Final Year Project Report

Full Unit - Final Report

Regression Algorithms for Learning

Nikita Bogachev

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science (Artificial Intelligence)

Supervisor: Zhiyuan Luo

Department of Computer Science
Royal Holloway University of London
April 08, 2021

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 9694 of the main body (12865 total)

Student Name: Nikita Bogachev

Date of Submission: 08.04.2021

Signature:

Table of contents

Chapter 1: Introduction

1.1 Aims and goals of the project

The main aim of the project is to implement a popular machine learning technique called ridge regression, apply it to real-world data, analyse its performance, and compare it against other data analysis methods. Additionally, it is required to use object-oriented design for the final deliverables, use a real-world dataset and have a graphical user interface.

I specifically chose to pursue a degree in computer science with AI, as software engineering allows me to embrace my creative side as well as my interest in mathematics. Simply put, working on something and seeing it come to life and work as intended is truly inspiring. I had this feeling with my music and prose, and I still do, as I have with programming. Furthermore, as AI and Machine Learning is on the rise, having a specialisation in this area pointed me towards picking an appropriate project to have a more diverse understanding of what are the algorithms so broadly used in practice. I believe that implementing them myself rather than using the built-in scikit-learn functions in «real life» procedures will allow me to understand the background theory significantly better, even when I do use scikit-learn methods.

Seeing as I had previous experience with data warehouse project in banking, being able to not only extract and manipulate data with SQL and visualise it for presentations to the board but also to be able to predict some outcome from such data is what will make me stand out to potential recruiters in the future.

Additionally, I have decided to implement other regression algorithms such as KNN Regression, Least Squares Regression and Lasso Regression, inspired by the material studied in CS3920 Machine Learning, as I consider it a highly rich source of content for the project.

1.2 Abstract

Machine learning is a relatively new topic in computer science, and most lucrative as well. The question whether computers will be able to learn has been a pressing one since the first programming languages, as we were already able to tell computers to do as we ask, but the aim was to go further: write programs that let *them* decide the outcome of the request without us humans necessarily knowing it ourselves, as our minds could only analyse that much data efficiently and under short time. As this has became more popular, more and more research has been going into this new field of computer science, with Professors Vladimir Vovk and Vladimir Vapnik in the lead with conformal prediction and support-vector machine methods, both used in application to the collective term of regression algorithms.

With the ever increasing progress in machine learning, regression algorithms (or regression analysis) now play an important role in it. They allow us to learn dependencies between multidimensional attributes and continuous outcomes. The idea behind regression is estimating the relationships between two types of variables — a dependant (a.k.a the outcome) variable and a set of independent variables called «predictors». The most common form of this concept that most would be familiar with is from high school mathematics. It is called linear regression, and the aim is to find a line that fits the scattered data as closely as possible, for example the least squares method that we have been applying in the CS2900 Multidimensional Data Processing course using Python and a module called NumPy, which are both commonly used for large data manipulation.

As such, the aforementioned «predictors» indicate that the goal of regression algorithms is prediction output based on a supplied data set, which ties into the concept of machine learning. This approach have various uses, and is often used to predict things like stock value changes, as the algorithm is fed past stock data to analyse and learn upon and then applied onto real-time data to predict and learn on the go, adapting the possible outcome of various changes in the

stock market. To use regression algorithms it is important to carefully justify why existing relationships have predictive power for a new context or why a relationship between two variables has a causal interpretation.

There are two main types of learning — supervised and unsupervised.
Supervised learning *— building a model on the training data and then being able to make accurate predictions on new, unseen data that has the same characteristics as the training set that we used* [1].
Unsupervised learning is *concerned with analysing data without labels* [2].

In this project I am using supervised learning approach as data sets used have clear labels y. The supervised learning method can be applied further to classification and regression methods of learning algorithms. Both of these methods deal with label prediction, however in slightly different ways: if the number of labels is *finite*, the problem is known as **classification** and the labels are sometimes referred to as classes, while in the case when the number of potential labels is *infinite* the problem is called **regression**. In this project I will be looking at the main regression algorithms commonly used in machine learning.

Chapter 2:
Background theory and proof-of-concept

First of all, I want to take a step back and ask - what is machine learning? As defined by Tom Mitchell in his treatise on the subject,

«A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E» [3].

This is a quite complex way of presenting the answer, so yet another question - what does this actually mean? Rephrasing the above, we can measure the performance of a machine learning algorithm, which is how well a task is executed, based on some input experience. However, this is subjective, as the concept of «well» can relate to output accuracy, time taken, or even both, but usually it is related to a measure of accuracy for the algorithm. Regression algorithms in question use this definition of accuracy and this measure of performance is called R^2, which I will explain later in more detail.

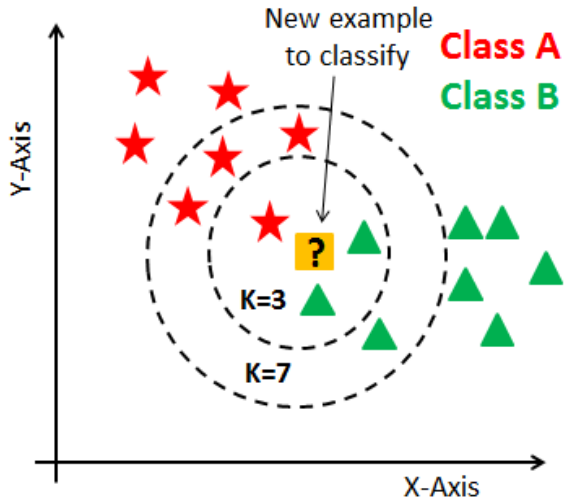
So, how can a computer gain experience? When designing anything related to artificial intelligence, be it classification and/or regression algorithms for machine learning, or intelligent multi-agent systems, there is some sort of notion of knowledge. In the case of learning algorithms, we usually have a training set of X and y - the attributes and the labels - which we can use to train the algorithm. In programming, this action is called a `fit()` function. This function trains the algorithm on existing data in a unique way to each given algorithm and supplies the computer with the knowledge about relationships between X and y attributes. The algorithms also have a prediction function that runs the main formula, which varies between algorithms, on a given test set of X values. All of this will be explored in more detail in the following chapters.

For this project I will be exploring 4 popular regression algorithms: K Nearest Neighbours (KNN) Regression and Linear Regression, the latter of which consists of three types — Least Squares, Ridge and Lasso, what do they have in common and what are their differences. While I have built each algorithm myself, they all follow the structure of the scikit-learn versions of the algorithms studied in CS3920, meaning they have three main functions: `fit()`, `predict()` and `score()`. For the purpose of this project I am focusing on analysis of the algorithms, so for all the 4 datasets I am using all the data is known. Due to this, the data in the project all share a similar basic principle:

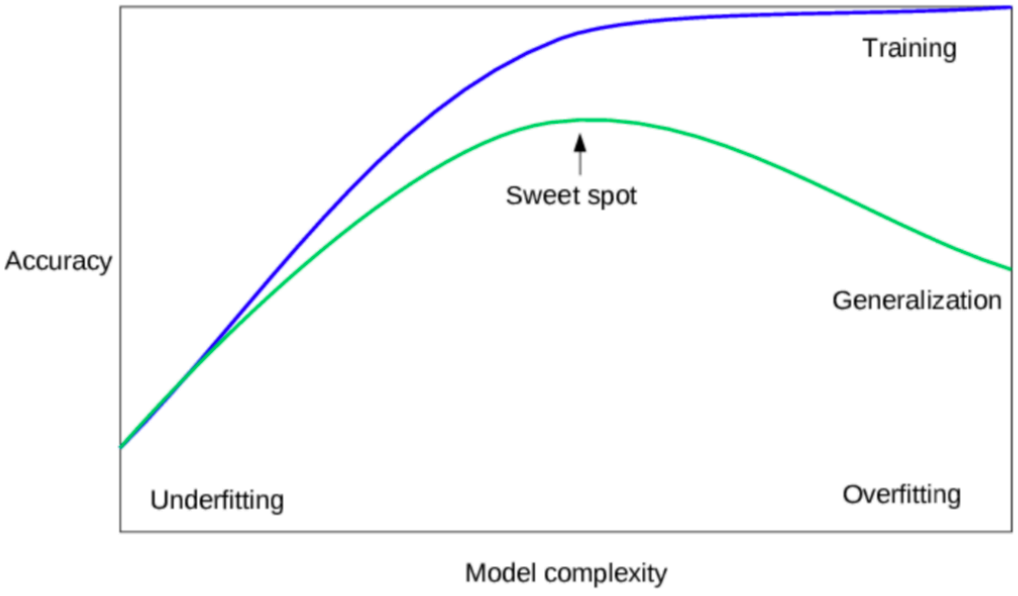
- The data is presented as a matrix where each row has some number of attributes of x ($x_i \ldots x_n$) like age, sex, and height, with the last element being some label y.
- The data is randomly shuffled and is split into training sets and test set. Usually, the training set is 70% of the original matrix with the remaining 30% being the test data.
- The goal of the algorithm is to determine some dependency between attributes and labels of the X_train and y_train and to predict the y labels for all of X_test using the values of the row's attributes, and then compare them against the real y_test labels to find the accuracy of the algorithm.

2.1 K Nearest Neighbours Regression

The first algorithm, and likely the most straightforward one, is the K Nearest Neighbours algorithm, developed by Evelyn Fix and Joseph Hodges in 1951, and later expanded by Thomas Cover.
It is a nonparametric algorithm, meaning it does not take a closed form that can be expressed as a function $f(x)$. The idea behind this algorithm is, as the name implies, to find some number K of nearest neighbours to our currently selected data entry. For example, when used on the Iris dataset, one row of X_train and, respectively, X_test contains four attributes - sepal length, sepal width, petal length, petal width and species - and y_train and y_test contain labels of 0, 1 or 2 representing species: setosa, versicolor, or virginica. Graphically, the algorithm does the following [4]:



The ? marker is our current tuple of X_test and we want to find its nearest neighbours in X_train. In this graph, there are two possible labels - A or B. If we set k=3, the nearest elements are 2 items of class B and 1 item of class A - so since regression uses the mean to find the predicted label, it will set class B as our label for ?. But, if we use k=7, we will have 4A and 3B and the mean label for ? will be A. So, for different k we can get different predictions! This indicates the importance of choosing the correct value of k for most accurate prediction possible, as the model can easily underfit or overfit depending on the value of k [5].



So how to implement this algorithm? The KNN object takes in 3 arguments: both train sets of X values and y labels, and a value of k that determines the number of neighbours we will consider for the algorithm. Based on this value of k, the `fit()` function of the algorithm uses the training data and a single test tuple from the test set of X values to create a sorted array of distances of length k. It does so using the Euclidean distance formula:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \ldots + (p_n - q_n)^2}$$

Where p is some point of the form *(p_1, p_2 … p_n)* and q is another point of a similar form *(q_1, q_2 … q_n)*. The goal is to find the difference between each value of the point, e.g. a point in 3D space has attributes x, y, and z, so we find p_x - q_x, p_y-q_y and p_z-q_z.
And in code form the following operation happens:

```
distance = []
for data in range(len(self.X_train)):
    E_distance = np.linalg.norm(self.X_train[data] - testTuple)
    self.distance.append([self.X_train[data], E_distance, self.y_train[data]])
```

Where `data` is one row in train set X and `testTuple` is one row of X_test, both of which follows the form mentioned above.
This loop is the first half of the `fit()` function for KNN. It takes one data entry (one row) of test set X and runs it against every entry/row of the training set X to find the Euclidean distance to each point.

