Logistic regression Hyperparameter

How to increase the Accuracy Score?

Elham Mahdian

Machine learning Exercise

Sep 2022

Types of Logistic Regression

Types of Logistic Regression:

- Binary Logistic Regression: The target variable has only two possible outcomes such as Spam or Not Spam, Cancer or No Cancer.
- Multinomial Logistic Regression: The target variable has three or more nominal categories such as predicting the type of Wine.
- Ordinal Logistic Regression: the target variable has three or more ordinal categories such as restaurant or product rating from 1 to 5.

Hyperparameter Tuning

Hyperparameter tuning is an optimization technique and is an essential aspect of the machine learning process. A good choice of hyperparameters may make your model meet your desired metric. Yet, the plethora of hyperparameters, algorithms, and optimization objectives can lead to an unending cycle of continuous optimization effort.

Logistic Regression Hyperparameters

The main hyperparameters we may tune in logistic regression are: solver, penalty, and regularization strength (sklearn documentation).

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=0, solver='liblinear', tol=0.0001, verbose=0,
                   warm_start=False)
```

max_iter

max_iter is an integer (100 by default) that defines the maximum number of iterations by the solver during model fitting.

multi_class

multi_class is a string ('ovr' by default) that decides the approach to use for handling multiple classes. Other options are 'multinomial' and 'auto'.

Verbose

Verbose is a non-negative integer (0 by default) that defines the verbosity for the 'liblinear' and 'lbfgs' solvers.

warm_start

warm_start is a Boolean (False by default) that decides whether to reuse the previously obtained solution.

n_jobs

n_jobs is an integer or None (default) that defines the number of parallel processes to use. None usually means to use one core, while -1 means to use all available cores.

l1_ratio

l1_ratio is either a floating-point number between zero and one or None (default). It defines the relative importance of the L1 part in the elastic-net regularization.

Dual

Dual is a Boolean (False by default) that decides whether to use primal (when False) or dual formulation (when True).

Tol

Tol is a floating-point number (0.0001 by default) that defines the tolerance for stopping the procedure.

fit_intercept

fit_intercept is a Boolean (True by default) that decides whether to calculate the intercept b_0 (when True) or consider it equal to zero (when False).

intercept_scaling intercept_scaling float, default=1

Useful only when the solver 'liblinear' is used and self.fit_intercept is set to True. In this case, x becomes [x, self.intercept_scaling], i.e. a "synthetic" feature with constant value equal to intercept_scaling is appended to the instance vector. The intercept becomes intercept_scaling * synthetic_feature_weight.

Note! the synthetic feature weight is subject to l1/l2 regularization as all other features. To lessen the effect of regularization on synthetic feature weight (and therefore on the intercept) intercept_scaling has to be increased.

class_weight

class_weight is a dictionary, 'balanced', or None (default) that defines the weights related to each class. When None, all classes have the weight one.

random_state

random_state is an integer, an instance of numpy.RandomState, or None (default) that defines what pseudo-random number generator to use.

Definition of some of the hyperparameters in detail

- **class_weight**

What is Class Imbalance?

Class imbalance is a problem that occurs in machine learning classification problems. It merely tells that the target class's frequency is highly imbalanced, i.e., the occurrence of one of the classes is very high compared to the other classes present. In other words, there is a bias or skewness towards the majority class present in the target. Suppose we consider a binary classification where the majority target class has 10000 rows, and the minority target class has only 100 rows. In that

case, the ratio is 100:1, i.e., for every 100 majority class, there is only one minority class present. This problem is what we refer to as class imbalance. Some of the general areas where we can find such data are fraud detection, churn prediction, medical diagnosis, e-mail classification, etc.

Note: *To check the performance of the model, we will be using the f1 score as the metric, not accuracy.*

The reason is if we create a dumb model that predicts every new training data as 0 (no Diabetes) even then we will get very high accuracy because the model is biased towards the majority class. Here, the model is heavily accurate but not at all serving any value to our problem statement. That is why we will be using f1 score as the evaluation metric. F1 score is nothing but the harmonic mean of precision and recall. However, the evaluation metric is chosen based on the business problem and what type of error we want to reduce. But, the f1 score is the go-to metric when it comes to class imbalance problems.

