6.867: Homework 1

Anonymous authors

***Abstract*— This is Machine Learning homework 1. Topics are gradient descent, linear basis function regression, ridge regression, sparsity and LASSO. All work has been done in Python.**

I. Gradient Descent

In this section, we implemented Gradient Descent to find the minimum of a function.

A. Part 1

Our Gradient Descent (GD) implementation begins with an initial guess, x_0, of the argument x_{min} that minimizes the function, $f(x)$. The implementation is applied to two functions, a negative Gaussian and a quadratic bowl. The gradient, df/dx, can be calculated in closed form (for simple functions), and the next estimate of x_{min} is updated according to the rule:

$$x_{i+1} = x_i - \alpha \cdot \frac{\partial f(x_i)}{\partial x} \tag{1}$$

Here, α is the step size, or learning rate, which controls how much the gradient affects the next estimate. The rule is applied until either a pre-determined maximum number of iterations, n_{iters}, occurs, or the function's value at the current iteration differs from the previous iteration's estimate by less than ϵ. Each of these paramters, (α, n_{iters}, and ϵ) impact the GD solution in different ways.

In Fig. 3, the intial guess, x_0, is varied to show that a close intial guess leads to convergence in only a few steps. As the norm of difference between initial and final estimates grows, the number of iterations increases. For certain initial guesses, we observed that the algorithm did not converge in some maximum number of iterations. For the negative Gaussian, this is because the gradient is very small far from the mean.

The convergence limit determines when the GD algorithm stops, seen in Fig. 2. The error increases if ϵ is too large, as on the right of Fig. 1.

In GD, the norm of the gradient decreases as the solution converges, which is shown for the quadratic bowl function in Fig. 5.

B. Part 2

For more complicated functions, the gradient is not always as easy to compute. Instead, we can use finite difference approximation instead of a closed form gradient. We have used the central difference approximation, because it balances the gradient between the forward and backward direction and thereby often results in a more precise output. The central

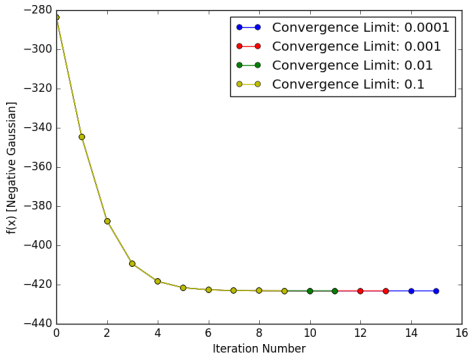

Fig. 1: As convergence limit, ϵ, increases, the iteration stops earlier. From this view, the four solutions differ very little.

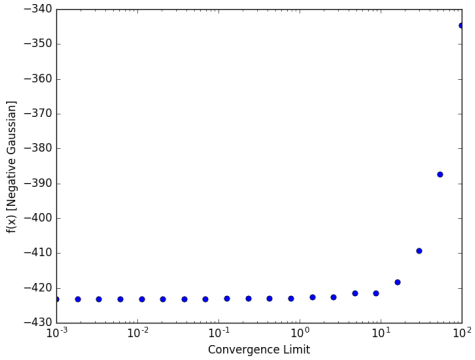

Fig. 2: The final value returned by GD can differ substantially if convergence limit is set too loosely.

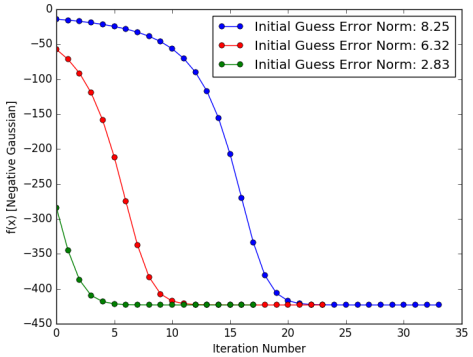

Fig. 3: When initial guess x_0 is far from true minimum, it takes many iterations (blue) to converge. As initial guess approaches final solution, number of iterations decreases.