While the `norm()` function usually determines the «size» of a single matrix, here we are obtaining the mathematical distance between two data points, so their difference normalised is the «size» of the distance. The function then fills up an array of distances of the form [X_train[data], E_distance, y_train[data]], where the data is used as the index of the current row (with attributes of train set X and train set label y) and E_distance is the calculated Euclidean distance from test tuple to the train tuple.

Distance measures may vary depending on the type and size of data. There are two other possible formulas for distance when we are dealing with continuous variables [6]:

Manhattan: $d(p,q) = |p_1 - q_1| + |p_2 - q_2| + \ldots + |p_n - q_n|$
Minkowski: $d(p,q) = ((p_1 - q_1)^a + (p_2 - q_2)^a + \ldots + (p_n - q_n)^a)^{1/a}$

Realistically, any of these three can be applied. I picked Euclidean as the most familiar approach. An exception would happen only in the case of categorical variables like Strings, where we need to use Hamming distance and run both strings against each other to see if their characters match [7]:

$$D_H(p,q) = |p_1 - q_1| + |p_2 - q_2| + \ldots + |p_n - q_n|$$
$p = q \Rightarrow D = 0$
$p \neq q \Rightarrow D = 1$
(where p_n is a character of string p at index n, same for q respectively).

After the distance array is filled with data, it must be sorted using a sorting algorithm or a `sort()` function. The line below sorts the distances using the values of the second element in the distance array (therefore of index 1 which we supply to the `itemgetter` object), which is distance and the only element which we can base the sorting on.

```
self.distance.sort(key=operator.itemgetter(1))
```

The sorted distances array can now be used for predictions. For this purpose, a new empty array is made for the labels list. As the loop executes, this array is filled with the said labels and the mean label is found — this is the regression approach.

```
labels_list = []
for x in range(self.k):
    labels_list.append(self.distance[x][2])
return round(np.mean(labels_list))
```

As can be seen above, this loop only runs k times. Since the algorithm have sorted the distances array for this current tuple by distance, it is able to get just the top K nearest neighbours for our current tuple - but it only needs the labels, so it retrieves them from the distance array by using index [x] to specify the row (which equals to the current k, so if we have k=3, the indexes are 0, 1 and 2 — meaning 1st neighbour, 2nd neighbour, etc) and then index [2] to specify the third column, which was set for labels earlier.

In summary, the actual output of the `fit()` method is a single predicted label for the current tuple. Then, the `predict()` method does the `fit()` method in a loop for each entry/row of X_test, returning a single label and filling up the y_pred array of predicted labels.

In a real situation, we would not know the real labels of the set we build our predictions upon. But if we do, we can use these labels to find the accuracy score. For this, we use the notion of R^2 [8]:

$$R^2 = \frac{TSS - RSS}{TSS}, where$$

$TSS = \sum_{i=1}^{n} (y_{pred_i} - y_{test_i})^2$, also called Total Sum of Squares.

$RSS = \sum_{i=1}^{n} (y_{test_i} - \overline{y}_{test_i})^2$, also called Residual Sum of Squares.

In other words:
- TSS is the sum of squared difference between predicted and actual y labels
- RSS is the sum of squared difference between actual y labels and the mean value of actual y labels.
- R^2 is the difference between TSS and RSS divided by the value of TSS. This computes an accuracy score, which must be between -∞ and 1.

In code format it will look very similar:

```
TSS = np.sum((y_test - np.mean(y_test)) ** 2)
RSS = np.sum((self.y_pred - y_test) ** 2)
R2 = (TSS - RSS) / TSS
```

An example output when we use Least Squares on the Boston dataset:

```
TSS = 10541.366141732284
RSS = 3057.5770600069536
TSS - RSS = 7483.78908172533
R2 = 0.7099448952918643
```

This seems like a bunch of large numbers, but there is a reason to why they are used. Samrat Kar has explained this quite well:

«TSS works as a cost function for a model which does not have an independent variable, but only y intercept (mean ȳ). This gives how good is the model without any independent variable. When independent variable is added the model performance is given by RSS. The ration of RSS/TSS gives how good is the model as compared to the mean value without variance. Lesser is this ratio lesser is the residual error with actual values, and greater is the residual error with the mean. This implies that the model is more robust. So, 1-RSS/TSS is considered as the measure of robustness of the model and is known as R^2» [9]

Adding to what is said above, we can evidently use RSS/TSS to compute the error rate. This also gives us an alternative formula for R^2:

$$R^2 = 1 - \frac{RSS}{TSS}$$

We can see how R^2 changes with k. Let us test the algorithm on the Iris dataset, with k=1 first.

```
y_pred = [0 1 0 2 1 0 1 0 2 1 1 1 0 2 1 2 1 2 0 0 1 2 0 2 1 2 2 0 1 2 1 2 0 1 1 2 1 2]
y_test = [0 1 0 2 1 0 1 0 2 1 1 1 0 2 2 2 1 2 0 0 1 2 0 2 2 2 2 0 1 2 1 2 0 1 1 2 1 2]
0.9178378378378378
```

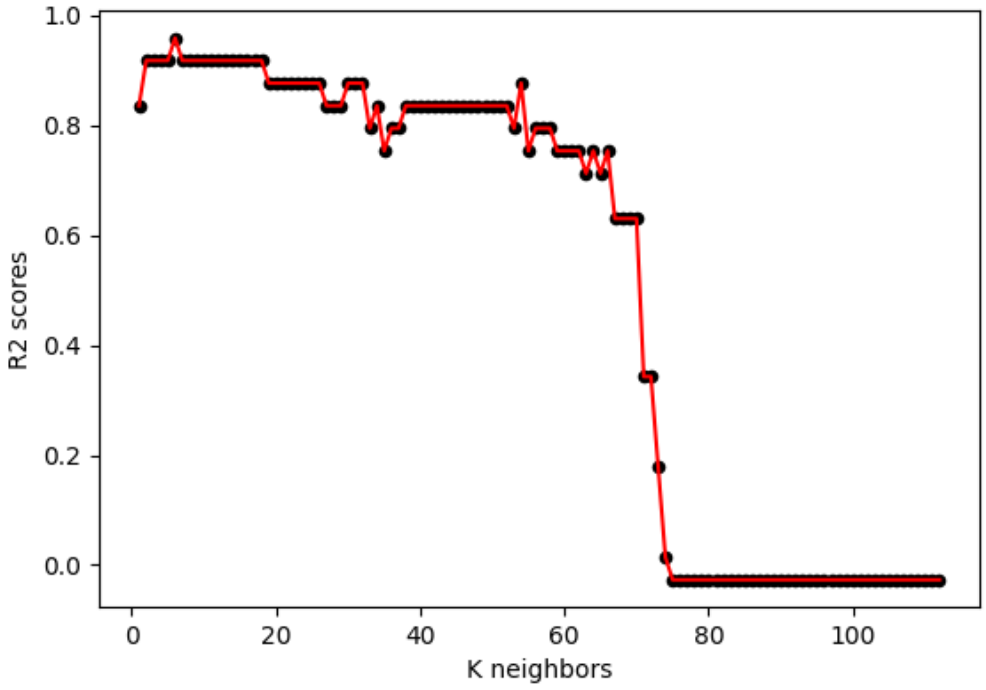
And now if we increase k to 3…

```
y_pred = [0 1 0 2 1 0 1 0 2 1 1 1 0 2 2 2 1 2 0 0 1 2 0 2 1 2 2 0 1 2 1 2 0 1 1 2 1 2]
y_test = [0 1 0 2 1 0 1 0 2 1 1 1 0 2 2 2 1 2 0 0 1 2 0 2 2 2 2 0 1 2 1 2 0 1 1 2 1 2]
0.9589189189189189
```

We can see a slight increase in score results. Iris dataset is quite large and we can use the entirety of the X_train to widen the range of neighbours. Let us see the extreme results if we take k = 75:

```
y_pred = [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1]
y_test = [0 1 0 2 1 0 1 0 2 1 1 1 0 2 2 2 1 2 0 0 1 2 0 2 2 2 2 0 1 2 1 2 0 1 1 2 1 2]
0.05513513513513496
```

Choosing the right k value is very important, as accuracy jumps up and down depending on the number of neighbours, eventually decreasing (forming a sort of an inverse U shape). This graph below illustrates this on the Iris dataset.



2.2 Linear Regression

Moving further to linear regression algorithms. The general idea behind them (i.e. building a line of best fit) may be familiar to most, but in reality these are a bit more complex. This subchapter will be dedicated to three main algorithms: Least Squares, Ridge and Lasso.
The following algorithms now take a parametric approach. Parametric algorithms assume a function form of $y = f(X)$ to have a formulaic way (or closed form) of finding the label. Such algorithms are easier to fit and the coefficients, w, have simple interpretations and can be easily estimated [10].

Linear regression can go two ways - simple linear and multiple linear, with the latter dealing with multidimensional data.

For univariate linear regression, prediction of y is quite simple and takes the form of

$$y = w * x + b$$

This means that to predict a given label y, the algorithm takes the current value from the X matrix, multiplies it by some value *w* and elevates or brings it down by a value of *b*. For this reason, *b* is also called the intercept. In some notations, *b* can also be represented as b_0 or w_0 and *w* can be represented as b_1 or w_1. In my notation and this report, I will use b_0 as my notation for intercept, *w* as an overall array of coefficients, and $b_1 \ldots b_n$ as the actual coefficients used on the X values.

Lets take a completely random example with some numbers as our data:
X = [1, 2, 5, 3, 7, 9]
y = [0, 0, 1, 2, 2, 1]

To be able to use it on the algorithm, we need to reshape X into a single column as the algorithm analyses X data row by row. So we turn each element into its own row. However, reshaping y is not necessary as it analyses the list element by element in any case.

```
X.reshape(-1, 1)
```

So, we only have one attribute of X for each label of y. For simplicity, it is better to represent it as a matrix, which will be more akin to a table or a dataset, and will would look like this:

X	y
1	0
2	0
5	1
3	2
7	2
9	1

This representation explains why we would only have a single w value for our prediction - we only have one feature of X, and the coefficient formulae we will look at below only builds one w for each feature. However, we will always have only one value for b which will be also explained below.

It gets more interesting when we would have 2 or more features. Lets add another feature (a column with random values) to the table:

```
X = np.array([1, 3],
             [2, 8],
             [5, 4],
             [3, 5],
             [7, 7],
             [9, 6])
```

X1	X2	y
1	3	0
2	8	0
5	4	1
3	5	2
7	7	2
9	6	1

Now our formula to find y_pred will also change. We now would have two values of *w*, b_1 and b_2.

$$y = b_0 + x_1 b_1 + x_2 b_2$$

We now have two features to consider and two coefficients to apply to them, with the slope b_0 being added separately. There is a small trick for doing it - we add a column of ones to our data.

