**Report**

This report will address the following three issues:

* L1 / L2 regularization: describe the difference and the reasons behind it
* Number reversal: implement several approaches in Python, perform a quantitative comparison and write a report describing and explaining the findings
* Two buses problem (the average wait times for buses running once per 20 and 30 minutes): provide the result along with the solution
  1. **L1/L2 regularization: difference and reasons behind it**

When solving the regression problem, we can sometimes face the situation of two features in a sample being linearly dependent. In this case it is guaranteed that there will be such vector of weights that ⟨, ⟩ = 0  for any . In this case, if some is a solution to the optimization problem, then is also a solution for any . Here’s why: considering we have the typical regression function with the regularization (we’ll consider L2 in this explanation, but using L1 instead would also result the same conclusion and proof the needed statement):

Our goal is to minimize this function. Assuming that some is a solution to the optimization problem, let’s look at how the function (1) would look for the value **.**

Let’s separately transform all the parts:

1. Evaluating residual sum of squares

As we can see, it remained unchanged in comparison to the one in formula (1).

1. Evaluating regularization term:

=

Combining all the parts together, we get:

So, this proves that if some is a solution to the optimization problem, then is also a solution for any **,** and thus does not affect the residual sum of squares. The regularization term might change, but since does not affect the data fitting, we can adjust without changing the optimality of concerning the original regression problem.

Thus, the solution not only does not have to be unique, but it can also be arbitrarily large in modulus. This creates computational difficulties. Small errors in features increase greatly when predicting the answer, and in gradient descent, errors accumulate due to operations with too large numbers. Moreover, when two features are linearly dependent, their weights lose their physical meaning. It even may turn out that the growth of weight of some feature which was supposed to lead to the increase of target, will become negative. This makes the model not only inaccurate but also hardly interpreted. To deal with this, the problem is usually regularized.

Another common problem that may occur is overfitting. It occurs when an algorithm fits too closely or even exactly to its training data, resulting in a model that can’t make accurate predictions or conclusion from any data other than the training data. When the model trains for too long on sample data or when the model is too complex, it can start to learn the “noise,” or irrelevant information, within the dataset. When the model memorizes the noise and fits too closely to the training set, the model becomes “overfitted,” and it is unable to generalize well to new data. If a model cannot generalize well to new data, then it will not be able to perform the classification or prediction tasks that it was intended for.

Regularization is one of the simplest yet effective ways to prevent overfitting and improve abilities and accuracy of the model in general. Two most common types of regularization are L1 (or Lasso) regularization and L2 (or Ridge) regularization.

Regularization adds a penalty term to the loss function, discouraging the model from assigning too big weight to any one feature.

Assuming we have a certain number of features and let be the weight of the feature i. So, when adding regularization, we solve the following problem instead of the original one:

where – is the parameter which is called the regularization coefficient

L(f, X, y) – loss function

may be one of the following values depending on what type of regularization we use:

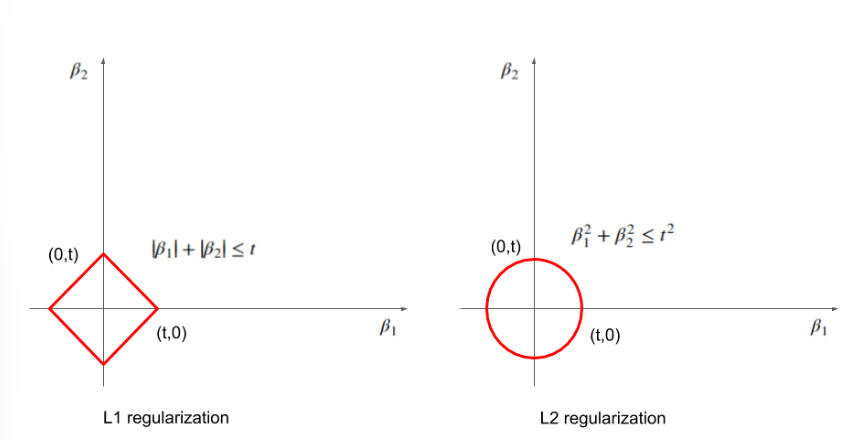
or

**Explaining the difference**

* **Lasso regression** reduces select features weights to zero, thus removing that features from the model
* **Ridge regression** does not enact feature selection. While lasso regression’s penalty term can remove features from the model by shrinking coefficient values to zero, ridge regression only shrinks feature weights towards zero but never to zero which will be discussed later in the report.

