Regularization and Feature Selection

In the previous chapter, we used computational graphs and gradient descent to build a linear regression model and then interpreted it through its weights. On the healthcare dataset, the most important attribute indeed received the largest weight, but the model also relied on many others. This raised two issues. First, some attributes were strongly correlated, which made interpretation harder. Second, all attributes remained in the model, since none of their weights was zero. It would be useful to construct a model that keeps only some attributes and ignores the rest. Such a model would be easier to use, easier to interpret, and could also tell us which attributes are not important for prediction.

We can implement the above idea of excluding certain input variables from the model using a procedure called *regularization*. But before introducing it, let's start with an example where additional attributes can strongly harm learning.

Polynomial Regression

We start with the data in Table 2. There is nothing particularly special about this data, except for its apparent unsuitability for linear regression, as also shown in the figure below.

x	y
0.19	0.28
0.09	0.24
0.16	0.48
0.34	0.60
0.59	0.71
0.87	0.51
0.89	0.37
0.68	0.62
0.36	0.51
0.82	0.48

Table 2: Training data.

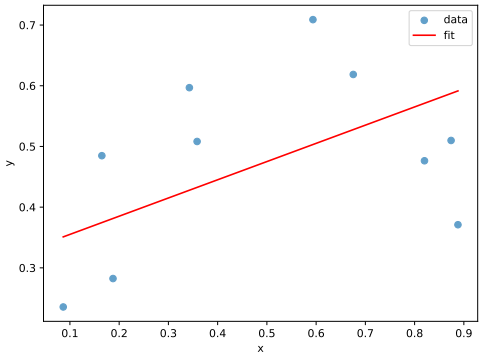

Figure 1: Linear regression does not model our initial data from Table 2 very well.

We could say that this data cannot be modeled with linear regression. Or can it? Let's add a new, derived attribute x^2 to the table and

use this extended table as input to linear regression. It will find the weights of the attributes for the model

$$y = w_0 + w_1 x + w_2 x^2.$$

We have thus added one more attribute to the linear model. As a function of x, the resulting model is quadratic rather than linear. But it is still linear in the parameters w_0, w_1, and w_2, so we can fit it with the same method as before.

We can again visualize the results of training such a model in the (x, y) plane, as shown in the figure below. Excellent! The model now fits the training data well. The value of the cost function that

x	x^2	y
0.19	0.04	0.28
0.09	0.01	0.24
0.16	0.03	0.48
0.34	0.12	0.60
0.59	0.35	0.71
0.87	0.76	0.51
0.89	0.79	0.37
0.68	0.46	0.62
0.36	0.13	0.51
0.82	0.67	0.48

Table 3: To the previous data table (Table 2) we added a new column, i.e., a new attribute computed from column x.

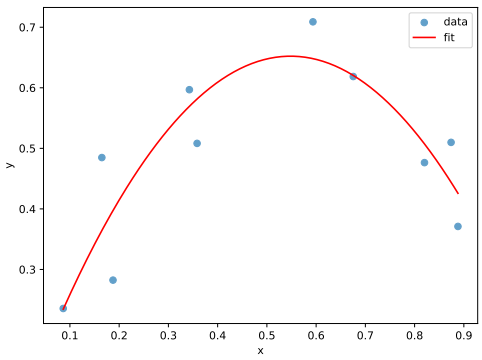

Figure 2: This is also linear regression.

we minimize in linear regression (mean squared error) is also much smaller than in the case where we did not add the new variable to the table. It decreased from 0.018 for the basic data to only 0.005.

Nothing really stops us from continuing to add more attributes. We could, for instance, add higher-order powers. Say x^3, x^4, all the way up to, say, x^7. We managed to reduce the cost function a bit further, this time to 0.003. Success, right!?

Such an extension of the training set is called *polynomial expansion*, and the trick of combining it with linear regression is called *polynomial regression*.

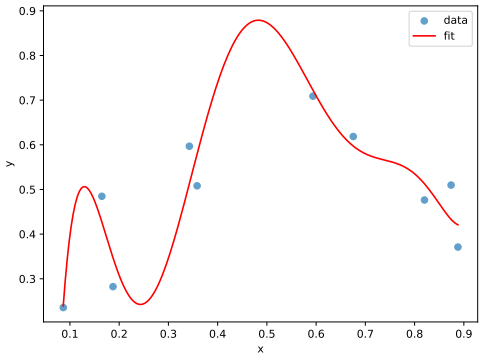

Figure 3: And this is also linear regression.

The desire to minimize the cost function has led us (almost) to the

What would happen if we added attributes up to a ninth-degree polynomial, i.e., including x^9? What would the value of the objective function be then? Why?

extreme. The model started to completely adapt to the training set and as such is no longer useful for prediction.