X0	X1	X2	y
1	1	3	0
1	2	8	0
1	5	4	1
1	3	5	2
1	7	7	2
1	9	6	1

```
ones = np.ones(shape=self.X_train.shape[0]).reshape(-1, 1)
self.X_train = np.concatenate((ones, self.X_train), 1)
```

Taking the number of rows (X_train.shape[0]), we create an array of ones and, just like we did above, reshape it into a column. And then we rewrite X_train as a new matrix where we add the old X_train next to that new column of ones. This will allow us to create the intercept in any formula as the formula now becomes:

$$y = b_0 * 1 + x_1 b_1 + x_2 b_2$$

But for simplicity, this 1 is usually not written.

So, we have our X_train and y_train and some matrix of X similar to X_train which we will use as our X_test. How do we find the array of *w* to build our array of predictions? I will look at three options in the next subchapters.

2.2.1 Least Squares

Least Squares in its most concise form was first published by Legendre in 1805 [11], and further improved by Carl Friedrich Gauss in 1809, who was claiming to be in possession of the method since 1795. This has led to some dispute with Legendre, but Gauss has succeeded in improving his method by connecting the original least squares with the principles of probability and to the normal distribution, and also demonstrated that arithmetic mean is indeed the best estimate of the location parameter. The Gaussian method is the one we use today, after in an early demonstration of its strength it was successfully used to predict the future location of Ceres, smallest known dwarf planet. [12].

This is the general formula for least squares coefficients [12], for one feature.

$$w = \frac{\sum_i x_i y_i}{\sum_i x_i^2}$$

The way it works is that the algorithm would go element wise through the column (as mentioned earlier, we create one coefficient w for each column, i.e. feature) and add the product of current X and y to the sum $\sum xy$ and the square value of current X to $\sum x^2$.

However, this only works on a single column. Doing this in a loop for all columns for a transposed X_train may work, but it is inefficient. In the case of multiple linear regression (where X set has more than one feature), we have a different formula for coefficients which can be expressed in a matrical closed form [13]:

$$w = (X'X)^{-1}X'Y$$

Breaking it all down:
- find the inverse of the dot product of X_train and transposed X_train
- find the dot product of transposed X_train and y_train
- find the dot product between the two obtained products.

We now have obtained an array of coefficients, where b (the intercept b_0) is the first element of `w[]`. The reason it works is since b0 is found for our column of ones, we can observe the following if applied to our one-column formula:

$$b_0 = \frac{\sum 1 y_i}{\sum 1^2} = \sum y_i$$

Since our value of x is 1 for all i, it is essentially a sum of all y-values. That is why the value of b_0 is usually significantly larger than the rest of `w[]`.

Going back to the matrical formula, it works in a somewhat similar manner to the simpler formula for one column. Transposing X_train allows us to find X'X, which is essentially the x^2 part, while X'Y is the xy part. The reason why we are transposing is the same reason why we have been reshaping rows into columns - now we transpose them instead as we need them to be rows to find the dot product, as y_train is a column vector.

In code, this can be expressed as a NumPy function called `solve()`:

```
self.coeffs = np.linalg.solve(np.transpose(self.X_train) @ self.X_train,
                              np.transpose(self.X_train) @ self.y_train)
```

In this function, the `@` symbol means dot product. This, however, solves a slightly different equation. We imagine the formula above as $X'Xw = X'Y$. We are looking for *w*, so we take the left side X'X and the right side X'Y as our arguments for `solve()`. This finds us our *w.* Therefore, all the `fit()` function does is it creates a column of ones, attaches that column in front of X_train and solves the equation above for an array of *w*.

Now that we have all the coefficients, we can find the predicted y labels using `predict()`.

```
intercept = self.coeffs[0]  # set first element as the intercept b0
b_vals = self.coeffs[1:]  # set the rest of coefficients starting from b1
pred = []  # array for predictions
for entry in X_test:  # pick  row in X_test
    y_current = intercept  # start as y = b0 + ...
    for xi, bi in zip(entry, b_vals):  # each value of X in a row has its own coefficient
                                       in b_vals
        y_current += bi * xi  # find the product and add to current y value
    pred.append(y_current)
self.y_pred = np.copy(pred)
return self.y_pred
```

This illustrates the formula $y = b_0 + x_1b_1 + x_2b_2 + \ldots + x_nb_n$ as we take a single row of X_test and build a y value for it using the formula above ($y = b_0$, $y \mathrel{+}= b_i * x_i$). This is done in a double loop, where the outer loop iterates through rows and inner loop iterates through elements of the row and coefficients.

Similarly to KNN, if we know our y_test we can find the accuracy score, once again using the notion of R^2 which is the right way to calculate accuracy of a regression algorithm. Since there is no parameter in Least Squares, whatever prediction it returns is what you get depending on the dataset supplied.

And now, moving onto more complex versions of linear regression…

2.2.3 Ridge Regression

The theory behind ridge regression was first introduced by Hoerl and Kennard in 1970 in their *Technometrics* papers "RIDGE regressions: biased estimation of nonorthogonal problems" and "RIDGE regressions: applications in nonorthagonal problems" [14].

Ridge regression builds up on the concept of least squares by introducing a new value of alpha, α. Alpha is a limiting factor that «shrinks» the coefficients — this is called regularisation or tuning. Regularisation prevents our model from overfitting (which is when our predicted labels are overfit and may get too high, resulting in a lower R^2 score). This is done due to the actual products of each $w * x$ operation being altered with alpha, meaning the final y label will also be altered. Depending on the chosen value of alpha, like with k for KNN, the final result may vary accordingly.

Like with least squares, we can use a matrical formula [15]:

$$w = (X'X + aI)^{-1}X'Y$$

where I is the identity matrix of $X'X$ and b (the intercept b_0) is the first element of w.

The question is, how does this regularisation happens? Now we add a matrix aI to $X'X$ which is the same dimensions as $X'X$ but is filled with zeroes and has a diagonal of the value of α going from top left to bottom right. For α = 0.8:

```
[[0.8 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.8 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.8 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.8 0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.8 0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.8 0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.8 0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.8 0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.8 0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.8 0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.8 0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.8 0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.8]]
```

The aim of ridge is to bring the coefficients closer to zero (but not equal to 0) with this new regularisation parameter. However, alpha must stay between 0 and 1 and can be up to any number of decimal places. Let us look at two extremes and first take the value of α to be 0 on Boston dataset, this would give us the Least Squares solution as it has no alpha:

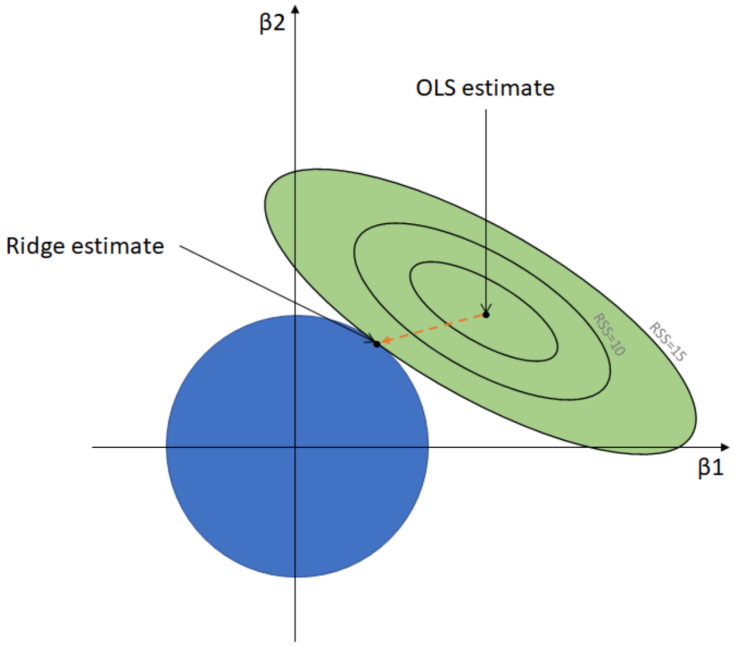
```
b0 = 34.466318552232565
b_vals = [-1.13953709e-01  3.93052116e-02  2.42446092e-03  3.19606223e+00
 -1.42238929e+01  3.87110173e+00 -6.10184926e-03 -1.40899633e+00
  3.20885675e-01 -1.21833888e-02 -9.53020968e-01  9.60776426e-03
 -5.32847468e-01]
```

But if we increase α to 1, we would see a decrease in the values.

```
b_0 = 12.530669021474328
b_vals = [-0.10859091  0.04134394 -0.01507946  3.30626432 -2.94181449
4.95977359
 -0.01256303 -1.07958435  0.242332   -0.01090823 -0.59861351  0.01318949
 -0.48906032]
```

As can be seen, a large value of α did not make any coefficient equal to exactly zero. In this way, Ridge creates a «soft limit» on coefficients.

We can present this regularisation graphically, with references to an article by Qshick [16] (see next page).



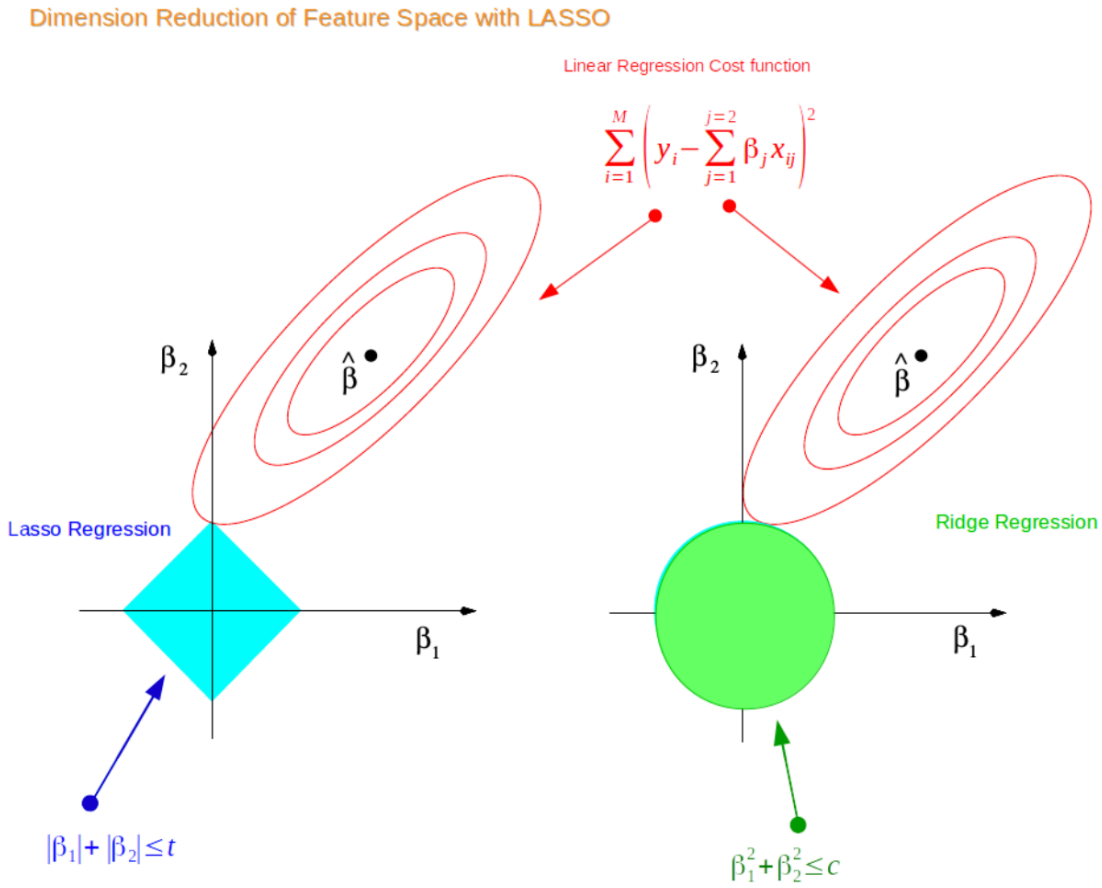
Each ellipse in the green area is a connection of spots where the RSS value is the same, centered with the Ordinary Least Squares (OLS) estimate where the RSS value is the lowest. Unlike the OLS estimate, the ridge estimate changes as the size of the blue circle changes. It is simply where the circle meets the most outer contour of OLS. How ridge regression works is how we tune the size of the circle, i.e this regularisation parameter α. For this reason, ridge β's can never be zero, but can get close to it [16].