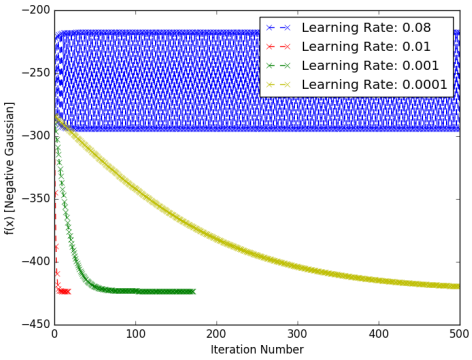

Fig. 4: If learning rate, α, is too large (blue), the solution will oscillate or diverge each iteration. If α is too small, it will take many iterations to converge (yellow). Convergence takes only a few iterations for proper α (red).

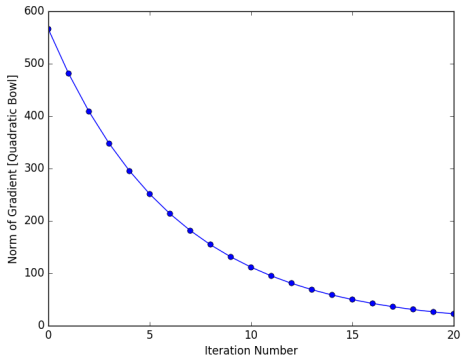

Fig. 5: Norm of gradient decreases with each iteration in GD. This example minimizes the Quadratic Bowl function.

difference approximation of a function with a two dimensional input vector (x, y) and the step size δx and δy is given by:

$$\nabla \boldsymbol{f}(x,y) = \begin{bmatrix} f_x & f_y \end{bmatrix} \approx$$
$$\approx \begin{bmatrix} \frac{f(x+0.5\delta x,y)-f(x-0.5\delta x,y)}{\delta x} & \frac{f(x,y+0.5\delta y)-f(x,y-0.5\delta y)}{\delta y} \end{bmatrix}$$

Table I-B shows that the difference between approximated gradient and analytic gradient is small across a range of (x,y) values for both function types. The choice of difference step $\delta x = 0.01$ is what led to this small error. In Fig. 6, the norm of the difference between the real and approximate gradient gets quite large when δx is also large.

C. Part 3

Next, we consider two types of gradient descent on a dataset. Batch gradient descent (GD) calculates the gradient at each step using every sample in the data set. Stochastic gradient descent (SGD), instead uses a single sample each iteration, looping through a random ordering of the dataset to add some stochasticity. We implemented both algorithms and evaluated them on the same data set, with the same initial guess. For SGD, the equation (1) is modified to accept a single sample, $(x^{(j)}, y^{(j)})$ as input to the gradient:

$$\theta_{i+1} = \theta_i - \eta_i \cdot \nabla_\theta J(\theta_i; x^{(j)}, y^{(j)}) \tag{2}$$

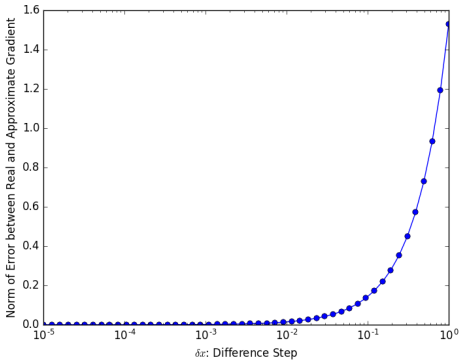

Fig. 6: Norm of gradient approximation error increases as δx increases. This plot is for the Negative Gaussian.

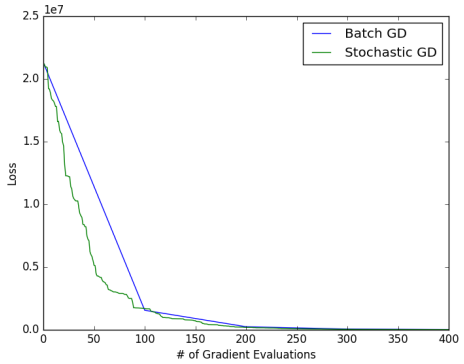

Fig. 7: Stochastic Gradient Descent (green) initially approaches convergence faster than Batch Gradient Descent (blue). Batch GD monotonically decreases in norm error, but SGD randomly oscillates and sometimes increases. Batch GD looks piecewise linear, because each iteration involves 100 (size of dataset) evaluations of the gradient, whereas SGD only evaluates once per iteration.

We use a common form of learning rate parameter $\eta_i = (\tau_0 + i)^{-k}$ to guarantee convergence with $\tau_0 = 10^8$ and $k = 0.6$.

