so the R^2 is rather low. To the right is a model that clearly is not right for the data, but still manages to record a high R^2 value. This example also shows that that linear regression is not always the right model choice (although, in many cases, a simple transformation of the predictors or the dependent variable can resolve this problem, making the linear regression paradigm still relevant).

Task 1 is divided into two subtasks, Task 1a and Task 1b.

Task 1a

You must implement this function in `regression.py`:

```
def compute_single_var_models(dataset):
    '''
    Computes all the single-variable models for a dataset

    Inputs:
        dataset: (DataSet object) a dataset

    Returns:
        List of Model objects, each representing a single-variable model
    '''
```

More specifically, given a dataset with P predictor variables, you will construct P univariate (*i.e.*, one variable) models of the dataset's dependent variable, one for each of the possible predictor variables (in the same order that the predictor variables are listed), and compute their associated R^2 values. Note that when we ask for the R^2 value for a model, we mean the value obtained by computing R^2 using the model's $\boldsymbol{\beta}$ on the data that was used to *train* it.

Most of the work in this task will be in your `Model` class. In fact, if you've designed `Model` correctly, your implementation of `compute_single_var_models` shouldn't be longer than three lines (ours is actually just one line long).

Also, notice how this function (and the next one we'll ask you to implement) doesn't return R^2. That value should be an attribute of your `Model`, so there's no need to return it as an additional return value.

Note:

You may **not** use the NumPy `var` method to compute R^2 for two reasons: (1) our computation uses the biased variance while the `var` method computes the unbiased variance and will lead you to generate the wrong answers and (2) we want you to get practice working with NumPy arrays.

Task 1b

You must implement this function in `regression.py`:

```
def compute_all_vars_model(dataset):
    '''
    Computes a model that uses all the predictor variables in the dataset

    Inputs:
        dataset: (DataSet object) a dataset

    Returns:
        A Model object that uses all the predictor variables
    '''
```

More specifically, you will construct a single model that uses all of the dataset's predictor variables to predict the dependent variable. According to the second property of R^2, the R^2 for this model should be the largest R^2 value you will see for the training data.

At this point, you should make sure that your code to calculate a model's R^2 value is general enough to handle models with multiple predictor variables (*i.e.*, multivariate models).

Take into account that, if you have a good design for `Model`, then implementing Task 1b should be simple after implementing Task 1a. If it doesn't feel that way, ask on Piazza or come to office hours so we can give you some quick feedback on your design.

Testing your code

As usual, you will be able to test your code from IPython and by using `py.test`. When using IPython, make sure to enable autoreload before importing the regression.py module:

```
In [1]: %load_ext autoreload

In [2]: %autoreload 2

In [3]: import regression
```

You could then test your Task 1a code by creating a `DataSet` object (let's assume we create a `dataset` variable for this) and running the following:

```
In [3]: univar_models = regression.compute_single_var_models(dataset)
```

`univar_models` should then contain a list of `Model` objects. However, checking the R^2 values manually from IPython can be cumbersome, so we have included a Python program, `output.py`, that will print out these (and other) values for all of the tasks. You can run it from the command-line like this:

```
$ python3 output.py data/city
```

If your implementation of Task 1a is correct, you will see this:

```
City Task 1a
    !!! You haven't implemented the Model __repr__ method !!!
    R2: 0.1402749161031348

    !!! You haven't implemented the Model __repr__ method !!!
    R2: 0.6229070858532733

    !!! You haven't implemented the Model __repr__ method !!!
    R2: 0.5575360783921093

    !!! You haven't implemented the Model __repr__ method !!!
    R2: 0.7831498392992615

    !!! You haven't implemented the Model __repr__ method !!!
    R2: 0.7198560514392482

    !!! You haven't implemented the Model __repr__ method !!!
    R2: 0.32659079486818354

    !!! You haven't implemented the Model __repr__ method !!!
    R2: 0.6897288976957778
```

You should be producing the same R^2 values shown above, but we can't tell what model they each correspond to! For `output.py` to be actually useful, you will need to implement the `__repr__` method in `Model`.