To understand the reasons behind that, let’s visualize what happens when we apply two different regularization. The general idea is that we are restricting the allowed values of our coefficients to a certain "region" and within that region, we wish to find the coefficients that result in the best model. It comes from the geometric interpretation of regularization terms in loss function, which shapes will be later described for each of regularizations. Assuming we have two features and with the weights and as follows. Without regularization, our objective is to find the global cost minimum. By adding a regularization penalty, our objective becomes to minimize the cost function under the constraint that we have to stay within our “budget” (let’s mark it **t**).

According to the above mathematical formulation for L1, the equation becomes**| + ≤ t.** As L1 is restriction on the summation of the absolute value (first order) of betas, its available area should be a square with straight edge line given by equation **| +**  **≤ t**. While the ridge regression is described by  the equation  **≤ t,** which means that The L2 is restriction on the summation of the second order of betas, its available area should be a round circle given by ≤ .



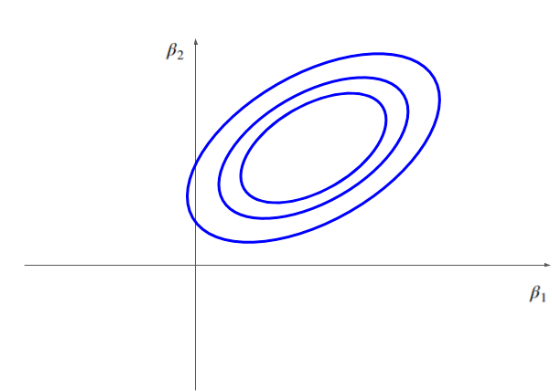
*Figure 1: The shape of available area in L1 regularization and L2 regularization*

Talking about the Loss function, which is supposed to be minimized, is determined as the shape of ellipse – typical for linear regression which is the one considered here due to several reasons:

* It’s widely used and understood, making easier to visualize and explain geometric properties of the loss function. Also the impact of L1 and L2 regularizations can be clearly demonstrated.
* Linear regression’s loss function has a quadratic nature which leads to an ellipsoid kind of lines. And the quadratic loss is immensely popular because it often allows us to derive closed-form expressions for the parameters that minimize the empirical risk and for the expected loss.

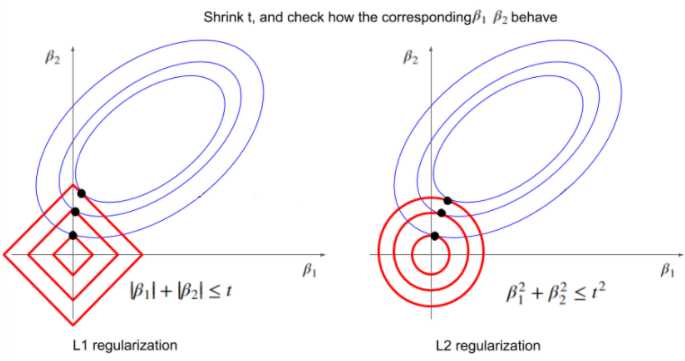
All ellipses have the same short axis to long axis ratio. The size of the ellipse is controlled by beta1 and beta2. Any points on the same ellipse (beta1, beta2) results in the same ellipse size. These lines are the iso-size line or contour line. **By adjusting beta1 and beta2, we can get smaller or larger size of ellipse, or in other words, smaller or larger Loss function.**

**The following figure illustrates the shape of Loss function:**



*Figure 2: The shape of Loss function*

With restrictions on the choices of beta1 and beta2, we need find the smallest Loss among them. The smallest reachable loss is the smallest ellipse (blue) tangent to the available area (red). Now, let's try shrink the t, and see how the corresponding (beta1, beta2) behave.



*Figure 3: Shrink t, and check how the corresponding beta1, beta2 behave*

With the decease of t, both methods get smaller beta1 and beta2. Here comes the difference between L1 and L2: the tangent point for L1 cases is more likely to be achieved on the axis (some parameter be reduced to zero), while the tangent point for L2 cases is more likely to be got on a none-axis point (none zero points).