In Fig. 7, Stochastic Gradient Descent (green) initially approaches convergence faster than Batch Gradient Descent (blue). Batch GD monotonically decreases in norm error, but SGD randomly oscillates and sometimes increases. Batch GD looks piecewise linear, because each iteration involves 100 (size of dataset) evaluations of the gradient, whereas SGD only evaluates once per iteration. So, SGD intially learns faster than Batch GD.

After many iterations, shown by Fig. 8, Batch Gradient Descent catches up to Stochastic Gradient Descent and is more accurate. SGD seems to converge within a range, but continues to oscillate as long as learning rate is non-zero. So, Batch GD has better accuracy than SGD. The tradeoff here is speed vs. accuracy.

II. Linear Basis Function Regression

In this section, we consider two types of linear basis functions for regression: polynomial and cosine. We try different model complexities to compare the performance.

TABLE I: Norm of gradient approximation error increases as δx increases. This plot is for the Negative Gaussian evaluated at (3,4).

Gradient vs Finite Difference ($\delta x = 0.01$)				
Function	Gradient	Approximation	Error	Error Norm
Neg. Gaussian: $f(3,4)$	[-4.2258, -3.6221]	[-4.2375, -3.6299]	[-0.01178, -0.00785]	0.0141
Neg. Gaussian: $f(12,4)$	[11.455, -34.365,]	[11.472, -34.440,]	[0.017, -0.074,]	0.0764
Quad. Bowl: $f(6,2)$	[-330, -350]	[-329.95, -349.95]	[0.05 0.05]	0.0707
Quad. Bowl: $f(10^3,90)$	[10050, 5500]	[10050.05, 5500.05]	[0.05, 0.05]	0.0707

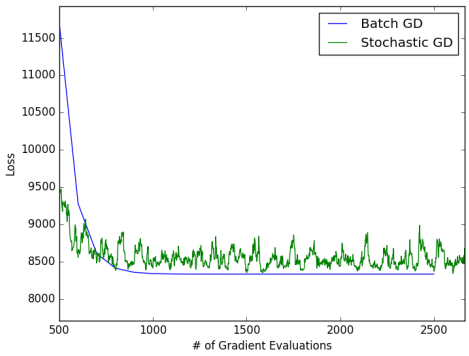

Fig. 8: After many iterations, Batch Gradient Descent catches up to Stochastic Gradient Descent and is more accurate. SGD seems to converge within a range, but continues to oscillate as long as learning rate is non-zero.

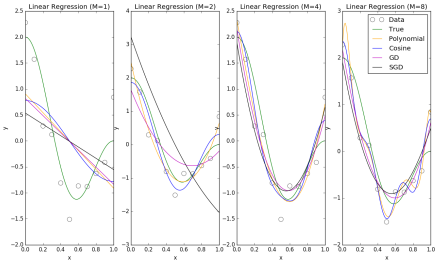

Fig. 9: For different M values (1,2,4,8) a polynomial (orange) and cosine (blue) basis is used to fit the dataset (black circles). The true function used to generate the data is shown in green in each plot. For low model complexity (left plot), the model doesn't capture the variation in the data. For high model complexity (right plot), the model overfits the data. Also shown is Batch GD (magenta) and SGD (black) on SSE.

A. Part 1

At first, we don't know the true basis for our dataset, and want to try *polynomial* models of different complexities. Given an array of data points X and a vector Y of values, a simple polynomial basis,

$$\phi_0(x) = 1, \phi_1(x) = x, ..., \phi_M(x) = x^M \tag{3}$$

is applied to the X points, for several values of M.

Fig. 9 shows the data as black circles, and the true function from which the data is generated in green. The orange in each subplot is the polynomial basis model, and blue is a cosine model (described in [pt 4]). (The other curves will be described later.) The model complexity is insufficient to capture the variation in the data with M=0 Fig. 9 (leftmost) since the model is a line with zero slope. For M=1, the linear-plus-constant basis, is still insufficient to capture the concave structure in the dataset. M=2 seems like a good choice visually, as it does not intersect every data point exactly but captures the concave shape. On the rightmost plot, M=10 overfits the data, which means it evaluates exactly to the correct values in the dataset but does not look like the smooth, green line from which the data was generated.

