FYS-STK4155 Project 1

Bendik Steinsvåg Dalen & Gabriel Sigurd Cabrera

September 25, 2019

Abstract

Introduction

Data

Method

Generalization of Multidimensional Polynomials

If we wish to construct a p-dimensional polynomial of degree d, we need to know what terms need to be included to give us a completely generalized polynomial. For a 1-D polynomial of second degree, we would have three terms:

$$f(x) = \beta_1 + \beta_2 x + \beta_3 x^2$$

Where β is a vector containing each coefficient. For a 2-D polynomial of second degree, we would have 6 terms:

$$f(x, y) = \beta_1 + \beta_2 x + \beta_3 y + \beta_4 xy + \beta_5 x^2 + \beta_6 y^2$$

And for a 3-D polynomial of second degree, we would have 10 terms:

$$f(x, y, z) = \beta_1 + \beta_2 x + \beta_3 y + \beta_4 z + \beta_5 xy + \beta_6 xz + \beta_7 yz + \beta_8 x^2 + \beta_9 y^2 + \beta_{10} z^2$$

There are many possible combinations of p and d, and the number of terms blows up significantly as these values increase. We can, however, create a general expression [1] for any p and d using summation notation:

$$f(\mathbf{x}) = \sum_{\sum_{j=1}^{d} i_j \leq p} \left(\beta_{i_1, i_2, \cdots, i_d} \prod_{k=1}^{d} x_k^{i_k} \right) \tag{1}$$

Alternatively, a simple `python` script can be used to find all the terms' exponents by calculating all permutations of the natural numbers from zero to d in sets of length p, then removing all results whose sum is greater than d.

```
    def get_exponents(p,d):
        powers = np.arange(0, d+1, 1)
        powers = np.repeat(powers, p)
        exponents = list(permutations(powers, p))
        exponents = np.unique(exponents, axis = 0)

        if p != 1:
            expo_sum = np.sum(exponents, axis = 1)
            valid_idx = np.where(np.less_equal(expo_sum, d))[0]
            exponents = np.array(exponents, dtype = np.int64)
            exponents = exponents[valid_idx]
        else:
            exponents = np.array(exponents, dtype = np.int64)

        return len(exponents)
```

Ordinary Least-Squares (OLS) Regression

We are given a $p+1$-dimensional dataset[1] consisting of N datapoints per feature such that:

$$\mathbf{X} = \begin{bmatrix} X_{1,1} & X_{1,2} & \cdots & X_{1,p} \\ X_{2,1} & X_{2,2} & \cdots & X_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ X_{N,1} & X_{N,2} & \cdots & X_{N,p} \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2}$$

Where the $N \times p$ matrix $\mathbf{X}$ contains the dataset's *input data*, and the N-vector $\mathbf{y}$ contains its *output data*, such that each row in $\mathbf{X}$ corresponds to a single output in $\mathbf{y}$.

Now, we wish to find a p-dimensional polynomial of degree d which most closely matches our dataset. We will need a design matrix $\mathbf{A}$; this will require using the knowledge presented in (1), since a design matrix should contain each polynomial term as an individual column.

Next, we will be using the method of *least squares* [2], whereby we attempt to minimize the *residual sum of squares*:

$$RSS(\beta) = \sum_{i=1}^{N} (y_i - \mathbf{A}_i^{\mathrm{T}} \beta)^2$$

Where $\mathbf{A}_i$ represents the i^{th} row in $\mathbf{A}$, and *beta* is the set of coefficents in the aforementioned polynomial; in matrix form, this can be written more concisely:

$$RSS(\beta) = (\mathbf{y} - \mathbf{A}\beta)^{\mathrm{T}} (\mathbf{y} - \mathbf{A}\beta)$$

To minimize the RSS, we can differentiate it with respect to β and set the right-hand side equal to zero – this allows us to solve for β, which will give us the coefficients to the polynomial that best matches our dataset:

$$\mathbf{A}^{\mathrm{T}}(\mathbf{y} - \mathbf{A}\beta) = 0 \iff \mathbf{A}^{\mathrm{T}} y = \mathbf{A}^{\mathrm{T}} \mathbf{A} \beta$$

Solving for β then gives us our desired result:

$$\beta = (\mathbf{A}^{\mathrm{T}} \mathbf{A})^{-1} \mathbf{A}^{\mathrm{T}} \mathbf{y} \tag{3}$$

[1] Meaning a set of p input features and 1 output.