By default, the value of class_weight=None, i.e. both the classes have been given equal weights. Other than that, we can either give it as 'balanced' or we can pass a dictionary that contains manual weights for both the classes.

When the class_weights = 'balanced', the model automatically assigns the class weights inversely proportional to their respective frequencies.

To be more precise**,** the formula to calculate this is:

wj=n_samples / (n_classes * n_samplesj)

Here,

- wj is the weight for each class(j signifies the class)
- n_samplesis the total number of samples or rows in the dataset
- n_classesis the total number of unique classes in the target
- n_samplesjis the total number of rows of the respective class

- **C (or regularization strength)**

C is known as a "hyperparameter." The parameters are numbers that tell the model what to do with the characteristics, whereas the hyperparameters instruct the model on how to choose parameters. Regularization will penalize the extreme parameters, the extreme values in the training data leads to overfitting.
A high value of C tells the model to give more weight to the training data. A lower value of C will indicate the model to give complexity more weight at the cost of fitting the data. Thus, a high Hyper Parameter value C indicates that training data is more important and reflects the real world data, whereas low value is just the opposite of this.

- **Penalty**

Penalty is a string ('l2' by default) that decides whether there is regularization and which approach to use. Other options are 'l1', 'elasticnet', and 'none'. Regularization techniques play a vital role in the development of machine learning models. Especially complex models, like neural networks, prone to overfitting the training data. Broken down, the word “regularize” states that we’re making something regular. In a mathematical or ML context, we make something regular by adding information which creates a solution that prevents overfitting. The “something” we’re making regular in our ML context is the “objective function”, something we try to minimize during the optimization problem.

To put it simply, in regularization, information is added to an objective function. We use regularization because we want to add some bias into our model to prevent it overfitting to our training data. After adding a regularization, we end up with a machine learning model that performs well on the training data, and has a good ability to generalize to new examples that it has not seen during training.

L1 regularization

L1 regularization, also known as L1 norm or Lasso (in regression problems), combats overfitting by shrinking the parameters towards 0. This makes some features obsolete. It’s a form of feature selection, because when we assign a feature with a 0 weight, we’re multiplying the feature values by 0 which returns 0, eradicating the significance of that feature. If the input features of our model

have weights closer to 0, our L1 norm would be sparse. A selection of the input features would have weights equal to zero, and the rest would be non-zero.

For example, imagine we want to predict housing prices using machine learning. Consider the following features:

- **Street** – road access,
- **Neighborhood** – property location,
- **Accessibility** – transport access,
- **Year Built** – year the house was built in,
- **Rooms** – number of rooms,
- **Kitchens** – number of kitchens,
- **Fireplaces** – number of fireplaces in the house.

When predicting the value of a house, intuition tells us that different input features won't have the same influence on the price. For example, it's highly likely that the neighborhood or the number of rooms have a higher influence on the price of the property than the number of fireplaces.

So, our L1 regularization technique would assign the fireplaces feature with a zero weight, because it doesn't have a significant effect on the price. We can expect the neighborhood and the number rooms to be assigned non-zero weights, because these features influence the price of a property significantly.

Mathematically, we express L1 regularization by extending our loss function like such:

$$LossFunction = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y} - Y)^2 + \lambda \sum_{i=1}^{N} \mid \theta_i \mid$$

Essentially, when we use L1 regularization, we are penalizing the absolute value of the weights.

In real world environments, we often have features that are highly correlated. For example, the year our home was built and the number of rooms in the home may have a high correlation. Something to consider when using L1 regularization is that when we have highly correlated

features, the L1 norm would select only 1 of the features from the group of correlated features in an arbitrary nature, which is something that we might not want.

Nonetheless, for our example regression problem, Lasso regression (Linear Regression with L1 regularization) would produce a model that is highly interpretable, and only uses a subset of input features, thus reducing the complexity of the model.