B. Part 2

Next, we implemented a calculation sum-of-squares error (SSE) (Bishop 3.26):

$$E(w) = \sum_{i=1}^{N} (y^{(i)} - w^T \cdot \phi(x^{(i)}))^2 \tag{4}$$

and its gradient. To confirm that the analytic gradient is close to the approximate gradient, we measured the norm difference between the two methods for a single run of Batch GD. The average norm error (over 100 iterations) was 0.0119 with $\delta x = 0.01$. This implies the gradient was likely implemented correctly.

C. Part 3

We then tried Batch GD on the SSE function. It was sensitive to the intial guess, and would not always converge to a good solution if initial guess was far off. This could be because the SSE function is more complicated than the simple negative gaussian and quadratic bowl from the first problem. Similar to previous experience with GD, if learning rate was too high, it would diverge and if learning rate was too low, the parameters would not change enough between iterations.

Fig. 9 shows the Batch GD solutions compared with other methods. Batch GD (magenta) and does not seem to overfit the data as much at high model complexity (right side), but does not capture the trend as well as regression methods (left-middle) for low complexity. The Batch GD parameters used across models are $\epsilon = 10^{-3}$, $\alpha = 0.01$, and $n_{iters} = 100$.

For SGD (black in Fig. 9) the performance is similar to Batch GD. It also depends on initial guess being close to final answer. The SGD parameters used across models are $\epsilon = 10^{-8}$, $\tau_0 = 10^8$, $k = 0.6$, $n_{iters} = 10000$. The convergence limit for SGD had to be set lower than in Batch GD to ensure randomness does not cause the algorithm to terminate very early. Note there is a scaling factor (dataset size) between the two GD algorithm's n_{iters} in our definition which explains the large difference. The M=2 SGD solution does not look very accurate. Presumably this particular model's parameters could be adjusted for good performance but it demonstrates

the brittleness as is.

D. Part 4

Since we know the dataset was actually generated from a sum of cosines, a set of cosine basis functions should lead to a better fit than polynomials. The cosine basis functions with intercept term ϕ_0 are:

$$\phi(x) = (1, cos(\pi x), cos(2\pi x), ..., cos(M\pi x)) \quad (5)$$

Again referring to Fig. 9, there is a similar trend (in both polynomial and cosine bases) of underfitting and overfitting for small and large model complexities, respectively. The M=2 cosine model fits the data pretty well, similar to the M=2 polynomial. This is no coincidence, because the data was generated from a pair of cosines (M=2), specifically $y = cos(\pi x) + cos(2\pi x)$.

The actual values of the weight vector's elements can also be compared to the true function in Table II. The M=2 case does not give the exact same values as the actual function's coefficients, but this could possibly be improved using regularization, more data, data with lower noise, or removing the bias (w_0) term. With the complex model (bottom row, M=8), the high frequency components (7, 8) have very low magnitude.

III. Ridge Regression

A. Introduction

Ridge regression is a form of regularization. In regularization high values of the weight coefficients w are penalized. If the weights are of high value it is likely that the generated model is overfitted. The penalizing term is added to the cost function of linear regression, which gives the regularized error of general regularization (Bishop 3.29):

$$\frac{1}{2}\sum_{n=1}^{N}((t_n - \boldsymbol{w}^T\phi(\boldsymbol{x}_n))^2) + \frac{\lambda}{2}\sum_{j=1}^{J}|w_j|^q \quad (6)$$

N is the total number of samples, t_n is the target vector, also referred to as y, w is the vector of weight coefficients, $\phi(x)$ is the feature map, x_n the samples, λ is the regularization coefficient, J is the number of elements of w. Setting $q = 2$ gives the quadratic regularizer ridge regression, while $q = 1$ gives the LASSO estimator.

Again, the feature map $\phi(x)$ is given by a simple polynomial basis with intercept term as in (3).

Given the feature map $\phi(x)$ and training set (x_i, y_i) we want to find the coefficient w which minimizes the regularized error (6). To do so, we compute the gradient of the regularized error, set it to zero and solve for w. For the quadratic regularizor, there is the following closed-form solution existing (Bishop 3.28):

$$\boldsymbol{w} = (\lambda\boldsymbol{I} + \boldsymbol{\phi}^T\boldsymbol{\phi})^{-1}\boldsymbol{\phi}^T\boldsymbol{t} \quad (7)$$