L2 Regularization

If we take a closer look at the model weights in the example above, we notice an interesting phenomenon. In the second-degree model, the weights are still relatively small. However, when we start adding new attributes, i.e., powers x^3, x^4, x^5, and so on, the weights quickly become very large. Some are positive, others negative, and their absolute values can become thousands or even tens of thousands of times larger than the values of the input variables.

This also explains why the fitted function becomes more complicated. Large weights make the prediction very sensitive to small changes in x. The model then starts to follow minor fluctuations in the training data, including noise. As a result, it can fit the training set very well while performing poorly on new examples.

Observing the values of the weights when adding attributes, as we did above, leads us to the following idea. In addition to wanting a good fit to the training data, when building a linear model we may also want the weights to be as small as possible, making the model simpler. We therefore need to include, in the cost function, not only the model error but also a penalty for large weights. Since we do not care whether a weight is positive or negative, we square the weights. We usually omit the weight w_0, as it only represents the intercept of the function. The cost function of linear regression thus becomes

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} w_j^2.$$

The first term of the cost function is the same as before, i.e., the mean squared prediction error. The second term penalizes large weights. Since this is a sum of two terms that are not in the same units and are conceptually different, we need to properly weight the second term. We do this using the parameter λ, which we call the *regularization strength*. In the implementation, we therefore only modify the computation of the loss function inside the `LinReg` class that implements linear regression:

```
def loss(self, xs, ys):
    yhats = [self(x) for x in xs]
    loss = sum([(y - yhat)**2  for y, yhat in zip(ys, yhats)]) \
           / Value(len(xs))
    loss += self.reg_strength * sum([w**2 for w in self.weights]) \
            / len(self.weights)
    return loss
```

The `loss` function assumes that we have properly initialized the variable `self.reg_strength` in the class. We additionally divide the regularization term by the number of model weights, noting that `self.b` is not included.

When the regularization strength $\lambda = 0$ (that is, when `self.reg_strength` equals 0.0), we obtain exactly the same model as in standard linear regression. As we increase the value of λ, the penalty for large weights becomes increasingly important. The model therefore prefers smaller weights, and as a result the function $y(x)$ becomes smoother and simpler. What happens if we increase the regularization strength to a very large value? In this case, the second term in the cost function dominates, and the model tends toward making all weights as close to zero as possible.

What will the value of w_0 be in this case? The cost function can then be written as

$$J(w_0) = \frac{1}{n}\sum_{i=1}^{n}(y_i - w_0)^2.$$

Let's find the minimum of this function. We differentiate with respect to w_0:

$$\frac{\partial J}{\partial w_0} = \frac{1}{n}\sum_{i=1}^{n} 2(w_0 - y_i).$$

At the minimum, the derivative must be zero:

$$\sum_{i=1}^{n}(w_0 - y_i) = 0.$$

From this it follows that

$$nw_0 = \sum_{i=1}^{n} y_i,$$

or

$$w_0 = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

With very strong regularization, the model therefore predicts only the mean of the training targets y. In other words, the simplest possible regression model is the one that ignores the input attributes and always predicts the average target value.

The procedure we just described is called *regularization*. Since we square the weights, we more specifically refer to it as *L2 regularization*. This approach is also commonly known in the literature as ridge regression.

The name comes from the shape of the optimization problem: the added term $(\lambda \sum_j w_j^2)$ changes the surface of the cost function so that it forms a pronounced ridge along directions where parameters are poorly determined due to strong correlations between attributes. The regularization term "rounds" this ridge, giving the problem a unique solution and keeping the weights bounded.

Evaluating Model Accuracy with R^2

Let's spend a bit more time on predicting with the mean value on the training set. That is, on the simplest regression model, which we

build without taking attribute values into account. We expect that models that do take these values into account will be more accurate, i.e., that their error will be smaller. It therefore makes sense to evaluate the ratio between the error obtained by an attribute-informed model and our baseline model, where we predict using the mean value:

$$\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2},$$

where $\hat{y}_i$ is the prediction of our model and $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ is the mean value of the target variable. If the value of this ratio is close to 0, it means that our model is much better than predicting with the mean. If it is equal to 1, the model is as good as the baseline model, and if it is greater than 1, it is even worse.

Since we want a measure where higher values indicate a better model (ideally close to 1) and worse values are closer to 0, we subtract this ratio from 1 and obtain the R^2 measure:

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

The R^2 measure is often convenient because it is dimensionless and compares the model directly with the baseline that always predicts the mean. RMSE, by contrast, is expressed in the units of the target variable. For this reason, R^2 is often easier to compare across different problems. It can also be interpreted as the proportion of variance explained by the model, although this interpretation should be used with some care. An R^2 value close to 1 indicates a good fit, a value near 0 means that the model performs similarly to the baseline, and negative values indicate that it performs even worse.