The math behind it may due to the fact that the first order derivative is a constant value for L1's straight edge, except for its sharp vertex which doesn't have the definition of the first order derivative. While in L2, the curve (it is the circle) is smooth and always has the first order derivative on any points on the red circle. To find the tangent point is to match the points where they share the same slope (first order derivative) on both the edge of available area (red diamond) and the target ellipse (blue ellipse). When we shrink the t and construct a new available area (red), the area in L2 method (red circle) can always find some points on its curve satisfying our requirement, the points may be on the axis (zero for some beta), maybe not (none zero for some beta). When it comes to L1 method, the slope on the straight edge of available area (red diamond) is constant. If 't' is relatively larger, it means beta1 and beta2 has wider range on the contour line to match the point with the same slope. But if the 't' is small enough, there is no chance for the L1 method to match the points in such a tiny range. The only possible choice left is the vertex, where there is no definition of slope, so it can be any slope and match any points on the blue ellipse.

That is why L1 always have tangent point on the axis (or say vertex) and dropout coefficients by setting them to zero after shrinking the regulation, and L2 typically results a none-zero but tiny coefficients. The core reason L2 regularization does not set coefficients to exactly zero lies in the smooth, continuous nature of the penalty term and its gradient, leading to gradual shrinkage rather than creating the conditions for exact zeros. Let’s discuss it more precisely:

* the squared value function is differentiable everywhere.
* the gradient of λ​ is 2λ​

This leads to a smooth gradient that continuously shrinks the coefficients towards zero but never actually sets them to zero. The penalization is proportional to the value of the coefficient, resulting in a gradual reduction rather than an abrupt setting to zero.

However, in practical applications with finite precision arithmetic, it's theoretically possible for very small coefficients to become numerically zero, especially if they are extremely close to zero. But this is due to computational precision limits rather than the regularization process itself.

**Summary and conclusion:**

Regularization techniques like L1 and L2 play a crucial role in the development of robust machine learning models. They help in preventing overfitting, enabling models to generalize better to unseen data. The choice between L1 and L2 regularization depends on the dataset characteristics and the specific requirements of the problem.

L1 regularization is key for feature selection and model interpretability, so it may be effective to use it in the following cases:

* when there’s high-dimensional data with many features, but we suspect only a few are actually important
* feature selection is crucial, as L1 can drive the coefficients of irrelevant features to zero
* the model needs to be interpretable, and we want to identify the most significant features: L1 tends to select one feature from a group of correlated features and set the others to zero, which can help with interpretability but may ignore the combined effect of correlated features.

Meanwhile L2 is more about controlling model complexity complexity and preventing overfitting through a balanced contribution of all features and should be used when:

* You have a smaller number of features, and you expect all of them to influence the output.
* You prioritize prediction accuracy over interpretability: distributes the penalty across correlated features, leading to more stable coefficients but potentially less interpretability as it keeps all features.

There are also may be some problems with interpreting linear models and therefore both regularizations may be used to try solving them. We can compare both regularizations on different features:

### ****Complexity and Interpretability****

* **L2 Regularization**: Leads to models where all features contribute a little, making it harder to interpret the importance of each feature individually.
* **L1 Regularization**: Can improve interpretability by setting some coefficients to zero, but the remaining non-zero coefficients can still be affected by multicollinearity.

### ****Effect of Regularization on Interpretation****

* **L1 Regularization**: The sparsity induced can simplify interpretation by reducing the number of active features. However, it might give an incomplete picture if important features are correlated and some are set to zero.
* **L2 Regularization**: Provides a more balanced model where all features contribute, but the contribution of each feature might be harder to interpret due to the distributed shrinkage.

Concluding, L1 and L2 do not solve all challenges related to the interpretation of linear models. The choice between L1 and L2 regularization should consider the specific needs for interpretability and the nature of the dataset. For high-dimensional data with many features, L1 is preferable for its feature selection capability. For datasets where all features are expected to contribute to the output and the focus is on predictive accuracy rather than interpretability, L2 is more suitable.

**Dealing with multicollinearity:** multicollinearity (high correlation among features) is a common problem that may occur and both of the regularizations are capable of dealing with it. It is commonly said that it is more efficient to use L2 regularization for the following reasons. By penalizing the magnitude of the coefficients, L2 regularization ensures that the model does not become overly reliant on any single feature, thereby maintaining a balance in the contribution of all features. This characteristic of L2 regularization is especially valuable when dealing with multicollinearity, as it helps in distributing the effect of correlated variables across multiple features, enhancing the model’s stability and performance. Shortly, L2 can reduce the variance of the coefficient estimates, which are otherwise inflated due to multicollinearity. This stabilization makes the model's predictions more reliable. L1 regularization can address the multicollinearity problem by constraining the coefficient norm and pinning some coefficient values to 0. It makes it possible to automatically get rid of features that participate in the relations of approximate linear dependence, respectively, saving from the problems associated with multicollinearity.