B. Training and testing on the same data

In this section a model is generated from a given training set and tested on the same dataset. After having determined the weights through (7), we apply the weights to determine the hyperparameters λ and M. Fig. 10 shows that linear regression (λ close to zero) and ridge regression with a low λ visually fits the dataset well. This is the case for less complex models with lower degree polynomials. Once the model complexity increases (i.e. M increases) linear regression tends to overfit the data. However, ridge regression is used to restrict the overfitting in complex models. This does not seem important in the given example, where the dataset is only two-dimensional. However, higher-dimensional data often requires more complex models to be estimated and therefore often require the use of a regularizor. In Fig. 12 we can see how linear regression overfits the dataset for a high order polynomial, while ridge regression does not. Fig. 11 illustrates how a decreasing λ separates the weights and an increasing λ draws the weights towards zero, also known as *weight decay*.

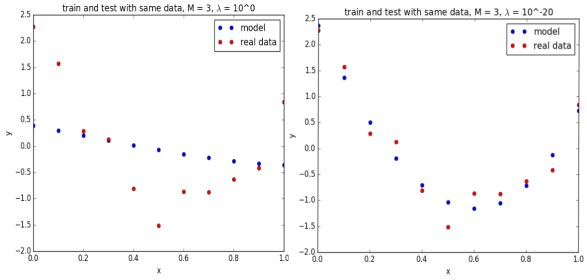

Fig. 10: The two plots show the effect of a decreasing regularization coefficient λ on the estimator. In the left image, the regularization coefficient $\lambda = 1$. If λ increases, it penalizes higher weights w stronger, which encourages them to drive to zero. This drives the model target vector y to converge to zero as well (see left). A smaller λ does penalize higher weights less, leaving them free to form the polynomial seen on the right. A $\lambda = 0$ would equal an estimator without regularization.

C. Training and testing on different data

In practice, the weights of an estimator are trained on a training dataset, the hyperparameters are determined by a validation data and finally the performance is evaluated on the test dataset. In the earlier section, we have used one dataset for all. Now, we are given three datasets: A, B and a validation set. Fig. 13 shows the learned models and their evaluation.

The sum of least squares error (SSE) is a valuable measurement index to evaluate the performance of a trained model. The SSE is given by (4). Varying along the hyperparameters we can find the one with the best fit on the validation data. Table III shows the influence of a varying regularization coefficient λ. For training with A, a λ around zero shows the best results, while a training with B has the best performance with $\lambda = 1$. Further evaluation of the model can be achieved by looking at the R squared error, which will produce similar results in a scale adapted the mean of the validation dataset.

TABLE II: Cosine Basis Function Weights

Cosine Basis Function Weights									
M	0	1	2	3	4	5	6	7	8
Actual	0	1	1	-	-	-	-	-	-
1	0.0	0.78	-	-	-	-	-	-	-
2	-0.11	0.78	1.19	-	-	-	-	-	-
4	-0.13	0.76	1.16	0.09	0.22	-	-	-	-
8	-0.15	0.77	1.1	0.1	0.16	-0.05	0.38	0.01	0.03

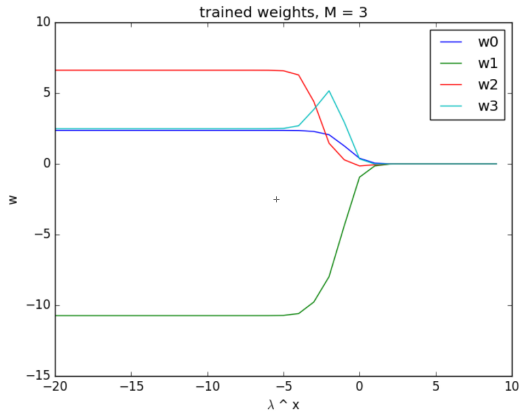

Fig. 11: [x-axis is $log(\lambda)$] If λ gets close to zero, the weight coefficients get separated (left). A λ of exactly zero would equal the case of not having a regularization term as higher weights w would not get penalized anymore. There is an intersection region around $\lambda = 10^{-3}$. After the intersection region λ becomes large and drives the weights to zero. There are $M + 1$ elements of w (polynomial basis with intercept).