L1 Regularization

With L2 regularization, we penalized large weights by adding the sum of their squares to the cost function. As a result, the weights became smaller, but typically none of them became exactly zero. The model therefore still used all attributes, just with somewhat smaller weights. However, if our goal is for the model to completely eliminate some attributes, we need a slightly different approach. Instead of squaring the weights, we can include the sum of their absolute values in the cost function. We obtain the cost function

$$J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} |w_j|.$$

What happens when we increase the regularization strength λ? Similar to before, the model starts to prefer smaller weights. But this time something else interesting happens: some weights become exactly zero. Such an attribute is effectively no longer used by the model. We could say that it has been removed from the model.

L1 regularization therefore has an additional useful property: it can perform *feature selection*. Some weights become exactly zero, which means that the corresponding input variables are removed from the model.

If we keep increasing the regularization strength, there will be fewer and fewer non-zero weights. For very large values of λ, only the weight w_0 will remain, and the model will again become a constant that predicts the mean value of the training data.

The procedure we just described is called *L1 regularization*. In the statistical literature, it is also known as *LASSO* (short for *Least Absolute Shrinkage and Selection Operator*). Its key feature is precisely that, in addition to regularization, it also enables automatic feature selection, which is often very useful in practice. In principle, implementing this regularization again only changes the computation of the objective function. Below is a function that implements both types of regularization discussed so far:

```
def loss(self, xs, ys):
    yhats = [self(x) for x in xs]

    loss = sum([(y - yhat)**2  for y, yhat in zip(ys, yhats)]) \
           / Value(len(xs))
    if self.reg == 'l1':
        loss += self.reg_strength * sum([abs(w) for w in self.weights]) \
                / len(self.weights)
    elif self.reg == 'l2':
        loss += self.reg_strength * sum([w**2 for w in self.weights]) \
                / len(self.weights)
    return loss
```

In many cases, this is enough. If we want to encourage exact zeros even more strongly, we can add an extra step after each gradient update and set very small weights to zero. In practice, this means choosing a threshold and replacing all weights below it in absolute value by zero. This kind of thresholding strengthens the tendency of L1 regularization to remove unimportant attributes.

```
if model.reg == 'l1':
    shrink = learning_rate * model.reg_strength
    for p in model.parameters():
        if abs(p.data) < shrink:
            p.data = 0
```

Graphical Comparison of Regularizations

We can also understand regularization in a slightly different way. Instead of adding the regularization term to the cost function, we can formulate the problem as a constrained optimization:

$$\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \qquad \text{subject to} \qquad R(\mathbf{w}) \leq s.$$

Here, the function $R(\mathbf{w})$ determines the size of the weights, and the parameter s limits how large these weights are allowed to be.

If we only have two weights, w_1 and w_2, we can visualize this problem in the (w_1, w_2) plane. The error contours (without regularization) are ellipses. The solution is found at the point where the lowest ellipse first touches the feasible set of weights.

For L2 regularization, the constraint is

$$w_1^2 + w_2^2 \leq s,$$

which represents a circle in the plane. The point of contact with the ellipse therefore usually occurs somewhere along the smooth boundary of the circle, so the weights are smaller, but typically none of them is exactly zero.

For L1 regularization, the constraint becomes

$$|w_1| + |w_2| \leq s,$$

which forms a diamond shape in the (w_1, w_2) plane. Since the diamond has sharp corners along the coordinate axes, the ellipses often touch it exactly at one of these corners. At such a point, one of the weights is exactly zero.

Regularization and Accuracy on Training and Test Sets

So far, we judged the model only by how well it fit the training data. Regularization changes this, because it introduces a trade-off between fit and simplicity. As the regularization strength increases, training accuracy usually decreases. The more important question is what happens on the test set.

Let's observe these relationships through an experiment. We will take the body fat dataset that we already used in the previous chapter and now split it into training and test data. To make the effect of regularization more pronounced, we will sample a very small train-

Why would such a data split make the effect of regularization more visible?

ing set and leave all the remaining data for testing:

```
df = pd.read_csv("body-fat-brozek.csv")
feature_names = df.columns[1:].tolist()
ys = df.iloc[:, 0].tolist()
X = df.iloc[:, 1:].values.tolist()

X_arr = np.array(X)
mean = X_arr.mean(axis=0)
std = X_arr.std(axis=0)
X_norm = ((X_arr - mean) / std).tolist()

X_train, X_test, y_train, y_test =
   train_test_split(X_norm, ys, test_size=0.95, random_state=42)
```