Using the set of coefficients given by (3), we can then match the dataset from (2) as effectively as possible.

Ridge Regression

The solution for β given in (3) can be used without issue in many cases, but if the matrix $\mathbf{A}$ is singular[2], we run into an issue – namely, we cannot take the inverse of a singular matrix! As a result, we must look to more robust methods; one such method is called *ridge regression*.

The process of obtaining our vector of coefficients β via ridge regression is functionally very similar to that of OLS. The main difference is that we include an extra term in the residual sum of squares:

$$RSS(\beta) = \sum_{i=1}^{N}(y_i - \mathbf{A}_i^{\mathrm{T}}\beta)^2 + \lambda \sum_{i=1}^{N} \beta^2$$

In matrix form, this can be rewritten:

$$RSS(\beta) = (\mathbf{y} - \mathbf{A}\beta)^{\mathrm{T}}(\mathbf{y} - \mathbf{A}\beta) + \lambda\beta^{\mathrm{T}}\beta$$

Where λ, known as the *hyperparameter*, is a scalar value. Performing the same process as in the previous subsection, we are left with a solution similar to that in (3):

$$\beta = (\mathbf{A}^{\mathrm{T}}\mathbf{A} + \lambda\mathbf{I})^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{y} \tag{4}$$

In cases where $\mathbf{A}$ is singular, it is therefore possible to make very few changes to the OLS algorithm and still get a good result, one must simply optimize the hyperparameter and find a λ that minimizes the *mean squared error* (yet to be introduced) of our polynomial approximation.

LASSO Regression

The *least absolute shrinkage and selection operator*, commonly abbreviated as *LASSO*, is a method that implements the **L1** norm in place of the **L2** (or Euclidian) norm used in ridge regression. The residual sum of squares is therefore given by:

$$RSS(\beta) = \sum_{i=1}^{N}(y_i - \mathbf{A}_i^{\mathrm{T}}\beta)^2 + \lambda \sum_{i=1}^{N} |\beta| \tag{5}$$

Unfortunately, differentiating the above with respect to β will not work as intended, since we cannot take the matrix-form derivative of $\lambda \sum_{i=1}^{N} |\beta|$. As a result, we must use an iterative *gradient descent* method to minimize the right-hand side of (5).

[2]Meaning that $\det(\mathbf{A}) = 0$

Algorithm 1 The LASSO algorithm, over the course of 500 iterations.

$z = \sum_i A_i^2$
$i = 0$
while $i \leq 500$ **do**
    $i = i + 1$
    $j = 0$
    while j ¡ p **do**
        $\hat{y} = \sum_{k \neq j} \beta A_{*,k}$
        $\rho = \sum_k A_{*,k}(\mathbf{y} - \hat{\mathbf{y}})$
        if $\rho < -\lambda/2$ **then**
            $\beta_j = (\rho + \lambda/2)/z_j$
        else if $\rho > \lambda/2$ **then**
            $\beta_j = (\rho - \lambda/2)/z_j$
        else
            $\beta_j = 0$
        end if
    end while
end while

Mean Squared Error

To get a measure of success with respect to the implemented method and parameters, we can calculate the mean difference in the squares of each measured output y_i and their respective predicted outputs $\hat{y}_i$:

$$MSE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{6}$$

The lower the MSE, the closer the polynomial approximation is to the original dataset. If it is too low, however, we run the risk of overfitting our dataset, which is not desireable either – fortunately, this not an issue within the scope of this report.

$\mathbf{R^2}$ Score

Another measure of success is the *coefficient of determination*, colloquially known as the R^2 score, is given by the following expression:

$$R^2 = 1 - \frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{N} (y_i - \bar{y}_i)^2} \tag{7}$$

The closer R^2 is to one, the closer the polynomial approximation is to the input/output dataset, although a perfect score can once again arise due to overfitting just as in the case of the MSE.

Results

Discussion

Appendix

References

[1] M. (https://math.stackexchange.com/users/58320/macavity), "What is the general form of a polynomial of degree n and with m variables?." Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/2482654 (version: 2017-10-21).

[2] T. Hastie, R. Tibshirani, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction.* Springer, 2013.