We suggest your string representation be in the following form: the name of the dependent variable followed by a tilde (~) and the regression equation with the constant first. If you format the floats to have six decimal places, the output for Task 1a will now look like this:

```
City Task 1a
    CRIME_TOTALS ~ 575.687669 + 0.678349 * GRAFFITI
    R2: 0.1402749161031348

    CRIME_TOTALS ~ -22.208880 + 5.375417 * POT_HOLES
    R2: 0.6229070858532733

    CRIME_TOTALS ~ 227.414583 + 7.711958 * RODENTS
    R2: 0.5575360783921093

    CRIME_TOTALS ~ 11.553128 + 18.892669 * GARBAGE
    R2: 0.7831498392992615

    CRIME_TOTALS ~ -65.954319 + 13.447459 * STREET_LIGHTS
    R2: 0.7198560514392482

    CRIME_TOTALS ~ 297.222082 + 10.324616 * TREE_DEBRIS
    R2: 0.32659079486818354

    CRIME_TOTALS ~ 308.489056 + 10.338500 * ABANDONED_BUILDINGS
    R2: 0.6897288976957778
```

For example, the first model uses only one variable, `GRAFFITI` to predict crime, and has an R^2 of only 0.14027. The last model uses only `ABANDONED_BUILDINGS`, and the higher R^2 (0.6897) tells us that `ABANDONED_BUILDINGS` is likely a better predictor than `GRAFFITI`.

Take into account that you can also run `output.py` with the Stock dataset:

```
$ python3 output.py data/stock
```

The full expected output of `output.py` can be found in the City and Stock pages. However, please note that *you do not need to check all this output manually*. We have also included automated tests that will do these checks for you. For example, in Task 1 you can run the following:

```
$ py.test -vk task1
```

Warning: No hard-coding!:

Since we only have two datasets, it can be very easy in some tasks to write code that will pass the tests by hard-coding the expected values in the function (and returning one or the other depending on the dataset that is being used)

If you do this, you will receive ZERO POINTS on your entire test score, regardless of whether other tasks are implemented correctly without hard-coding

Task 2: Building and selecting bivariate models

If you look at the output for Task 1a, none of the predictor variables individually are particularly good predictors for the dependent variable (they all had low R^2 values compared to when using all predictors). In this and subsequent tasks, we will construct better models using multiple variables without using all of them and risking over-fitting.

For example, we could predict crime using not only complaints about garbage, but graffiti as well. For example, we may want to find $\boldsymbol{\beta} = (\beta_0, \beta_{\text{GARBAGE}}, \beta_{\text{GRAFFITI}})$ in the equation

$$\hat{y}_n = \beta_0 + \beta_{\text{GARBAGE}} x_{n\ \text{GARBAGE}} + \beta_{\text{GRAFFITI}} x_{n\ \text{GRAFFITI}},$$

where $\hat{y}_n$ is a prediction of the number of crimes given the number of complaint calls about garbage ($x_{n\ \texttt{GARBAGE}}$) and graffiti ($x_{n\ \texttt{GRAFFITI}}$).

For this task, you will test all possible bivariate models ($K = 2$) and determine the one with the highest R^2 value. We suggest that you use two nested for-loops to iterate over all possible combinations of two predictors, calculate the R^2 value for each combination and keep track of one with the highest R^2 value.

Hint: Given three variables *A*, *B*, *C*, we only need to check *(A, B)*, *(A, C)*, and *(B, C)*. Take into account that a model with the pair *(A, B)* of variables is the same as the model with *(B, A)*.

To do this task, you will implement this function in `regression.py`:

```
def compute_best_pair(dataset):
    '''
    Find the bivariate model with the best R2 value

    Inputs:
        dataset: (DataSet object) a dataset

    Returns:
        A Model object for the best bivariate model
    '''
```

Unlike the functions in Task 1, you can expect to write more code in this function. In particular, you should not include the code to loop over all possible bivariate models within your `Model` class; it should instead be in this function.

You can test this function by running:

```
$ py.test -vk task2
```

You should also look at the output produced by `output.py`. How does the bivariate model compare to the single-variable models in Task 1? Which pair of variables perform best together? Does this result make sense given the results from Task 1? (You do not need to submit your answers to these questions; they're just food for thought!)

Task 3: Building models of arbitrary complexity

How do we efficiently determine how many and which variables will generate the best model? It turns out that this question is unsolved and is of interest to both computer scientists and statisticians. To find the best model of three variables we could repeat the process in Task 2 with three nested for-loops instead of two. For *K* variables we would have to use *K* nested for-loops. Nesting for-loops quickly becomes computationally expensive and infeasible for even the most powerful computers. Keep in mind that models in genetics research easily reach into the thousands of variables. How then do we find optimal models of several variables?

We'll approach this problem by using heuristics, which will give us an approximate (as opposed to an exact) result. We are going to split this problem into two tasks. In Task 3, you will determine the best *K* variables to use (for each possible value of *K*). In Task 4, you will determine the best value for *K*.