Before training, we also standardized the data. This puts all features on a comparable scale and usually makes gradient descent more stable and faster. We now train the model with several regularization strengths and report accuracy on both the training and test sets.

```
reg_strengths = [0.001, 0.05, 0.01, 0.1, 0.5, 1, 5, 10, 20, 30]
for reg_strength in reg_strengths:
    lr = LinReg(n_inputs=len(feature_names), reg='l1',
                reg_strength=reg_strength)
    model = train(lr, X_train, y_train, n_epochs=1000,
                  batch_size=None, learning_rate=0.01)
    # print("Weights (most to least important):")
    # print_weights(model, feature_names)

    # evaluate the model on the training and testing sets
    y_pred_train = [model(xi).data for xi in X_train]
    r2_train = r2_score(y_train, y_pred_train)

    y_pred_test = [model(xi).data for xi in X_test]
    r2_test = r2_score(y_test, y_pred_test)

    non_zero_weights = len([w for w in model.weights if w.data != 0])
    print(f"lambda: {reg_strength:5.2f}, R2: {r2_train:.3f} & " \
          "{r2_test:.3f}, non-zero weights: {non_zero_weights}")
```

The results show a monotonic decrease in accuracy on the training set as the regularization strength increases:

```
lambda:  0.00, R2: 0.925 & 0.493, non-zero weights: 14
lambda:  0.05, R2: 0.927 & 0.490, non-zero weights: 13
lambda:  0.01, R2: 0.933 & 0.398, non-zero weights: 14
lambda:  0.10, R2: 0.929 & 0.472, non-zero weights: 14
lambda:  0.50, R2: 0.898 & 0.518, non-zero weights: 7
lambda:  1.00, R2: 0.889 & 0.540, non-zero weights: 8
```

```
lambda:  5.00, R2: 0.756 & 0.484, non-zero weights: 2
lambda: 10.00, R2: 0.725 & 0.592, non-zero weights: 3
lambda: 20.00, R2: 0.564 & 0.453, non-zero weights: 1
lambda: 30.00, R2: 0.496 & 0.414, non-zero weights: 1
```

However, the behavior on the test set is quite different, where the model is most accurate at regularization strength $\lambda = 10$. We would observe similar behavior with L2 regularization as well, but we leave that and further experiments to the reader. The question that arises, however, is how to find the regularization strength that leads to a model best suited for new data.

How We Find the Right Regularization Strength

Basically, it is hard, and we have to be very careful, especially because we want to report both the "right" regularization strength and an estimate of the accuracy of the model obtained in this way. Here we will assume that we search for the regularization strength by choosing it from some list and checking when our model achieves the best result on a separate set. Just as we did in the previous section. There, we got the best model at $\lambda = 10$, and its accuracy on the training set was $R^2 = 0.592$. So do we choose that model and predict that its accuracy on new examples will probably be $R^2 = 0.592$?

No. This estimate is biased, because we selected the regularization strength precisely based on the test set, which was therefore "used up" for learning. Such an accuracy estimate is therefore too optimistic and does not reflect the actual performance of the model on new data.

At this point, learning no longer consists only of fitting a linear model. It also includes choosing the regularization strength. We therefore have to evaluate the entire procedure on an independent test set.

So, to avoid bias in the accuracy estimate, we need to split the data into an additional test set. We will therefore divide the data into three disjoint sets:

1. **training set** (say 70% of the data), on which we train the model for a given regularization strength,

2. **validation set** (e.g., 15%), on which we observe the accuracy of the model for a given regularization strength and which allows us to select the appropriate strength, that is, the one for which the regularized model is most accurate on the validation set,

3. **test set** (e.g., 15%), on which we finally estimate the accuracy of the model.

Which model should we report? Not simply the one that happened to perform best on the validation set during the first split, because that model was trained on only part of the available data. Instead, after reserving the test set for the final evaluation, we repeat the model-selection procedure on the remaining data and then fit the selected model again.

We still need a training-validation split in the last step because model construction now includes hyperparameter selection. In other words, the final method is not just linear regression with fixed settings, but a complete procedure that also chooses the regularization strength.

Finally, let us describe the procedure proposed above in pseudocode:

```
# accuracy estimation
split data to train, validation, test
for reg in lambdas:
    model = fit(train, reg)
    score[reg] = evaluate(model, validation)
choose best reg
score = evaluate(fit(train, best_reg), test)

# derivation of final model
split data to train, validation
for reg in lambdas:
    model = fit(train, reg)
    score[reg] = evaluate(model, validation)
choose best reg
model = fit(train, best_reg)
```

Already quite complicated. But there is another problem here. The above is appropriate for really large datasets, but for smaller ones, the split into subsets itself can strongly affect the accuracy estimate. For that, we could use cross-validation and report an estimate as the average over several experiments. Would you know how to modify the above appropriately and include cross-validation?