L2 regularization

L2 regularization, or the L2 norm, or Ridge (in regression problems), combats overfitting by forcing weights to be small, but not making them exactly 0. So, if we're predicting house prices again, this means the less significant features for predicting the house price would still have some influence over the final prediction, but it would only be a small influence. The regularization term that we add to the loss function when performing L2 regularization is the sum of squares of all of the feature weights:

$$LossFunction = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y} - Y)^2 + \lambda\sum_{i=1}^{N}\theta_i^2$$

So, L2 regularization returns a non-sparse solution since the weights will be non-zero (although some may be close to 0).

A major snag to consider when using L2 regularization is that it's not robust to outliers. The squared terms will blow up the differences in the error of the outliers. The regularization would then attempt to fix this by penalizing the weights.

The differences between L1 and L2 regularization:

- L1 regularization penalizes the sum of absolute values of the weights, whereas L2 regularization penalizes the sum of squares of the weights.
- The L1 regularization solution is sparse. The L2 regularization solution is non-sparse.
- L2 regularization doesn't perform feature selection, since weights are only reduced to values near 0 instead of 0. L1 regularization has built-in feature selection.
- L1 regularization is robust to outliers, L2 regularization is not.

Which is better – L1 or L2 regularization?

Whether one regularization method is better than the other is a question for academics to debate. However, as a practitioner, there are some important factors to consider when you need to choose between L1 and L2 regularization. I've divided them into 6 categories, and will show you which solution is better for each category.

Which solution is more robust? L1

According to the definition provided by Investopedia, a model is considered robust if its output and forecast are consistently accurate, even if one or more of the input variables or assumptions are drastically changed due to unforeseen circumstances. [**Source**: Investopedia]

L1 regularization is more robust than L2 regularization for a fairly obvious reason. L2 regularization takes the square of the weights, so the cost of outliers present in the data increases exponentially. L1 regularization takes the absolute values of the weights, so the cost only increases linearly.

What solution has more possibilities? L1

By this I mean the number of solutions to arrive at one point. L1 regularization uses Manhattan distances to arrive at a single point, so there are many routes that can be taken to arrive at a point. L2 regularization uses Euclidean distances, which will tell you the fastest way to get to a point. This means the L2 norm only has 1 possible solution.

Which solution is less Computationally expensive? L2

Since L2 regularization takes the square of the weights, it's classed as a closed solution. L1 involves taking the absolute values of the weights, meaning that the solution is a non-differentiable piecewise function or, put simply, it has no closed form solution. L1 regularization is computationally more expensive, because it cannot be solved in terms of matrix math.

Which solution creates a sparse output? L1

By sparsity, we mean that the solution produced by the regularizer has many values that are zero. However, we know they're 0, unlike missing data where we don't know what some or many of the values actually are. As previously stated, L2 regularization only shrinks the weights to values close to 0, rather than actually being 0. On the other hand, L1 regularization shrinks the values to 0. This in effect is a form of feature selection, because certain features are taken from the model entirely. With that being said, feature selection could be an additional step before the model you decide to go ahead with is fit, but with L1 regularization you can skip this step, as it's built into the technique.

- **Solver**

Solver is the algorithm to use in the optimization problem. The choices are *{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}*, default='lbfgs'. The solver uses a Coordinate Descent (CD) algorithm that solves optimization problems by successively performing approximate minimization along coordinate directions or coordinate hyperplanes.

1. lbfgs relatively performs well compared to other methods and it saves a lot of memory, however, sometimes it may have issues with convergence.
2. sag faster than other solvers for large datasets, when both the number of samples and the number of features are large.
3. saga the solver of choice for sparse multinomial logistic regression and it's also suitable for very large datasets.
4. newton-cg computationally expensive because of the Hessian Matrix.