In terms of code, the `predict()` and `score()` method remain the same with the accuracy score still being calculated using the R^2 notation, with only `fit()` changing to accommodate the new parameter (with changes highlighted in red).

```
I = np.identity(len(np.transpose(self.X_train).dot(self.X_train)))
self.coeffs = np.linalg.solve(np.transpose(self.X_train) @ self.X_train
                              + I.dot(alpha),
                              np.transpose(self.X_train) @ self.y_train)
```

2.2.4 Lasso Regression

Lasso regression can be described as a more strict version of Ridge. The method was designed independently in geophysics literature in 1986, and Statistician Robert Tibshirani independently rediscovered and popularized this method in 1996, based on Breiman's nonnegative garrote [17] [18].
LASSO is actually an abbreviation for ***least absolute shrinkage and selection*** operator, and coincidentally the abbreviation fits it perfectly. Lasso creates a much harder limit on the coefficients by bringing some coefficients down to 0 if their value is too small with the usage of α. This way, if the coefficient is small, Lasso deems the feature it relates to as «insignificant» and therefore having small effect on the final prediction of the y label, virtually «catching it with a lasso» and removing that feature from the equation by setting it to 0 (e.g. $y = b_0 + b_1x_1 + 0x_2 + b_3x_3 + 0x_4 = b_0 + b_1x_1 + b_3x_3$ and so on).



As can be seen on the graph above [19], Lasso uses a diamond-shape regularisation compared to the circle of Ridge. The outer ellipse of the OLS touches the top edge of the diamond, meaning that for certain values of alpha the diamond will be small enough to bring that top edge (and therefore the point of lasso estimate where the two figures meet) exactly to zero.

However, Lasso does not have a closed form expression that is as straightforward as Least Squares and Ridge do. There have been at least four different variations of the expression I have came across during my research, but I will be looking at only the last two of them, as they a) work and b) made me realise how the first two work.

The first one is described in an article by Wessel van Wieringen [20]:

$$X'Xw(a) = X'Y - \frac{a}{2}z,$$
where $z_j = sign[w(a)_j]$

Therefore, I believe the Lasso coefficient formula can be written as:

$$w(a) = (X'X)^{-1}(X'Y - \frac{a}{2}z)$$

When $w(a) < 0$, z = -1, making the formula to be $w = (X'X)^{-1}(X'Y + \frac{a}{2})$, bringing it closer to zero.
When $w(a) > 0$, z = 1, making the formula to be $w = (X'X)^{-1}(X'Y - \frac{a}{2})$, bringing it closer to zero.
When $w(a) = 0$, $z \in$ [-1,1].

The last line is where I got lost, as z can be any real number between -1 and 1. Additionally, it was unclear to me how can we find both w and z at the same time because one needs the other, as the formula clearly uses the same value for $w(a)$. But it was a good start and the first mention of a closed form formula I could somewhat understand after months of trying.

In a research posted on the online portal of the Russian High School of Economics [21] I have discovered that the w and z used to find the w_{lasso} are both found from Least Squares first, which I suspected as potential solution to van Wieringen's formula.
The presented formula looked like this:

When $w_{ls} > 0$, $w_{lasso} = w_{ls} - \frac{a}{2}$.

When $w_{ls} < 0$, $w_{lasso} = w_{ls} + \frac{a}{2}$.

This formula combined looked like this:

$$w_{lasso} = sign(w_{ls})(|w_{ls}| - \frac{a}{2})_+$$

The subscript x_+ means that the content of this bracket will be the result of $max(x,0)$. However, this formula only works when a matrix is orthogonal (meaning $X'X = I$). Therefore, to implement the `fit()` function for my Lasso I have used both methods to create the following solution:

$$w_{lasso} = sign(w_{ls})(|w_{ls}| - \frac{a}{2})_+ \quad \text{iff } X'X = I$$
else
$$w_{lasso} = (X'X)^{-1}(X'Y - \frac{a}{2}sign(w_{ls})) \text{ where } w_{lasso_i} = 0 \text{ if } |w_{lasso_i}| \leq \frac{a}{2}$$

The first one solves for w when the data is orthogonal, solving for Least Squares first and then for each w_{ls} taking its absolute value and subtracting half-alpha from it. If that value is bigger than zero, it is multiplied by the sign of w_{ls} (-1 if negative, 1 if positive or 0 if zero) and taken as a value for w_{lasso}.

The second formula works in all other cases, by solving for w_{lasso} using signs of w_{ls} once again and looks somewhat similar to Ridge, but without using $max(x,0)$. Instead, this calculation is done post-factum by taking the absolute value of obtained w_{lasso} and checking if it is less or equal to half-alpha and if that is True, setting the value to zero. Essentially, that is the way $max(x,0)$ works as the only way to obtain a negative value inside the brackets for $(|w_{ls}| - \frac{a}{2})$ is for $|w_{ls}|$ to be smaller than half-alpha, setting the value inside brackets to zero with the + subscript.

The code implementation does these three things in this manner:

1) Find w_{ls} as a separate array using the formula used in Least Squares

```
w_least_squares = np.linalg.solve(np.transpose(self.X_train) @ self.X_train,
                                  np.transpose(self.X_train) @ self.y_train)
```

2) If matrix is orthogonal, then *w(Lasso) = sign(w(LS)) (|w(LS)| - 0.5α)+*

```
if np.transpose(self.X_train).dot(self.X_train) == np.identity(len(self.X_train)):
    self.coeffs = list(map(lambda w: np.sign(w) * max(abs(w) - (alpha / 2), 0),
                           w_least_squares))
```

The `list(map())` here creates the list with a lambda function, which is Python's built-in functional programming trick, with the function, of course, being the implementation posted by HSE, where for each w of w_{ls} we find its corresponding w_{lasso}.

3) Otherwise use the formula from the article by van Wieringen. For this purpose, I decided it would be easier to have an array that is purely signs of every value in w_{ls}. Then solve the formula is a similar format to how Least Squares and Ridge were implemented with `np.linalg.solve()`.

```
else:
    # we make a list of sign values of LS coefficients
    sign_list = []
    for w in w_least_squares:
        i = np.sign(w)  # -1, 1 or 0 if w=0
        sign_list.append(int(i))
    signs = np.copy(sign_list)
    self.coeffs = np.linalg.solve(self.X_train.T @ self.X_train,
                                  self.X_train.T @ self.y_train - (alpha / 2) * signs)

    # Implement max(x, 0) the direct way
    for i, coef in enumerate(self.coeffs):
        if abs(coef) <= alpha / 2:
            self.coeffs[i] = 0
```

The `.T` here does the same as `np.transpose()`.

As I have shown before, Ridge only decreases all of the coefficients by «limiting» them with α, so the bigger the value of α, the smaller will be the coefficients, but never exactly zero. Lasso chooses which coefficients have the smallest value and sets them to zero with the functionality of max().

To demonstrate the differences, I will once again use the Boston dataset and $\alpha = 0$, once again getting the same result as for OLS and Ridge($\alpha = 0$):

```
b_0 = 34.466318552232565
b_vals = [-1.13953709e-01  3.93052116e-02  2.42446092e-03  3.19606223e+00
 -1.42238929e+01  3.87110173e+00 -6.10184926e-03 -1.40899633e+00
  3.20885675e-01 -1.21833888e-02 -9.53020968e-01  9.60776426e-03
 -5.32847468e-01]
```

However, setting α to 1 will make a much bigger difference than it did in Ridge:

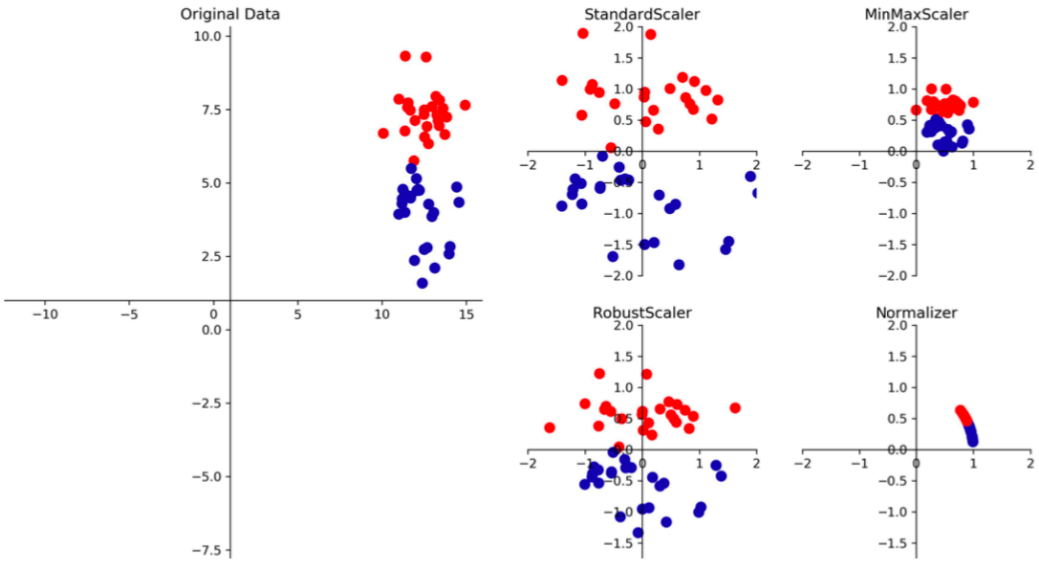
```
b_0 = 33.334136202482625
b_vals = [  0.           0.           0.           3.18320941 -13.41531083
   3.9160176    0.          -1.39014808   0.           0.
  -0.93443428   0.          -0.53156538]
```

As can be seen, this time some of the coefficients are set exactly to zero and the formula for y changes to use only some of the features as the others are discarded:
$y = b_0 + b_4x_4 + b_5x_5 + b_6x_6 + b_8x_8 + b_{11}x_{11} + b_{13}x_{13}$

However, we are using the raw original data of Boston, which may change our results. Usually, for data where features differ from each other a lot, for example some medical dataset for patients where one feature is age and the other is sex (which are not even of the same format), or if the data is not evenly spread out, it is important to use data normalisation. To do so, we are using Scalers.