**Case when x1=x2. How would the corresponding weights behave in case of L1 and L2 regularization?**

As it was said earlier, L1 adds sum of absolute values of the weights to the loss function. So, considering the case when , loss function will be penalized by the regularization term . L1 tends to drive some of the coefficients to zero to create a sparse model. Therefore, to minimize the penalty, L1 will drive either or (or maybe both of them) to zero to enforce sparsity. Shortly, it will attempt to combine the influence of and into a single coefficient, potentially setting one of the weights to zero. Geometrically (as it was explained earlier): when the contour lines of the loss function intersect with the L1 norm constraint, they often touch the constraint boundary at one of the axes. This results in many coefficients being exactly zero, leading to sparsity. Imagine an ellipsoid representing the loss function intersecting a diamond-shaped constraint. The points of intersection are more likely to occur at the corners of the diamond (where some weights are zero).

Talking about L2, it adds the sum of squares to the loss function. When , the regularization term λ() penalizes the sum of the squares of the coefficients. L2 regularization tends to shrink the coefficients but does not force them to zero unless necessary. Therefore, in the case where , ​L2 regularization will typically balance the weights and ​ in such a way that their combined effect is minimized under the penalty. This means that the weights will be shrunk towards zero, but not necessarily driven to zero. The regularization will distribute the "weight" of and across both and ​ in a balanced manner. Geometrically (as it was also mentioned earlier): when the contour lines of the loss function intersect with the L2 norm constraint, they touch the constraint boundary at more rounded points. This means the coefficients are generally shrunk towards zero but not exactly zero. Imagine an ellipsoid representing the loss function intersecting a spherical constraint. The intersection occurs at more evenly distributed points, leading to all weights being shrunk but not to zero.

* 1. **Number reversal**

**Problem formulation:** implement several approaches in Python, perform a quantitative comparison and summarize everything in the report.

In this part of the report two implementations will be analyzed and compared.

***Implementation 1*** *(using string reversal)****:***

**def string\_reversal**(number):

st = **str**(number)

st\_rev = st[::-1]

**return int**(st\_rev)

This implementation is based on converting the number into string, then reversing it using slice operation and finally converting the result string back into integer.

***Implementation 2*** *(using the while loop and the division):*

**def reversal\_using\_loop**(number) :

rev\_num = 0

**while** number > 0:

digit = number % 10

rev\_num = rev\_num \* 10 + digit

number = number // 10

**return** rev\_num

This implementation is pretty logical and straightforward and uses the while cycle. At first we initialize **rev\_num** variable which will store the final number. The loop continues as long as the ‘number’ is greater than 0. The next step is to extract the value of last digit, renew the rev\_num and then update the original number. As the loop ends, **rev\_num** variable contains the reversed number.

**Quantitative comparison:**

Quantitativecomparison will include the following components:

* Time complexity
* Space complexity
* Execution time

**Time complexity:**

*Implementation 1:*

Converting number to string requires O(n) time as well as string slicing and transforming string back to integer; step to step explanation of that:

* **str(number):** includes processing each digit of the integer => considering our number has n digits, this operation takes O(n) time.
* **st\_rev = st[::-1]:** includes iterating over each character of the string **st** and constructing the reversed string. Since **st** has n characters, this operation also takes O(n) time.
* **int(st\_rev):** includes iterating over each string character and parsing it and constructiong the final integer; so it also takes O(n) time.

So, the overall time complexity of the implementation O(n), where n is the number of digits in the input number.

*Implementation 2:*

All the operations performed inside the loop take O(1) time:

* **digit = number % 10:** the modulus operation computes the remainder of the division, which is a constant-time operation for fixed-size integers
* **rev\_num = rev\_num \* 10 + digit:** multiplication and addition of fixed-size integers are constant-time operations
* **number = number // 10:** Integer division is also a constant-time operation for fixed-size integers

The while loop iterates exactly n times (n – amount of digits in the number) and each iteration performs operations that take O(1) time. Therefore the loop in general takes O(n). That makes the time complexity for this implementation O(n) where n is the amount of digits of the input number.

**Space complexity:**

*Implementation 1:*

Converting integer to string requires O(n) space, where n is the number of digits of the input number. Space complexity of string slice is also O(n) due to the fact that the new string created at worst can be the same length as the input string if there are no letters removed. Which makes the total space complexity of this implementation is O(n), where n is the number length of the slice list.

