stochastic-gradient-descent

January 24, 2024

0.1 Stochastic Gradient Descent - Tutorial (Neuraldemy by Amritesh Kumar)

This tutorial is part of Neuraldemy and only accessible to people who have purchased it. Visit neuraldemy.com for more.

Since SGD is an optimization method, we can use to solve different loss functions or to train different models. It works both for classification and regression. In this tutorial,we will limit ourself to the Sklearn implementation only and in future tutorials we will see how we can use SGD and its variants in deep learning problems.

`SGDClassifier`

This estimator implements regularized linear models with stochastic gradient descent (SGD) learning: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate). SGD allows minibatch (online/out-of-core) learning via the partial_fit method. For best results using the default learning rate schedule, the data should have zero mean and unit variance.

This implementation works with data represented as dense or sparse arrays of floating point values for the features. The model it fits can be controlled with the loss parameter; by default, it fits a linear support vector machine (SVM).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

loss The loss function to be used.

- `hinge`: gives a linear SVM.
- `log_loss`: gives logistic regression, a probabilistic classifier.
- `modified_huber`: another smooth loss that brings tolerance to outliers as well as probability estimates.
- `squared_hinge`: like hinge but is quadratically penalized.
- `perceptron`: the linear loss used by the perceptron algorithm.
- `squared_error`, `huber`, `epsilon_insensitive`, `squared_epsilon_insensitive`: designed for regression but can be useful in classification as well; see SGDRegressor for a description.

More details about the losses formulas can be found in the User Guide.

penalty The penalty (aka regularization term) to be used. Defaults to `l2` which is the standard regularizer for linear SVM models. `l1` and `elasticnet` might bring sparsity to the model (feature selection) not achievable with `l2`. No penalty is added when set to `None`.

alpha Constant that multiplies the regularization term. The higher the value, the stronger the regularization. Also used to compute the learning rate when `learning_rate` is set to `optimal`. Values must be in the range [0.0, inf).

l1_ratio The Elastic Net mixing parameter, with 0 <= `l1_ratio` <= 1. `l1_ratio=0` corresponds to L2 penalty, `l1_ratio=1` to L1. Only used if penalty is `elasticnet`. Values must be in the range [0.0, 1.0].

fit_intercept Whether the intercept should be estimated or not. If `False`, the data is assumed to be already centered.

max_iter The maximum number of passes over the training data (aka epochs). It only impacts the behavior in the `fit` method, and not the `partial_fit` method. Values must be in the range [1, inf).

New in version 0.19.

tol The stopping criterion. If it is not `None`, training will stop when `(loss > best_loss - tol)` for `n_iter_no_change` consecutive epochs. Convergence is checked against the training loss or the validation loss depending on the `early_stopping` parameter. Values must be in the range [0.0, inf).

New in version 0.19.

shuffle Whether or not the training data should be shuffled after each epoch.

verbose The verbosity level. Values must be in the range [0, inf).

epsilon Epsilon in the epsilon-insensitive loss functions; only if loss is `huber`, `epsilon_insensitive`, or `squared_epsilon_insensitive`. For `huber`, determines the threshold at which it becomes less important to get the prediction exactly right. For epsilon-insensitive, any differences between the current prediction and the correct label are ignored if they are less than this threshold. Values must be in the range [0.0, inf).

n_jobs The number of CPUs to use to do the OVA (One Versus All, for multi-class problems) computation. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details.

random_state Used for shuffling the data, when `shuffle` is set to `True`. Pass an int for reproducible output across multiple function calls. See Glossary. Integer values must be in the range [0, 2**32 - 1].

learning_rate The learning rate schedule determines how the step size in the optimization process is adjusted over time. There are four options:

- `constant`: The learning rate (`eta`) remains constant and is set to `eta0`.
- `optimal`: The learning rate is dynamically adjusted based on a heuristic proposed by Leon Bottou.
- `invscaling`: The learning rate follows an inverse scaling schedule, where `eta = eta0 / pow(t, power_t)`.
- `adaptive`: The learning rate remains constant as long as the training loss is decreasing. If the loss fails to decrease for `n_iter_no_change` consecutive epochs, or if the validation score

does not increase by `tol` (if `early_stopping` is True), the current learning rate is divided by 5.

New in version 0.20: Added `adaptive` option.

eta0 The initial learning rate for the `constant`, `invscaling`, or `adaptive` schedules. The default value is 0.0 as `eta0` is not used by the default `optimal` schedule. Values must be in the range (0.0, inf).

power_t The exponent for the inverse scaling learning rate, with a default value of 0.5. It affects the rate at which the learning rate decreases in the `invscaling` schedule. Values must be in the range (-inf, inf).

early_stopping Whether to use early stopping to terminate training when the validation score is not improving. If set to True, it will automatically set aside a stratified fraction of training data as a validation set and terminate training when the validation score returned by the score method is not improving by at least `tol` for `n_iter_no_change` consecutive epochs.