2.3 Data normalisation and Scalers

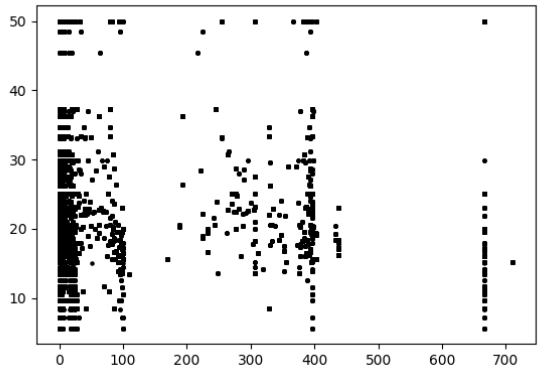
The Scalers are objects of the scikit-learn library that, as the name says, scale input data to bring it to a more universal format or «normalise it».



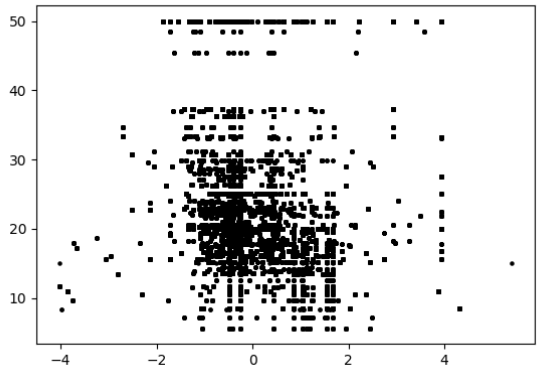
[22]

The graph shows a plot of some original data set with only two features, A and B, and on the right hand side 4 scalers that exist in scikit-learn, each modifying data in its own way. For example, The `StandardScaler` ensures that for each feature, the mean is zero and the variance is one, thus bringing all features to the same scale, while `MinMaxScaler` shifts all features to be positive values between 0 and 1 [22]. However, it is important to scale only the X data and keep labels the same. It is standard practice for the Scaler to accept the train set as input and then transform both the X_train and X_test.
In my project, I have been using the `StandardScaler` as it creates a better scatterplot to make a line of best fit.

I have plotted two graphs for the unscaled original Boston dataset and the one scaled with the StandardScaler, see next page.



Unscaled Boston dataset



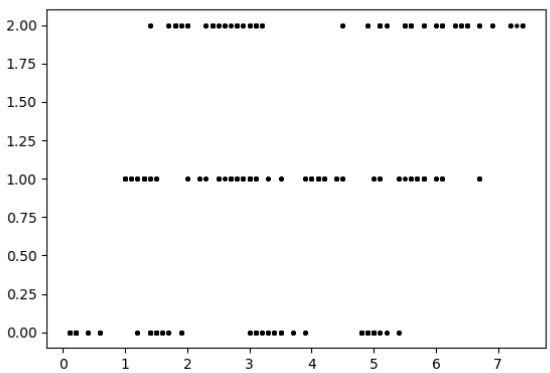
Scaled Boston dataset

Both of these graphs are what I have as a 2D singular plot. It plots all of the data as a single scatterplot cloud in 2D, so some data may be on top of each other, but it may be useful for overall prediction. As can be seen, similar to what the example earlier showed, the data is more spread and looks more normalised instead of the original skews towards the 0-100 bracket, with some more data spread out in 200-400 and around 700 bracket. Instead, the scaler modified the values that for the same labels the data is stretched between -4 and 4. This allows for a more accurate representation of relation between data and also improves the score.

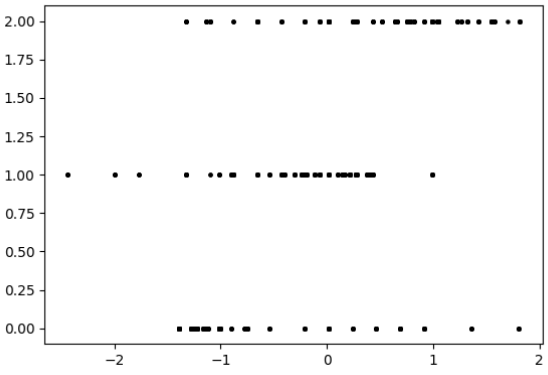
```
Unscaled Boston, Lasso model, alpha = 1:
score = 0.6523622748961698

Scaled Boston, Lasso model, alpha = 1:
score = 0.7090896490066322
```

That is a nearly 6% increase (65% -> 71%) in prediction accuracy! Yet, scalers do not always make much of a difference, and in some specific cases may even even make the prediction worse. It is a case by case scenario that depends on dataset, scaler and parameter choice. For example, for the Iris dataset the shape does not change much and only the data itself is being spread on a scale of -2 to 2 rather than 0 to 7:



Iris unscaled

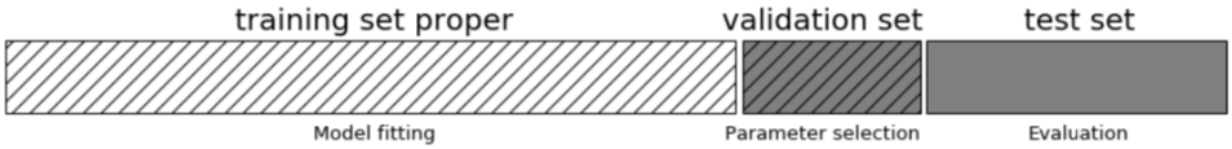


Iris scaled

2.4 Parameter selection, data validation and data snooping

As indicated by the changing scores for KNN, Ridge and Lasso, parameter selection is very important, as it heavily influences the accuracy score, and therefore prediction in a real situation. For this reason, we use the concept of validation set.

The naive way is to use grid search to try all possible parameters on the test data and save the best score. However, we are not allowed to use the test data for building our model, that is called data snooping and the accuracy may not carry over for new data. For this reason, one of the possible approach is to split the training data (both X and y) again with the same 70/30 ratio into *proper* and *validation* sets [22].



For example, for Lasso this is done the following way:

```
alpha_values = np.append(np.arange(0.0, 1.0, 0.01), 1.0)
best_ridge_score = 0
best_ridge_a = 0
X_train_pr, X_valid, y_train_pr, y_valid = train_test_split(self.X_train,
                                                            self.y_train,
                                                            random_state=1513)
for a in alpha_values:
    reg_model = RegressionModel(X_train_pr, y_train_pr)
    reg_model.lasso_fit(alpha=a)
    reg_model.predict(X_valid)
    score = reg_model.score(y_valid)
    if score > best_ridge_score and score ≠ 1.0:
        best_ridge_score = score
        best_ridge_a = a
```

As can be seen, instead of splitting original dataset into training and test sets, we now split training set even further into proper training set and validation set. Essentially, we are not making any predictions yet and still conform to the rule of not using test data, and instead build and test the model purely on the training data by fitting the proper set and making a prediction on the validation set for each new alpha.

2.5 Plotting a line of best fit

The GUI I have built uses the `polyfit()` function to build the line of best fit. How does that happen?
Now that we have our y_pred, inside both my `single_plot()` and `multi_plot()` I save the data as two long lists where I keep putting the same y label into `Y[]` for all of the features in a row. So if X_test and y_pred had something like [a b c d e f] and [y], now for one row the data inside `X[]` and `Y[]` would be [a b c d e f] [y y y y y y]. This allows me to build the scatterplot as well as the line of best fit:

```
self.model.predict(self.X_test)
X = []
Y1 = []
Y2 = []
fig = plt.figure()
for row, y1, y2 in zip(self.X_test, self.y_test, self.model.y_pred):
    for item in row:
        X.append(item)
        Y1.append(y1)
        Y2.append(y2)
    plt.scatter(X, Y1, color='black', s=5)
coeffs = np.polyfit(X, Y2, 1)
m = coeffs[0]
b = coeffs[1]
plt.plot(X, np.dot(X, m) + b, color='red')
```

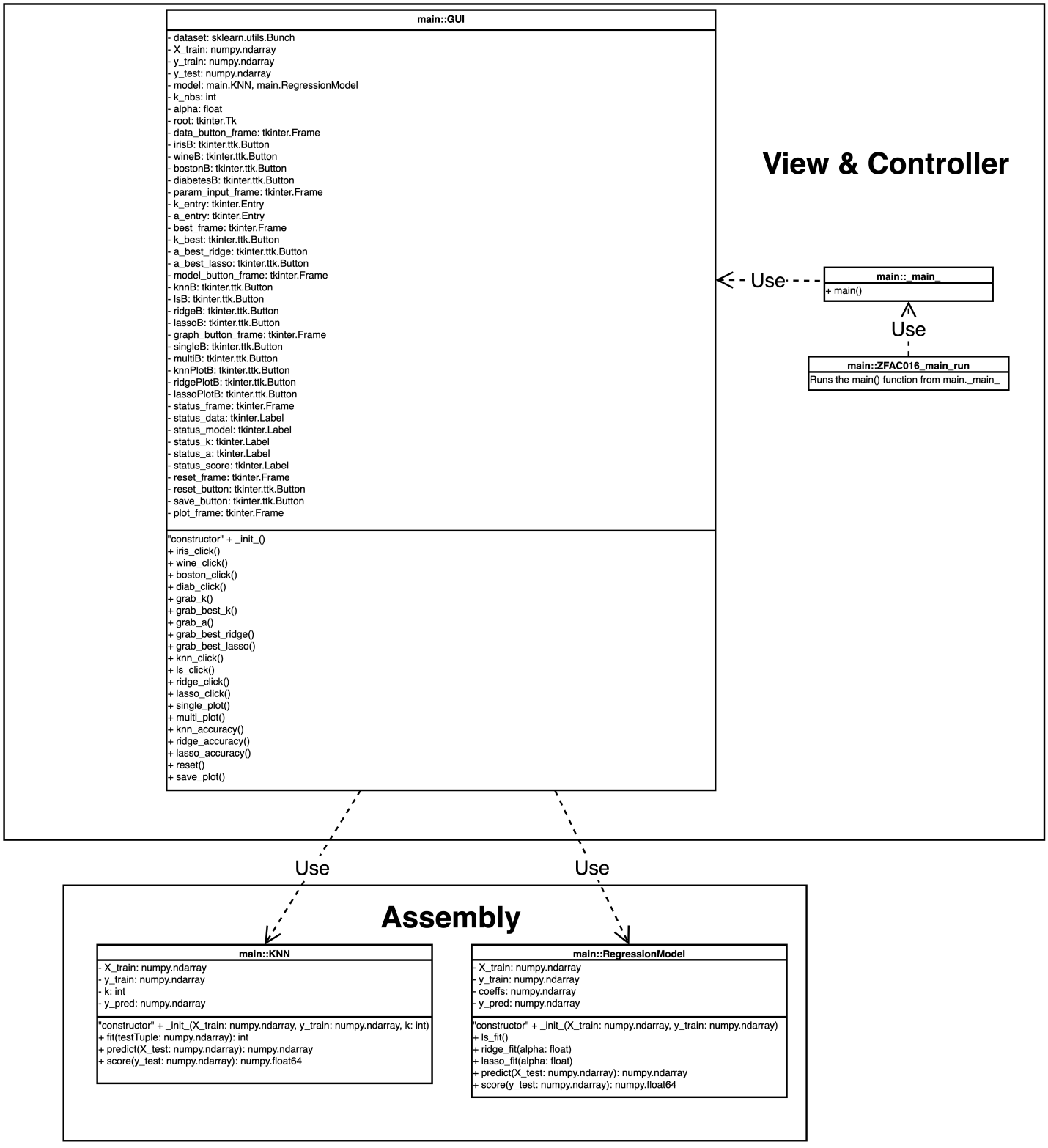
First, the model makes the prediction array inside the model by calling `predict()`. Then, it creates three lists - X for X_test data, Y1 for original y labels and Y2 for predicted y labels. First it goes into a row of X_test, y_test and y_pred, and for each value of X in a row appends the value to X, y_test label to Y1 and y_pred label to Y2. Therefore, instead of having a row with one label, now we have a list where each value of X has a corresponding label. Then, for each row of X_test, the original data is scattered.
After X and Y2 have been filled, we now can build our line of best fit. This works similarly to how Least Squares build its coefficients array for one feature, as essentially we now have one column of X data and one column of y data:

X	Y
a	y
b	y
c	y
e	y
f	y

So, np.polyfit() creates a similar coefficient array using the Least Squares formula $w = \frac{\sum_i x_i y_i}{\sum_i x_i^2}$.
We have one feature and one y intercept, so in this case b is the intercept and m is the only coefficient. And then on top of our scatterplot we can build $y = mx + b$.