*Implementation 2:*

Due to the fact that this implementation does not use any additional data structures meaning there is no extra space used for reversing the number and uses only constant amount of space for variables, the space complexity is O(1).

**Execution time:**

To measure the execution time for both implementations I decided to choose the **timeit** module and if to be more precise the **timeit()** function of this module. It creates a [Timer](https://docs.python.org/3/library/timeit.html#timeit.Timer) instance with the given statement, setup code and timer function and run its [timeit()](https://docs.python.org/3/library/timeit.html" \l "timeit.Timer.timeit" \o "timeit.Timer.timeit) method with  number executions. The optional globals argument specifies a namespace in which to execute the code. The timeit module is accurate and provides a simple way to time small bits of Python code. It has both a [Command-Line Interface](https://docs.python.org/3/library/timeit.html#timeit-command-line-interface) as well as a [callable](https://docs.python.org/3/library/timeit.html#python-interface) one. It avoids a number of common traps for measuring execution times.

Execution times were measured for different values of length of input number which may also be an indicator of scalability. A piece of code with the usage of timeit() function:

time\_string\_reversal = timeit.timeit(lambda: string\_reversal(number), number=1)

time\_reversal\_using\_loop = timeit.timeit(lambda: reversal\_using\_loop(number), number=1)

Result length = 5:

Execution time for implementation 1: 7.007000021985732e-06

Execution time for implementation 2: 6.037000275682658e-06

Result for length = 25:

Execution time for implementation 1: 6.837000000814442e-06

Execution time for implementation 2: 1.4418999853660353e-05

Result for length = 50:

Execution time for implementation 1: 7.575999916298315e-06

Execution time for implementation 2: 2.703600057429867e-05

Result for length = 100:

Execution time for implementation 1: 6.2339995565707795e-06

Execution time for implementation 2: 3.376299991941778e-05

As it is seen from the results, even though when the length of the input number is small execution times are very close to each other, when it comes to increasing the number of digits, the difference becomes more significant and shows that the first implementation is more effective in terms of taken time. It is also noticeable that the scalability of the first implementation is pretty good due to the fact that there’s no significant increase in execution with increase of length.

However it is also seen that sometimes the execution time sometimes decreases as the number length increases which is not very expectable and logical. This may happen because of some system python characteristics maybe or background processes that impact available CPU time for testing code. Another reason may be time resolution: the ‘timeit’ module relies on the system clock, which has finite resolution. Small timing differences may not be accurately captured, especially for very fast operations. As a possible solution to this issue, we can run code a few times before taking measurements to make results more stable.

* 1. **Two buses problem**

**Problem formulation:** findthe average wait times for buses running once per 20 and 30 minutes.

**Solution:**

As it is said in the task, each bus comes to the station strictly once in a certain amount of minutes. Therefore, it may be concluded that the waiting time for each bus may be seen as a uniformly distributed random variable (labeling of the uniform distribution:  **R[a, b]**, where **a** is the smallest possible value of the random variable and **b** – the biggest). Now let and be the random variables for waiting time of the first and second buses respectively. The way they are distributed as well as a distribution function for each of the variables are as following:

Now let **t (t ≤ )** be the time period from the moment a person comes to the bus station to the next bus to arrive. Let’s take a closer look at the following fraction : it is the probability of the bus i to arrive to the bus station during the waiting period t.

Here’s the proof of it:

Where **x** is a random variable, which reflects the waiting time.

Therefore, the is the probabilityof the bus i not to arrive during t minutes of waiting. Which means that if the value of does not belong to a subinterval [0, ] of the interval [0, ], then the probability of a general waiting time (T) to be greater than t is counted by the following formula:

Thus we have,

With and the final value of E(T) which is the average wait time (because expected value is the average value of random variable) is

**Answer:** Average wait time

Used websites and literature:

Problem 1 (L1/L2 regularization):

https://www.linkedin.com/pulse/intuitive-visual-explanation-differences-between-l1-l2-xiaoli-chen

<https://ethen8181.github.io/machine-learning/regularization/regularization.html#Reference>

<https://www.lexjansen.com/wuss/2018/131_Final_Paper_PDF.pdf>

https://www.statlect.com/glossary/loss-function

Problem 2 (Number reversal):

<https://docs.python.org/3/library/time.html>

https://superfastpython.com/time-time-vs-time-perf\_counter/