5. liblinearrecommended when you have a high dimension dataset - solving large-scale classification problems.

Note, 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale, therefore we preprocess the data with a scaler from sklearn.preprocessing.

Comparison between the methods

1. Newton's Method(*newton-cg***)**

Recall the motivation for gradient descent step at x: we minimize the quadratic function (i.e. Cost Function).

Newton's method uses in a sense a ***better*** quadratic function minimisation. A better because it uses the quadratic approximation (i.e. first AND *second* partial derivatives).

You can imagine it as a twisted Gradient Descent with the Hessian (*the Hessian is a square matrix of second-order partial derivatives of order n X n*).

Moreover, the geometric interpretation of Newton's method is that at each iteration one approximates f(x) by a quadratic function around xn, and then takes a step towards the maximum/minimum of that quadratic function (in higher dimensions, this may also be a *saddle point*). Note that if f(x) happens to be a quadratic function, then the exact extremum is found in one step.

Drawbacks:

1. It's computationally ***expensive*** because of the Hessian Matrix (i.e. second partial derivatives calculations).
2. It attracts to ***Saddle Points*** which are common in multivariable optimization (i.e. a point that its partial derivatives disagree over whether this input should be a maximum or a minimum point!).

2. Limited-memory Broyden–Fletcher–Goldfarb–Shanno Algorithm(*lbfgs***):**

In a nutshell, it is analogue of the Newton's Method, yet here the Hessian matrix is ***approximated*** using updates specified by gradient evaluations (or approximate gradient evaluations). In other words, using an estimation to the inverse Hessian matrix.

The term Limited-memory simply means it stores only a few vectors that represent the approximation implicitly.

If I dare say that when dataset is ***small***, L-BFGS relatively performs the best compared to other methods especially it saves a lot of memory, however there are some "*serious*" drawbacks such that if it is unsafeguarded, it may not converge to anything.

Side note: This solver has become the default solver in sklearn LogisticRegression since version 0.22, replacing LIBLINEAR.

3. **A Library for Large Linear Classification**(liblinear)

LIBLINEAR is the winner of ICML 2008 large-scale learning challenge. It applies *automatic parameter selection* (a.k.a L1 Regularization) and it's recommended when you have high dimension dataset (*recommended for solving large-scale classification problems*)

Drawbacks:

1. It may get stuck at a *non-stationary point* (i.e. non-optima) if the level curves of a function are not smooth.
2. Also cannot run in parallel.
3. It cannot learn a true multinomial (multiclass) model; instead, the optimization problem is decomposed in a "one-vs-rest" fashion, so separate binary classifiers are trained for all classes.

4. Stochastic Average Gradient(Sag):

SAG method optimizes the sum of a finite number of smooth convex functions. Like stochastic gradient (SG) methods, the SAG method's iteration cost is independent of the number of terms in the sum. However, by *incorporating a memory of previous gradient values the SAG method achieves a faster convergence rate* than black-box SG methods.

It is **faster** than other solvers for *large* datasets, when both the number of samples and the number of features are large.

Drawbacks:

1. It only supports L2 penalization.
2. Its memory cost of O(N), which can make it impractical for large N (*because it remembers the most recently computed values for approximately all gradients*).

5. SAGA:

The SAGA solver is a *variant* of SAG that also supports the non-smooth *penalty L1* option (i.e. L1 Regularization). This is therefore the solver of choice for **sparse** multinomial logistic regression and it's also suitable for *very Large* dataset.

Side note: According to Scikit Documentation: The SAGA solver is often the best choice.

You should carefully match the solver and regularization method for several reasons:

- 'liblinear' solver doesn't work without regularization.
- 'newton-cg', 'sag', 'saga', and 'lbfgs' don't support L1 regularization.
- 'saga' is the only solver that supports elastic-net regularization.

At the end, we can tune hyperparameters in logistic regression by using GridSearchCV. This gives us opportunity to compare different parameters and then decide which one is more appropriate for our model.

```
logModel = LogisticRegression()
```

```
param_grid = [
    {'penalty' : ['l1', 'l2', 'elasticnet', 'none'],
    'C' : np.logspace(-4, 4, 20),
    'solver' : ['lbfgs','newton-cg','liblinear','sag','saga'],
    'max_iter' : [100, 1000,2500, 5000]
    }
]
```

```
from sklearn.model_selection import GridSearchCV
```

```
clf = GridSearchCV(logModel, param_grid = param_grid, cv = 3, verbose=True, n_jobs=-1)
```

```
best_clf = clf.fit(X,y)
```

```
best_clf.best_estimator_
```

```
print (f'Accuracy - : {best_clf.score(X,y):.3f}')
```