And this therefore concludes the theoretical part of this report. In the following chapters, I will describe the design and structure of my Python project and the final deliverable.

Chapter 3:
Software Engineering and Development



3.1 Assembly and backend development

For this project I have been using Python 3.7 and the libraries of NumPy and Matplotlib, with some functions from Scikit-learn for testing the implemented algorithms, and TKInter for GUI design. The project was developed using GitHub VCS, making it easy to track my commits over the course of development, and some times pull older data, compare current and older versions, as well as managing branches.

In first term, the structure of the project had no object-oriented design and all algorithms were simple files that contained the algorithms as a single function. Each algorithm had their own respective test class, however they were not Unit Tests but simply files that printed out outputs:
- for KNN, it tested both classification and regression for k=1 and k=3 on the Iris and Ionosphere datasets and plotted the graph for classification and regression for `0 ≤ k ≤ range(len(X_test),`
- for parametric regression algorithms the tests were conducted on an arbitrary array of some numerical data for simple linear regression and Boston Housing dataset for multidimensional data.

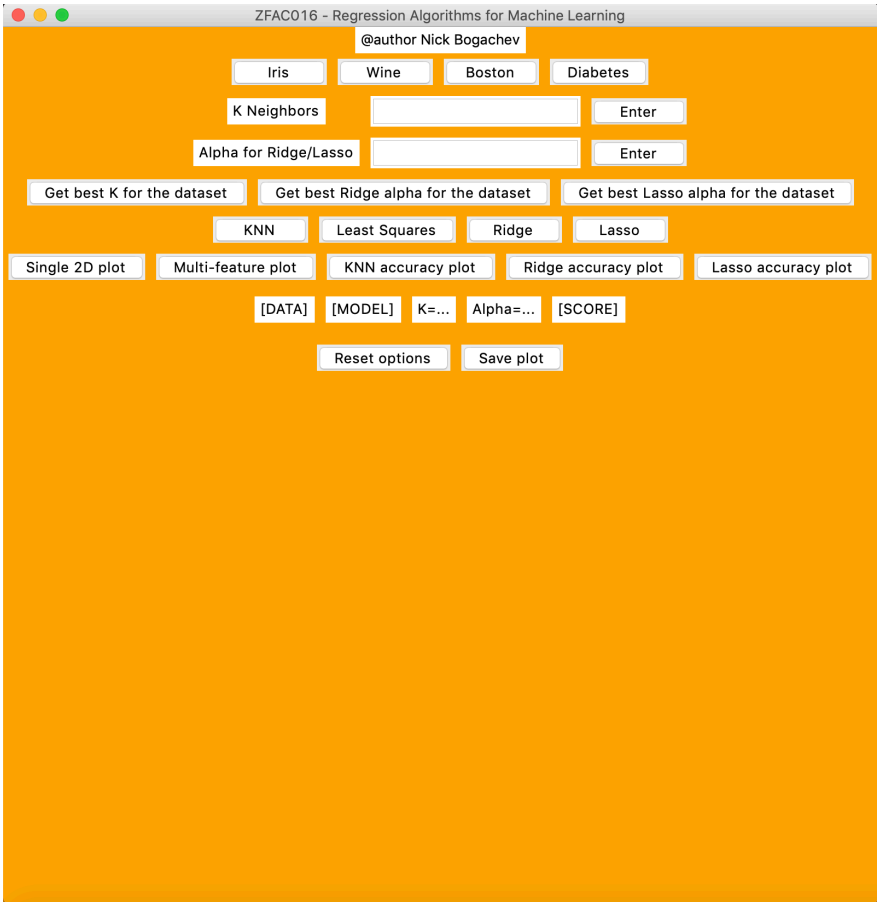
To prove myself successful, I was aiming to bring my algorithms accuracy as close to scikit-learn implementations as possible by comparing the score outputs and adjusting my algorithms until the difference was minimal.

During second term, I have redesigned the structure of the project to follow the specification, finished the Lasso algorithm, wrote unit tests, and created a graphical interface. The project now follows OOP design structure by having classes as objects with an `_init_()` function (the constructor) and various methods/functions, all of which can be seen in the UML diagram above. The two objects, `KNN` and `RegressionModel`, are the only objects of the Assembly section making up the backend. They hold all four of the algorithms described above, yet OLS, Ridge and Lasso are all fit inside the `RegressionModel` as they share functionality and differ only with the logistics of the `fit()` function. Both objects have constructors that initialise their attributes: they both accept X_train and y_train as arguments and create an empty list for y_pred, and then KNN also accepts the k value as an argument. The alpha for RegressionModel is only asked for in `ridge_fit()` and `lasso_fit()` as it is not needed in the constructor.

The testing during second term involved writing Unit tests according to the standards studied in CS2800. My tests check that every function successfully returns a value or if the returned list (like y_pred) is not empty. As we can't know more due to the variety of input data, I have put more effort into visualising the results and having the user decide if they are satisfied with them.

Moving on from testing, the project was developed with using Agile development. During first term, I have been working in sprints of 2 weeks each with regular calls every second Wednesday with my supervisor Prof. Luo, who advised me along the way and pointed me in the right direction often. Due to the amount of coursework in first term, I was doing 2 commits every sprint, so roughly once a week. In second term, I have been slowed down due to the national lockdown and being away from UK, commits were inconsistent and only two calls happened in second term. Despite this, I have successfully finished Lasso algorithm in time and produced a working GUI. More on this in the next subchapter.

3.2 View and frontend development: interactive GUI



This is starting frame of the GUI, developed with TKinter. The GUI object's constructor initialises all the frames, buttons and entry fields as attributes of this object as they need to be callable by the methods of GUI, thus the reason for saving all the interactive elements as attributes of the class.
The methods themselves correspond to each of the 20 buttons of the GUI. This is what they do:

- **Row 1, Datasets**: Four datasets for your choice. Iris and Wine work better with KNN, and Boston and Diabetes work better with the other three algorithms.

- **Row 2, K input**: Entry field for a value of K for K Nearest Neighbors algorithm. The result can be saved to the status bar in Row 7 by clicking Enter.

- **Row 3, Alpha input**: Entry field for a value of alpha for Ridge and Lasso algorithms. The result can be saved to the status bar in Row 7 by clicking Enter.

- **Row 4, Use the validation set method to get best parameters**: Buttons to get the best values for the currently chosen dataset. *They **won't work** if the dataset is not chosen*! The result will be printed in the fields of Row 2 and 3 and then must be saved to the status bar of Row 7 by clicking Enter.

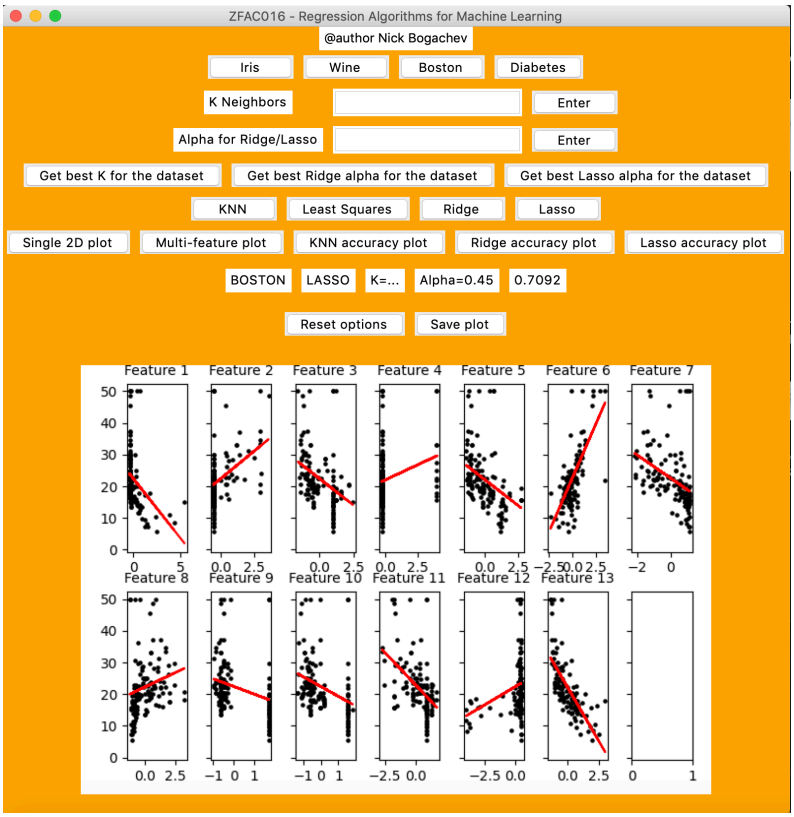
- **Row 5, Algorithm selection**: Choosing one should only happen **AFTER** you have chosen alpha or K and saved it to the status bar, otherwise the algorithm won't be able to make a prediction matrix.

- **Row 6, Plotting**: Choose what plot you wish to see the line of best fit drawn on.
  - **single 2D plot** will make a scatter plot of all data in 2D, sometimes with points on top of each other, and draw a line of best fit thought it. Must have DATA, MODEL and K/Alpha chosen in the status bar.
  - **multi-feature** will build a scatter plot for each feature of the matrix. Each column represent a feature, so each graph will show a given column values with the corresponding y-labels scattered wwith a line of best fit going through them. Must have DATA, MODEL and K/Alpha chosen in the status bar.
  - **Three accuracy plots** to see how the algorithm works on your currently selected dataset with increasing k or alpha. Must have DATA chosen in the status bar.