New in version 0.20: Added `early_stopping` option.

validation_fraction The proportion of training data to set aside as a validation set for early stopping. Must be between 0 and 1. Only used if `early_stopping` is True. Values must be in the range (0.0, 1.0).

New in version 0.20: Added `validation_fraction` option.

n_iter_no_change Number of iterations with no improvement to wait before stopping fitting. Convergence is checked against the training loss or the validation loss depending on the `early_stopping` parameter. Integer values must be in the range [1, `max_iter`).

New in version 0.20: Added `n_iter_no_change` option.

class_weight Preset for the `class_weight` fit parameter.

Weights associated with classes. If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

warm_start When set to True, reuse the solution of the previous call to `fit` as initialization, otherwise, just erase the previous solution. See the Glossary.

Repeatedly calling `fit` or `partial_fit` when `warm_start` is True can result in a different solution than when calling `fit` a single time because of the way the data is shuffled. If a dynamic learning rate is used, the learning rate is adapted depending on the number of samples already seen. Calling `fit` resets this counter, while `partial_fit` will result in increasing the existing counter.

average When set to True, computes the averaged SGD weights across all updates and stores the result in the `coef_` attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches `average`. So `average=10` will begin averaging after seeing 10 samples. Integer values must be in the range [1, `n_samples`].

[3]:
```
import numpy as np
from sklearn.linear_model import SGDClassifier
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.pipeline import make_pipeline
X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
Y = np.array([1, 1, 2, 2])

X   = StandardScaler().fit_transform(X)
clf = SGDClassifier(max_iter=1000, tol=1e-3)
clf.fit(X, Y)
print(clf.predict([[-0.8, -1]]))
```

```
[1]
```

[4]:
```
clf.coef_
```

[4]:
```
array([[6.15828171, 9.73709834]])
```

[5]:
```
clf.intercept_
```

[5]:
```
array([-10.])
```

The signed distance to the hyperplane (computed as the dot product between the coefficients and the input sample, plus the intercept) is given by `SGDClassifier.decision_function`. It predict confidence scores for samples.

[15]:
```
np.dot(X[0], clf.coef_[0]) + clf.intercept_[0]
```

[15]:
```
-23.631937682570594
```

[16]:
```
clf.decision_function(X)[0]
```

[16]:
```
-23.631937682570594
```

Same result. Also `loss="log_loss"` or `loss="modified_huber"` enables the predict_proba method. SGDClassifier supports multi-class classification by combining multiple binary classifiers in a "one versus all" (OVA) scheme. At testing time, we compute the confidence score (i.e. the signed distances to the hyperplane) for each classifier and choose the class with the highest confidence. Note that, in principle, since they allow to create a probability model, loss="log_loss" and loss="modified_huber" are more suitable for one-vs-all classification. Similar you can use `SGDRegressor` for regression tasks. Do check out their documentation.

Additionally, hyperparameter tuning is what makes SGD a bit difficult to work with so don't make to try out different hyperparameters combinations.

There is another class called `SGDOneClassSVM` for outlier detection using SGD in sklearn that you can implement. We will talk about outlier detection in another tutorial but if you are already familier just check out the parameters in the sklearn user guide.

With early_stopping=True, the input data is split into a training set and a validation set. The model is then fitted on the training set, and the stopping criterion is based on the prediction score (using the score method) computed on the validation set. The size of the validation set can be changed with the parameter validation_fraction. With early_stopping=False, the model is fitted on the entire input data and the stopping criterion is based on the objective function computed on the training data.

Stochastic Gradient Descent is sensitive to feature scaling, so it is highly recommended to scale your data. For example, scale each attribute on the input vector X to [0,1] or [-1,+1], or standardize it to have mean 0 and variance 1. Note that the same scaling must be applied to the test vector to obtain meaningful results. Use pipeline for efficiency.

> When using Stochastic Gradient Descent, the training instances must be independent and identically distributed (IID) to ensure that the parameters get pulled toward the global optimum, on average. A simple way to ensure this is to shuffle the instances during training (e.g., pick each instance randomly, or shuffle the training set at the beginning of each epoch). If you do not shuffle the instances—for example, if the instances are sorted by label—then SGD will start by optimizing for one label, then the next, and so on, and it will not settle close to the global minimum - Hands-on Machine Learning with Scikit-Learn, Book

[32]:
```python
# How to implement incremental learning

from sklearn.linear_model import SGDClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

X, y = make_classification(n_samples=1000, n_features=20, n_informative=10,␣
↪n_classes=2, random_state=42)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

X_train.shape, y_train.shape
```

[32]:
```
((800, 20), (800,))
```

[34]:
```python
X_train[:1]
```

[34]:
```
array([[ 1.63167308, -0.15034837, -0.53928297, -0.29743706,  1.38213799,
         1.73274901, -0.60882316,  0.59953651, -0.96249883,  1.18806188,
         1.55542169,  1.18182829, -1.10283311,  0.08202993,  0.95389814,
        -2.45274034,  0.11900876, -0.4949102 , -4.68849407, -3.02324846]])
```