How do we determine which *K* variables will yield the best model for a fixed *K*? As noted, the naive approach, test all possible combinations as in Task 2, is intractable for large *K*. An alternative simpler approach is to choose the *K* variables with the *K* highest R^2 values in the table from Task 1. This approach does not work well if your predictors are correlated. As you've already seen from the first two tasks, neither of the top two individual predictors for crime (`GARBAGE` and `STREET_LIGHTS`) are included in the best bivariate model (`CRIME_TOTALS ~ -36.151629 + 3.300180 * POT_HOLES + 7.129337 * ABANDONED_BUILDINGS`).

Instead, you'll implement a heuristic known as Backward elimination, which is an example of a greedy algorithm. Backward elimination starts with a set that contains all potential predictor variables and then repeatedly eliminates variables until the set contains *K* variables. At each step, the algorithm identifies the

variable in the model that, when removed, yields the model with the best R^2 value. That variable is eliminated from the of predictor variables.

To do this task, you will implement this function in `regression.py`:

```
def backward_elimination(dataset):
    '''
    Given a dataset with P predictor variables, uses backward elimination to
    select models for every value of K between 1 and P.

    Inputs:
        dataset: (DataSet object) a dataset

    Returns:
        A list (of length P) of Model objects. The first element is the
        model where K=1, the second element is the model where K=2, and so on.
    '''
```

You can test this function by running this:

```
$ py.test -vk task3
```

A word of caution:

In this task, you have multiple models, each with their own list of predictor variables. Recall that lists are stored by reference, and so if you are not careful, it is easy to inadvertently modify a list that you did not mean to modify.

Furthermore, well-written solutions avoid the `remove` method of lists.

Task 4: Selecting the best K

In Task 3, you computed a list of the best K-variable models for $1 \leq K \leq P$ using backward elimination. How do we choose among these models? The calculation of R^2 does not take into account the number of variables in a model and so, while it is a valid way to choose among different models, each with K variables, it is not a good way to choose among different values for K. Adjusted R^2, which does take the number of predictor variables in the models into account, is a reasonable alternative.

Essentially, adjusted R^2 replaces the biased variances used in R^2 with their unbiased counterparts.) Here's a formula for it that highlights the adjustment, rather than the change in the definition of variance:

$$R^2_{\text{adj}} = R^2 - (1 - R^2) \cdot \frac{K}{N - K - 1}$$

where N is the sample size (*i.e.*, the number of rows in the training data) and K is the number of predictor variables in the model. So, we need to find the model with the best *adjusted* R^2 value.

To do this task, you will implement this function in `regression.py`:

```
def choose_best_model(dataset):
    '''
    Given a dataset, choose the best model produced
    by backwards elimination (i.e., the model with the highest
    adjusted R2)

    Inputs:
        dataset: (DataSet object) a dataset

    Returns:
        A Model object
    '''
```

Note that this function should not repeat the backwards elimination algorithm; you should simply call the function from Task 3.

You can test this function by running this:

```
$ py.test -vk task4
```

Task 5: Training vs. test data

Please read this section very carefully

Until now, you have evaluated a model using the data that was used to train it. The resulting model may be quite good for that particular dataset, but it may not be particularly good at predicting novel data. This is the problem that we have been referring to as *over-fitting*.

Up to this point, we have only computed R^2 *values for the training data* that was used to construct the model. That is, after training the model using the training data, we used that same data to compute an R^2 value. It is also valid to compute an R^2 value for a model when applied to other data (in our case, the data we set aside as testing data). When you do this, you still train the model (that is, you compute $\boldsymbol{\beta}$) using the *training data*, but then evaluate an R^2 value using the *testing data*.

> ## Warning:
>
> When training a model, you should only ever use the training data, even if you will later evaluate the model using other data. You will receive **zero credit** for this task if you train a new model using the testing data.

To do this task, you will implement the following function in `regression.py`. We will apply this function to evaluate the model identified in Task 4.

```
def validate_model(dataset, model):
    '''
    Given a dataset and a model trained on the training data,
    compute the R2 of applying that model to the testing data.

    Inputs:
        dataset: (DataSet object) a dataset
        model: (Model object) A model that must have been trained
           on the dataset's training data.

    Returns:
        (float) An R2 value
    '''
```

If your `Model` class is well designed, this function should not require more than one line (or, at most, a few lines) to implement. If that is not the case, please come to office hours or ask on Piazza so we can give you some quick feedback on your design.

You can test this function by running this:

```
$ py.test -vk task5
```

Grading

Programming assignments will be graded according to a general rubric. Specifically, we will assign points for completeness, correctness, design, and style. (For more details on the categories, see our PA Rubric page.)