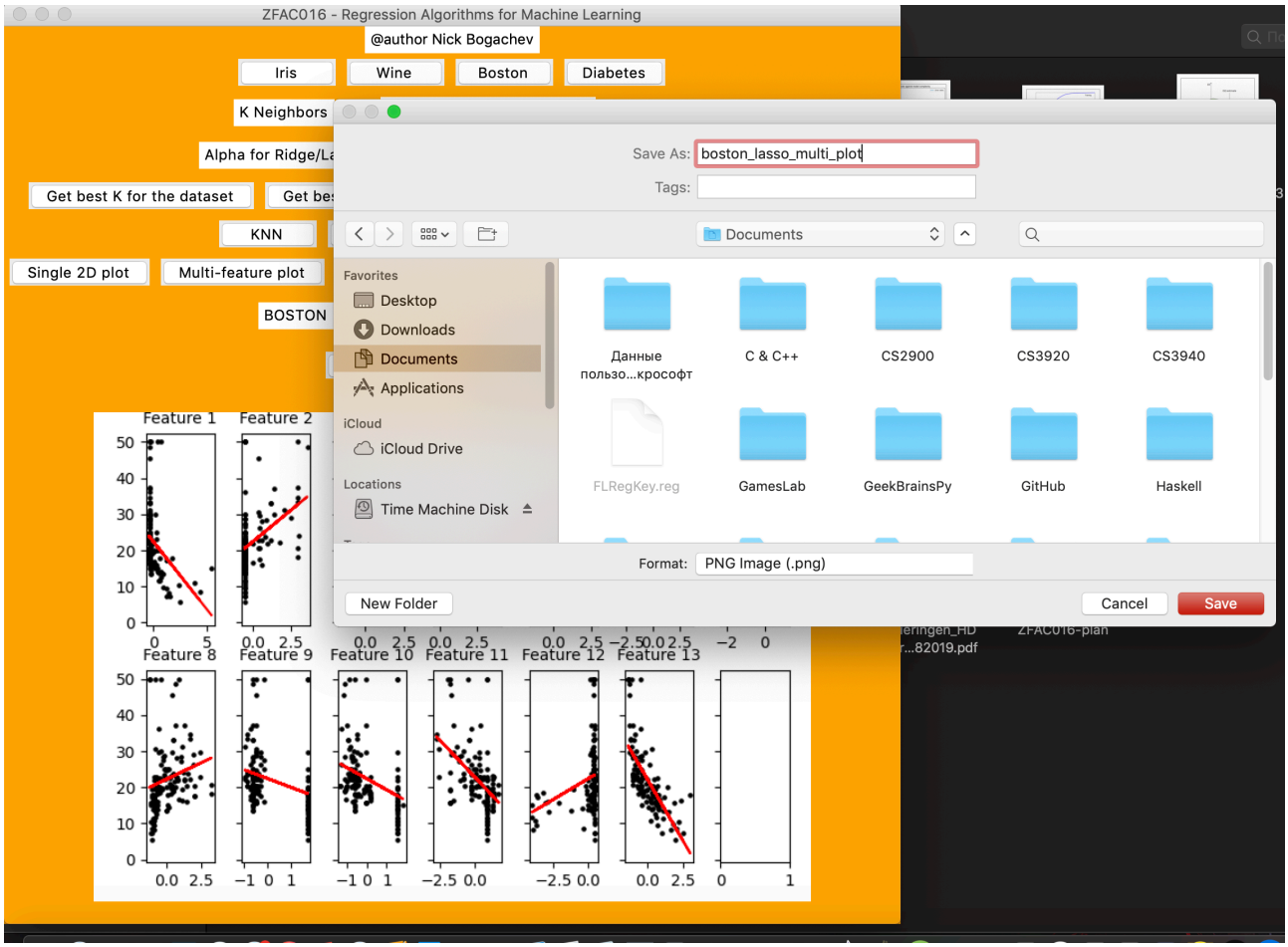
- **Row 7, The status bar**: This is to track your interactions and confirm that the system have saved your choices. Depending on actions currently being performed you can compare the above instructions to this status bar to make sure you have given correct arguments to the system.

- **Row 8, Reset and Save**: Reset your current options and save plot button. Reset button must be clicked before you make a new plot as the frame does not update without a full reset. Additionally, if you think you've made a mistake or something does not work this is works as a safe way to start over. The option to save the currently displayed plot will result in you being prompted with a window that defaults the file type to .png and asks you to select the directory where you would like to save the file and name it. After the GUI is closed, the directory will update with saved plots.

An example output and order of interactions:
- Choose Boston dataset. You will see [DATA] change to BOSTON.
- Lets say you type 0.8 in "Alpha for Ridge/Lasso" and click Enter. You will see Alpha=... change to Alpha=0.8
- This may be too high. Choose "Get best Lasso alpha for the dataset". You will see the field you've just used update with the float 0.04. Click Enter and you will see Alpha=0.8 change to Alpha=0.45.
- Now select Lasso and you will see [MODEL] change to LASSO.
- Click Multi-feature plot. The plot will be shown below and the [SCORE] in the status bar will update accordingly.
- Click Save plot and choose a directory to store it.
- Click Reset options to make a new plot.



The user may want to save the plot, and therefore will be presented with a dialog window to do so by clicking on the save plot button. By default, they will be presented to save it to as a .png file to their default hard drive directory, but on the screenshot below I have selected my Documents folder and gave the file a name of ‘boston_lasso_multi_plot’.



Chapter 4: Evaluation

Overall, I believe the project to be in good shape. All of the planned work has been finished and the program has an interactive UI. In terms of keeping up to the plan, it has deviated a lot since I have written it, as I had virtually zero knowledge of the topic when I wrote it. First term was completely different, starting off with using Python instead of Java to implement the algorithms, influenced by the CS3920 course and the labs program of learning how to use scikit-learn. The content and seemingly easy implementations inspired me to research these algorithms and implement them myself so that they can be used using the same 4-5 line code style. In second term, however, I stuck to my plan - mostly because it was very vague. I spent Christmas holidays researching, as planned, and I was specifically researching Lasso, making sure I wrote down every source I have looked into into my PyDoc comments. It took me till late February to finally understand Lasso, which delayed my ability to finish the project to the standard I planned at first. I have hoped to implement a more interactive UI that would allow to input user data and make predictions without knowing the y_test data. However, the project still fulfilled its purpose and stands up to the specification: Ridge regression is implemented, along 3 other regression algorithms, the program uses real life data (Iris, Wine, Boston and Diabetes datasets) and has a fully interactive graphical interface.

Looking back now, I have made a lot of progress since the start of the year in my ability to use Python, understanding of machine learning algorithms and writing graphical interfaces in Python. I believe I have managed to fulfil the purpose of this project and the final deliverable is able to let the user or project marker to test the 4 algorithms on 4 datasets using either single 2D scatter plot or multi-plot that illustrate each feature separately, creating 32 potential plots just from this data. And this is not counting the extra 12 plots of accuracy of the algorithms on each dataset, bringing the total number of available plots to 44. I believe this to be sufficient amount of visual data to be able to analyse the accuracy and differences between the algorithms, on top of the described differences in *Chapter 2*.

As for professional issues, a most apparent one is that this program cannot be used outside of this project's specification and purpose. It can very well be developed into an interactive prediction machine that accepts user data and plots the prediction, and saves the output predicted labels to the user's computer, as a possible future development, yet the purpose was to focus on the algorithms itself rather than user experience. Another obstacle I have met is of course the time limit. A standard final year dissertation is usually written in 12-30 months, yet actual term time for this course is 5, with the Winter break adding another month. It was sufficient for what I have produced, and I am positive the program serves it purpose, yet the possibilities for development would be much bigger if I had even double that amount.
As to more specific issues is the datasets themselves. While the project does use 4 different datasets, 2 of them are also the datasets I have used to develop and test the algorithms - Iris and Boston. As most development happened in first term, I was tweaking my algorithms until they gave me results I was happy with and/or close to scikit-learn outputs. While on one hand, I am certain I did the algorithms themselves correct as per my testing with scikit, but on the other - tweaking them to work on one specific dataset may not have been good practice. I got lucky that I did not make many errors and that when designing the GUI and including Wine and Diabetes in my code for the first time the programs worked on them.
Finally, the issue of plagiarism. While I made sure to document every source I have used, I am fully aware of the potential consequences of me missing some. For this reason, I have included even the university lectures that we are normally not required to include for our course assignments.

In summary, this project has boosted my knowledge and skills tremendously and was an inspiring research overall. I am looking forward to continuing my studies of machine learning, and would like to give a special mention to Professor Vladimir Vovk for his amazing lectures on the subject, which have largely inspired and influenced this project and aided my research, as well as thank my supervisor Professor Zhiyuan Luo for supporting me along the way.

Bibliography and references

[1] Vladimir Vovk, RHUL CS3920, Chapter 4: General principles of machine learning, p. 3

[2] Vladimir Vovk, RHUL CS3920, Chapter 2: Introduction to machine learning and Nearest Neighbours, p. 14

[3] Tom M. Mitchell, «Machine Learning», Chapter 1, p. 14

[4] Avinash Navlani, KNN classification using Scikit-learn

[5] Vladimir Vovk, RHUL CS3920, Chapter 4: General principles of machine learning, p. 6

[6] [7] Dr. Saed Sayad, KNN Regression

[8] Vladimir Vovk, RHUL CS3920, Chapter 5: Linear regression, p. 17

[9] Samrat Kar, Linear regression

[10] Vladimir Vovk, RHUL CS3920, Chapter 5: Linear regression, p. 5-6

[11] Legendre, Adrien-Marie (1805), Nouvelles méthodes pour la détermination des orbites des comètes [New Methods for the Determination of the Orbits of Comets] (in French)

[12] Wikipedia, Least squares, Regression analysis and statistics section

[13] Author not stated, Stat Trek by Harvey Berman, Regression Coefficients

[14] Hoerl, Arthur E., and Robert W. Kennard. “Ridge Regression: Biased Estimation for Nonorthogonal Problems.” *Technometrics*, vol. 12, no. 1, 1970
PDFs found at https://www.math.arizona.edu/~hzhang/math574m/Read/RidgeRegressionBiasedEstimationForNonorthogonalProblems.pdf and https://www.jstor.org/stable/1267352?seq=1

[15] Author not stated, NCSS, Chapter 335: Ridge Regression

[16] Qshick, Ridge regression for better usage

[17] Santosa, Fadil; Symes, William W. (1986). "Linear inversion of band-limited reflection seismograms". SIAM Journal on Scientific and Statistical Computing. SIAM. 7 (4): 1307–1330.

[18] Breiman, Leo (1995). "Better Subset Regression Using the Nonnegative Garrote". Technometrics. 37 (4): 373–84.

[19] Saptashwa Bhattacharyya, Ridge and Lasso Regression: L1 and L2 Regularization

[20] Wessel van Wieringen, Lasso Regression

[21] Author not stated, HSE, Ridge Regression and Lasso

[22] Vladimir Vovk, RHUL CS3920, Chapter 6: Data preprocessing, parameter selection, and inductive conformal prediction, p. 5, 6, 29, 30.

Appendices

A1: Original project plan

Timeline

By 18th October:
- research the topic thoroughly, read the relevant chapters of «Machine Learning» by Tom Mitchell
- advise the notes on ridge regression by Wessel N. van Wieringen for inspiration, help and possible approaches
- bonus: look into good articles and research papers in my national language to get a better understanding. Ask Prof. Vovk?

18th October - 1st November:
- By the end of October have the first early deliverable ready. Possibly try to develop in Java as it is the language I have the most experience and understanding in.

November:
- Spend first half of November expanding on the project. The early deliverable should have been a «frame» for any algorithm to be fit into using separate methods/classes. UML diagram would be good!
- Spend second half of November clearing up what have been made and working on the interim report.

30th November - 4th December:
- Final bug fixes, getting ready for interim review.

Christmas holidays:
- Research more material if required and get inspiration from people in the IT industry back home. Maybe have someone look at the programs and give a new perspective if that is allowed?
- Start working on final report — combine data from the diary and add post-factum analysis of past records of what has been created so far.
- Make sure to write down all used material and references beforehand.