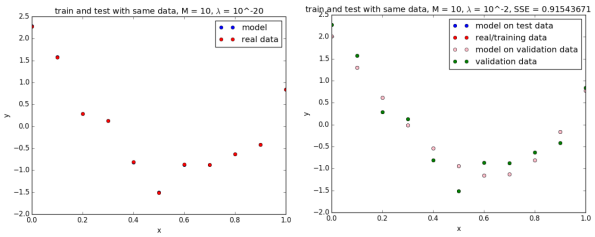

Fig. 12: This plot shows how complex models, represented by a the polynomial feature map ϕ of order $M = 10$ overfit the data with linear regression (left, where $\lambda \approx 0$). The quadratic regularizor penalizes the high weights and thereby achives to estimate the training data well (right, $SEE \approx 0.92$). Fig. 10 in comparison, shows a cubic polynomial feature map. Both, linear and ridge regression have performed comparibly well on low-order models.

$log(\lambda)$	SSE_A	SSE_B
-20	2.37	35.19
-1	2.55	34.80
0	4.35	32.10
1	16.17	32.18
2	29.54	62.68
10	76.51	76.51

TABLE III: Effect on λ on the sum of least squares error SSE on model trained with A and B and compared to validation dataset. A polynomial of $M = 3$ and $M = 1$ has been used for training with A and B, respectively.

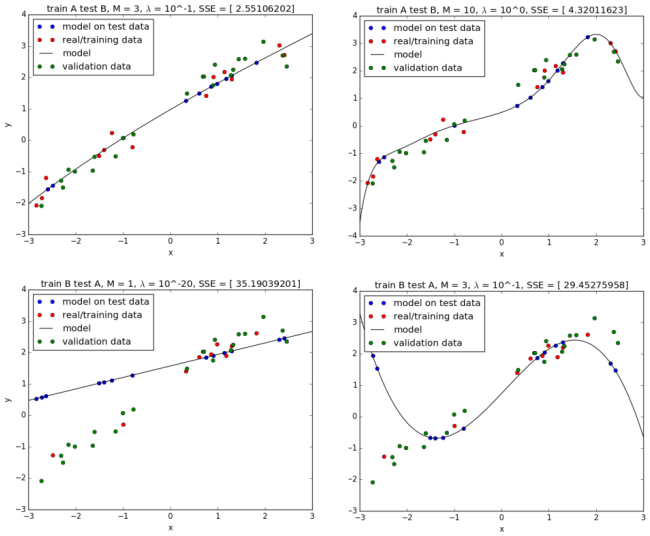

Fig. 13: The top two figures have trained a model with dataset A, applied on the test data B and the validation dataset. The model has been tested with lower (top-left) and higher (top-right) order polynomials. The higher order one is overfitted although there we have chosen a high value for λ. Other combinations of λ and M have been evaluated according to their SEE. The model of degree $M = 3$ and $\lambda = 0.1$ has shown the best performance with $SSE = 2.55$ and a visual graphical fit. The dataset B is characterized by one outlier value $(-2.6, 3.5)$ (s. bottom plots). In the bottom plots, the model was trained with dataset B and applied to dataset A and the validation data. We can see that the outlier value has significant influence on the learned model and skews the model. We have achieved best fit ($SSE \approx 29.45$) with $M = 3$ and $\lambda = 0.1$ (bottom-right). At the same time linear regression performed reasonably well with a linear curve (bottom-left). A solution to use B for training would be preprocessing the data to eliminate outlier values.

IV. Sparsity and Lasso

In ridge regression we have used a quadratic regularizor, which corresponds to the general regularized error function 6, where $q = 2$. Least absolute shrinkage and selection operator (Lasso) uses the L_1 norm, $q = 1$ to compute the error. Lasso gives a sparse solution model, i.e. by increasing the λ, it drives some weight coefficients to exactly zero. In this chapter, we compare the performance of Lasso to ridge regression.

A newly given dataset was generated by a sinoid feature vector and small noise ϵ. The feature vector is given by:

$$\phi(x) = (x, sin(0.4\pi * 1), ..., sin(0.4\pi x * 12)) \in \mathbb{R}^{13} \quad (8)$$

In Fig. 14, we can see that $\lambda = 0.1$ leads to a sparse estimation of the weight vector w and only four values remain

non-zero. Visualized in 2D, Lasso drives a weight tuple into the corners of a rectangle. Ridge regression drives the weights onto the circumference of a circle (Bishop 3.1.4).