[35]:
```python
# reshape the data
X_train, X_test = X_train.reshape(-1,10,20), X_test.reshape(-1,10,20)
y_train, y_test = y_train.reshape(-1,10), y_test.reshape(-1,10)

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[35]: ((80, 10, 20), (20, 10, 20), (80, 10), (20, 10))

[36]:
```
X_train[:1]
```

[36]:
```
array([[[ 1.63167308, -0.15034837, -0.53928297, -0.29743706,
          1.38213799,  1.73274901, -0.60882316,  0.59953651,
         -0.96249883,  1.18806188,  1.55542169,  1.18182829,
         -1.10283311,  0.08202993,  0.95389814, -2.45274034,
          0.11900876, -0.4949102 , -4.68849407, -3.02324846],
        [ 0.37686672, -0.43921776, -2.84604946, -5.30301617,
          0.34678255, -2.37660023,  2.48376599, -1.7353561 ,
         -0.42698037,  0.69675786,  0.92325869, -1.16033686,
          2.79122616, -1.10001664,  1.42124598,  1.6408563 ,
         -6.15943398,  1.43174948, -3.93127164, -2.64397156],
        [-0.31780484, -1.66766527, -0.60051462, -1.83230178,
          0.21671964, -3.39767161,  0.99056683,  0.65150094,
         -0.60329685, -1.41682353, -2.64950166,  0.365169  ,
          1.7584907 ,  0.57355884, -1.75386267, -1.89258629,
         -0.73953598,  0.2448973 , -0.26135528,  1.29018838],
        [-2.17532956,  1.2114275 , -2.37848601, -2.30161022,
         -0.17963037,  1.64313612, -0.50544721,  1.25889661,
         -1.63824566,  0.77913063, -3.18894802, -1.57722837,
          1.31847646, -0.25959672, -3.94262278,  0.5983126 ,
         -0.84377885, -0.85019775,  1.00848336,  0.83619768],
        [-5.19745618, -1.86583976,  3.42737308,  3.5057424 ,
         -0.67009057,  1.55613032,  0.02846894,  0.27915153,
          0.08434103,  0.91202367, -0.91485246,  1.97994837,
          2.82097419,  0.84945626, -0.25872383,  0.87434555,
         -2.05943039,  0.18616196,  3.23422148, -2.44326501],
        [-1.04188018,  1.2199432 , -2.93712532, -2.32819538,
         -0.66496651, -1.76304163,  0.7203575 ,  1.83552702,
          1.26835861, -1.54282215,  1.42970291, -0.1823147 ,
          3.00007186,  0.93680568,  3.58750469,  1.95850708,
         -5.53655724,  0.10643542, -1.29211653,  2.55918797],
        [ 1.8937337 , -1.05154756,  0.57777217,  2.51771026,
          0.36001647, -2.02236081, -0.9777544 , -2.37589305,
         -0.58506317, -0.22815135, -1.15637051,  0.28244976,
         -0.47090024,  1.17384202,  0.26848334, -0.78454768,
         -0.10429346, -0.72438517,  2.18493118,  2.9947892 ],
        [-3.01192353,  1.10920798,  0.45766392,  0.05357479,
         -1.44371837,  2.78491985, -0.61022069,  0.27003879,
          0.2898112 , -0.48280069, -0.46037126,  1.43802872,
          5.08671655,  0.2780796 , -0.29463265,  1.01364739,
          1.08290988,  0.06887464,  7.17194539, -1.16645861],
        [ 0.99037967, -0.34034048, -1.14477103, -0.02300832,
         -0.08701522,  1.7791201 , -0.22572263,  0.67802055,
          0.77354762, -0.58312283,  0.53951475,  0.4867415 ,
```

```
        0.17120972, -1.05094117,  0.69432882, -0.62764109,
       -1.57824516,  0.19731341, -2.29203385, -0.29979786],
      [-3.10399057, -1.12587083,  1.3109647 ,  2.62253727,
       -0.32641763,  0.63386776,  1.60740455,  1.37190853,
       -0.08244596,  0.42935071, -1.32691939,  1.16095269,
        1.29308026,  0.68714031,  1.06414109,  1.1738282 ,
       -0.2140669 , -0.14522865,  2.98404877,  1.52419733]]])
```

[37]:

```python
clf = SGDClassifier(random_state = 42)

epochs = 10

# loop through data in 10 epochs
for k in range(epochs):

    # Inner loop processes each sample in the training set
    for i in range(X_train.shape[0]):

        # Get the current batch (single sample)
        X_batch, y_batch = X_train[i], y_train[i]

        # Update the model with the current batch using partial_fit
        clf.partial_fit(X_batch, y_batch, classes=list(range(2)))
```

[41]:

```python
y_test_preds = []
for j in range(X_test.shape[0]):
    y_preds = clf.predict(X_test[j])
    y_test_preds.extend(y_preds.tolist())

print(accuracy_score(y_test.reshape(-1), y_test_preds))
```

```
0.675
```

That's how you can use incremental learning. We will see more about online learning in tensorflow or deep learning tutorials.