11th January - 19th February:
- In the first week of 2nd term, present the current work to supervisor, get advice and feedback on progress to define the final sprint.
- Spend time finishing any programs that need adjustments.
- Get the report draft ready - each section must have extensive content in it.

19th February - 26th of March:
- Full engagement with the report, keep track of the word count and try to expand and include as much detail as possible.
- Check quotes, references, bibliography.

A2: Installation instructions and README.txt

Date: 07.04.2021, for version: v2.1

The following project runs on Python 3.7, requires NumPy, Matplotlib, and Sklearn libraries as main dependancies (the full list is available in requirements.txt) and consists of two subdirectories - "main" and "test":
- "main" contains the currently implemented algorithms - KNN and Regression Model (includes Least Squares, Ridge and Lasso) - in the form of classes with the expected _init_, fit, predict and score methods. Additionally, "main" contains the GUI created with TKInter.
- "test" contains test classes for each algorithm accordingly.

The file that needs to be executed is "ZFAC016_main_run.py".

Due to Python not being able to create an executable in the same efficient manner Java does via a .jar file, to be able to use this project it is best to create a virtual environment in the terminal.

If you do not have Python 3 installed, it is best to do so beforehand. The current version can be installed from https://www.python.org/downloads/.
Check you have Python and PIP successfully installed by opening a Terminal window on your machine and typing the following:
- "python3 -V" to confirm you have Python installed and running a correct version (3.7.x)
- "pip3 -V" to confirm you have PIP installed.

Having Terminal opened, navigate to the directory where you have saved this project to:
<username>:cd path/to/project
e.g. if you saved it to Downloads on MacOS, it would be "cd Downloads/IndividualProject_2020_Nikita-Bogachev/".

Now you can create a virtual environment (or venv):
- For Mac and Linux:
  python3 -m venv env
  source env/bin/activate
- For Windows:
  py -m venv env
  .\env\Scripts\activate

Now that you have created and activated the virtual environment, install the required modules into it:
- Mac and Linux: python3 -m pip install -r requirements.txt
- Windows: py -m pip install -r requirements.txt

After the requirements have been installed, you can tell Python to execute the script mentioned above:
- Mac and Linux: python3 ZFAC016_main_run.py
- Windows: py ZFAC016_main_run.py

After a short wait, you will see a GUI in front of you. It may seem daunting at first from all the button and fields, but it is a fairly simple design. You will see the following interaction areas:

- Row 1: Four datasets for your choice. Iris and Wine work better with KNN and Boston and Diabetes work better with the other three algorithms.
- Row 2: Entry field for a value of K for K Nearest Neighbors algorithm. The result can be saved to the status bar in Row 7 by clicking Enter.
- Row 3: Entry field for a value of alpha for Ridge and Lasso algorithms. The result can be saved to the status bar in Row 7 by clicking Enter.
- Row 4: Buttons to get the best values for the currently chosen dataset. They won't work if the dataset is not chosen! The result will be printed in the fields of Row 2 and 3 and then must be saved to the status bar of Row 7 by clicking Enter.
- Row 5: Algorithm selection. Choosing one should only happen AFTER you have chosen alpha or K and saved it to the status bar, otherwise the algorithm won't be able to make a prediction matrix.
- Row 6: Choose what plot you wish to see the line of best fit drawn on.
  - single 2D plot will make a scatter plot of all data in 2D, sometimes with points on top of each other, and draw a line of best fit thought it. Must have DATA, MODEL and K/Alpha chosen in the status bar.
  - multi-feature will build a scatter plot for each feature of the matrix. Each column represent a feature, so each graph will show a given column values with the corresponding y-labels scattered wwith a line of best fit going through them. Must have DATA, MODEL and K/Alpha chosen in the status bar.
  - Three accuracy plots to see how the algorithm works on your currently selected dataset with increasing k or alpha. Must have DATA chosen in the status bar.
- Row 7: The status bar. This is to track your interactions and confirm that the system have saved your choices. Depending on actions currently being performed you can compare the above instructions to this status bar to make sure you have given correct arguments to the system.
- Row 8: Reset your current options and save plot button. Reset button must be clicked before you make a new plot as the frame does not update without a full reset. Additionally, if you think you've made a mistake or something does not work this is works as a safe way to start over. The option to save the currently displayed plot will result in you being prompted with a window that defaults the file type to .png and asks you to select the directory where you would like to save the file and name it. After the GUI is closed, the directory will update with saved plots.

An example output and order of interactions:
- Choose Boston dataset. You will see [DATA] change to BOSTON.
- Lets say you type 0.8 in "Alpha for Ridge/Lasso" and click Enter. You will see Alpha=... change to Alpha=0.8
- This may be too high. Choose "Get best Lasso alpha for the dataset". You will see the field you've just used update with the float 0.04. Click Enter and you will see Alpha=0.8 change to Alpha=0.04.
- Now select Lasso and you will see [MODEL] change to LASSO.
- Click Multi-feature plot. The plot will be shown below and the [SCORE] in the status bar will update accordingly.
- Click Save plot and choose a directory to save it after naming it.
- Click Reset options to make a new plot.

A3: Project diary entries

10.11.2020
New graph plotting function for KNN Classifier.

13.11.2020
Implemented linear regression using two formulas to calculate coefficients. For simple LR, w = E(xy)/E(x2) and b = mean(y) – w * mean(x). For multiple, w[] = (X' X)^-1 X' Y and b = w[0]
14.11.2020 Implemented ridge regression which introduces the notion of alpha. Here we can use a uniform formula for both simple and multiple: w[] = (X' X + αI)^-1 X' Y (where I is the identity matrix of X' X) and b = w[0] Comments: It turned out to be quite tedious to find specifically the formulae for both cases that I could understand, and the ones in lectures seem overly complicated. I quoted both sources in their respective functions. Future goals:

- Remember to come back and implement KNN Regressor (since classifier has no reason to be in this project and currently is a temporary skeleton)
- Lasso regression.
- Try to unify Linear, Ridge and Lasso in one function since I already have duplicate code.
- After this, full focus on report and preparation for interim review.

31.01.2021
Updated the formula for Lasso coefficients. This seems to work better but R2 score stays high even for high values of alpha, while scikit R2 goes negative as we increase alpha. Been stuck on this for months now, coming back to it over and over again and nothing seems to make me understand this any better. Pain. Need to discuss this with supervisor, otherwise when Lasso is finally done all that is left is to redesign the project with OOP in mind, implement graph plotting functions and an interactive UI.

08.02.2021
From "Lasso Regression" by Wessel van Wieringen I have found the following formula: X'X w = X'Y – 0.5 α ż, where ż(j) = sign{[w(α)]j}. Therefore, I believe the Lasso coefficient formula can be written as:

- w(a) = (X'X)^-1 (X'Y – 0.5αz)
- When w(a) 0, z = 1, making the formula to be w = (X'X)^-1 (X'Y – 0.5α), bringing it closer to zero.
- When w(a) = 0, z ∈ [-1,1] (which does not help to make this any more understandable)

A research from HSE presents an interesting and simple formula that builds on the Least Squares method:

- When w(LS) > 0, w(Lasso) = w(LS) – 0.5α.
- When w(LS) < 0, w(Lasso) = w(LS) + 0.5α.
- This formula combined will look like this: w(Lasso) = sign(w(LS) (|w(LS)| – 0.5α)+
- The sign function, if the one in NumPy library is used, will take care of the case when w(LS) = 0, taking the sign as 0 and therefore making w(Lasso) equally zero.

Yet another estimator update, this time based on a study by HSE. Works slightly better, but scikit R2 still does not fall below 0.60 even at high values of alpha. At least mine no longer falls to -233 but to -1.72. Still bad. While every new version of the formula improves the algorithm, nothing seems to work as good as scikit.

13.03.2021
I have made several commits since my last post and a fair bit of progress:

- On February 18th I've added a new formula for Lasso which is the best current iteration yet. If we take the rule that alpha for lasso is between 0 and 1 and a step of 0.001, the produced accuracy graph indicates a 0.45 results for alpha = 0…0.2 and ~0.60 for alpha = 0.6…1. If we use the higher end of alpha, results are somewhat acceptable. I will try to improve it one last time next week as time is running out.
- On February 19th I have redesigned the project structure from bare singular functions to classes (objects) with init/fit/predict/score structure in line with original scikit data. All classes were compimented with the respective Unit tests. In a later commit I included the forgotten AllTests file and "Pydoc" (is this a term? like Javadoc?) to KNN.
- On March 3rd, I have started my UI design. Currently I am not sure what is really expected as the specification says I need a UI, however the aim of the project is to compare results

and accuracy. So should that be a scatter plot with original and/or predicted data? 4 accuracy graphs showing each algorithm accuracy on a given dataset? This is a potential final point for discussion with my supervisor. Additionally, I have an "out there" idea to use PyGame/SimpleGUI to design the UI or at least see if that can be done with buttons and various user input as I have some experience with it, rather than using a fresh new library.
- Last week I have redesigned my Linear Regression model from 3 classes into one where the model can do a standard linear fit, ridge fit or lasso fit as predict and score functions are the same across the board.

The project is nearly finished and I expect to be done with the code around the 20th of March latest and then spend the week fully focusing on the report.

21.03.2021
Work done over the weekend:
- GraphPlot class is finished and is now able to build a variety of graphs: singular plot of all data scattered in 2D with a best fit line or a multiplot that calculates the number of subplots needed to accommodate all features and each subplots scatters the data for one column (i.e. feature).
- GraphPlot also builds 3 accuracy graphs: KNN for R against increasing K, and Ridge or Lasso for R^2 against increasing alpha. No accuracy graph for Least Squares as there is no x parameter we can choose to monitor changes in R^2, it directly depends on the dataset.

Adjustments:
- KNN Regression updated to calculate the MEAN label, not the average.
- RegressionModel's linear_fit() has been renamed to ls_fit() to reflect what it does more accurately. RegModelTest.py has been fixed accordingly.
- KNN classification has been removed as it is not the focus of the project. KNNTest.py has been fixed to reflect this change.

Additionally, during the design of GraphPlot I have discovered that applying a Scaler before using Lasso for prediction normalises the scores and keeps the R^2 between 0.70 and 0.71 for alpha from 0 to 1. A massive improvement!

25.03.2021
Fully functional UI minus the ability to import user data. All the functions from GraphPlot have been moved to the GUI class as button click commands. A detailed instruction on how to use the GUI will be given in the future update of the README.txt file. I am concerned I cannot find a simple and direct way to make the project a double-click-runnable file like I did with the Java calculator in 2nd year (which was literally an "export project as an executable" from either Eclipse or IJ, not sure), and I can't make it run from console either, only works in PyCharm. But hey, it works! Starting screen.

03.04.2021
The coding part is finished and the project is working. Since last post I have attempted to create a PyInstaller executable and (with many obstacles) I did. Only to find out that it will only work on the same OS as the one it was created on. So the decision was to write an instruction on how to open the project in a virtual environment. Additionally, I have removed the option to import user data as it would take more time than I currently have to implement the actual logic of splitting the data from a txt file. However, I have added the ability to save plots so they won't be completely lost on reset. This will be both useful for the report, as well as the end-user. Now after I have finished with the report I will clean up "PyDoc" (as I called it, not sure that is a real name) and make sure the code looks readable.