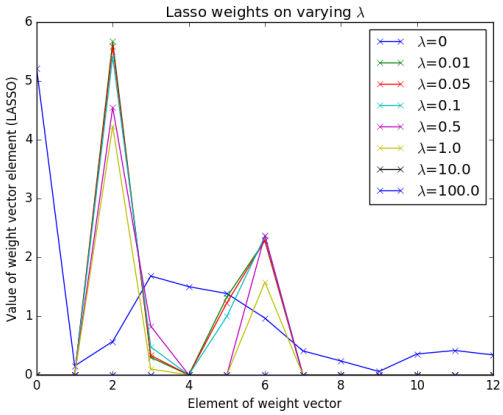

Fig. 14: This graphs shows the relation the lasso regularization coefficient λ to the weight vector w. $\lambda = 0$ equals an estimator without regularizing term. The further the λ increases, the further the weight elements are decaying to zero. $\lambda = 0.01$ leads to a reasonable separation of the weight vector, where four elements are non-zero: w_2, w_3, w_5 and w_6

We compare the weight coefficient from Lasso with the weights from ridge and the true w in Fig. 15 and visual inspection shows us that the estimated weights by lasso are very close to the true weights.

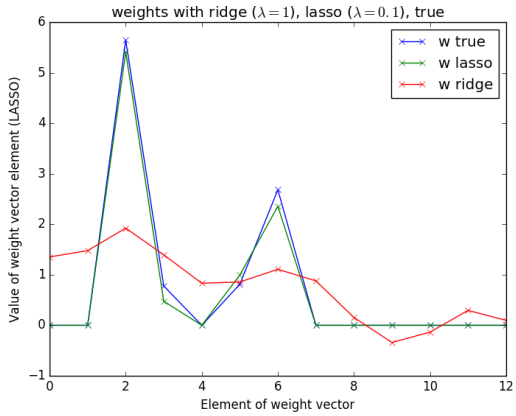

Fig. 15: This figure displays the weights from the lasso regularizor ($\lambda = 0.1$) with the estimated weights from ridge ($\lambda = 1$) and the true ones. On visual inspection, the lasso weights are very close to the true weights. All elements of the lasso weight vector have decayed to zero, such that only four non-zero elements remain. The weights from ridge regression differ strongly from the true weights and no weights are zero.

In Fig. 16, we use the learned weights of ridge, lasso and the true ones onto our validation dataset. This gives us knowledge about the fit of the estimator on the real data. Based on the smooth plot in Fig. 16 and the low SSE in Table IV, Lasso seems to generate the best model of this dataset.

	SSE_{val}	SSE_{test}	SSE_{train}
Lasso	1.13	1.39	0.19
Ridge	4.03	7.50	1.70
True	0.41	0.39	0.25

TABLE IV: This tables compares the estimators according to their sum of least squares error. We see that Lasso performs better than ridge on the validation and teh test dataset. I would not be useful to select an estimator by only looking on its performance on the trained data, as an overfitted model could achieve an error $SSE_{train} = 0$. In comparison to the true data, we see that Lasso performs reasonably well.

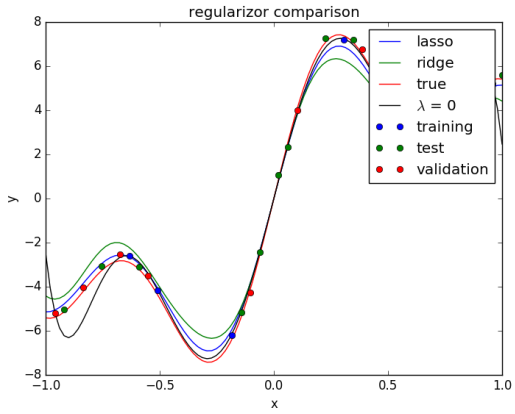

Fig. 16: In the plot, we have compared the performance the learned models from Lasso, ridge and linear regression ($\lambda = 0$) with the true sinoid function. The used weights are the ones displayed in Fig. 15. The plot also shows the used training, test and validation datasets. We can observe, that the model for the true weights and lasso weights is visually very close to the validation and test dataset. This observation gets confirmed by Table Table IV, in which we have analysed the SSE of the estimators. For this data we would select the Lasso estimator for evaluating a test dataset. The datapoints of the validation data do not fit onto the true sinoid curve, because they have been generated by adding small random noise.

V. Conclusion

We have given provided a look into gradient descent, linear basis function regression, ridge regression, sparsity and LASSO. Different datasets, weights and hyperparameters have been tested and compared regarding their performance.