softmax_regression_2

September 10, 2021

Name: Omar Khaled Mahmoud Safwat

Group: Alex. G3

[ ]:
```
import numpy as np
import pandas as pd

from sklearn import datasets
from sklearn.model_selection import StratifiedShuffleSplit

np.random.seed(42)
```

[ ]:
```
iris = datasets.load_iris()
X = iris["data"]
y = iris["target"]
df = pd.DataFrame({fname: values for fname, values in␣
 ↪zip(iris["feature_names"], X.T)})
df["target"] = y

df.head()
```

[ ]:
```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target
0       0
1       0
2       0
3       0
4       0
```

1 Mini-Batch GD

2 Softmax Regression

[ ]:
```python
class Softmax():
    """Class Implements Softmax Regresson for multinomial classification"""

    def __init__(self):
        self.X_train = None # User input data.
        self.y = None # Target feature
        self.y_ohe = None # Target Variable One-hot encoded
        self.weights = None # weights of model

    def encode_target(self, y):
        """One-hot encode target variable"""

        assert y.ndim == 1, "Target variable should be 1 dimensional"

        y_ohe = np.zeros((len(y), y.max() + 1))
        y_ohe[np.arange(len(y)), y] = 1
        return y_ohe

    def hypothesis(self, X=None, weights=None, batch_indices=None):
        """Hypothesis function"""
        if X is None:
            X = self.X
        if weights is None:
            weights = self.weights
        if batch_indices is None:
            batch_indices = list(range(len(self.y)))

        score_fun = X[batch_indices, :] @ weights # Score function for each␣
↪class, for each sample
        score_exp = np.exp(score_fun)

        # Returns a 2D Numpy array, prediction of each class for each sample
        # Shape: (m, k)
        # m: Number of samples, k: Number of classes
        return(score_exp / np.sum(score_exp, axis=1, keepdims=True))

    def loss(self, h, batch_indices):
        """Cost function"""
        y_ohe = self.y_ohe[batch_indices, :]
        h += 1e-7 # Add tolerance term for the log
        return -np.sum(y_ohe * np.log(h))

    def loss_prime(self, h, batch_indices):
        """Jacobian vector of cost function"""
        y_ohe = self.y_ohe[batch_indices, :]
```

```
        X = self.X[batch_indices, :]

        return (X.T @ (h - y_ohe))

    def initialize(self, learn_rate, batch_size):
        """
        Initialize first epoch

        Args:
            learn_rate (float): Learning rate for gradient descent

            guess (int, 2D numpy array): Initial guess for model weights

            batch_size (int): Training set batch size
        """
        # Initialize guesses
        self.weights_hist = np.random.randn(self.X.shape[1], self.y_ohe.
↪shape[1], 1) # Initialize weight vector, account for bias column

        # Initialize history
        self.loss_hist = np.empty(shape=(1))
        self.grad_hist = np.empty_like(self.weights_hist)

        self.n_points = batch_size
        self.epoch = 0
        self.learn_rate = learn_rate

    def update_weights_GD(self, idx_1, idx_2, weights):
        """
        Function updates weights along the direction of steepest descent and␣
↪stores results

        Args:
            idx_1 (int): Start index for the current batch

            idx_2 (int): End index for the current batch (non inclusive)

            weights (2D numpy array): Model weights
        """

        # Compute predictions
        self.h_pred = self.hypothesis(batch_indices=self.train_indx[idx_1:␣
↪idx_2], weights=weights)

        # Compute weights gradient
        grad_new = self.loss_prime(self.h_pred, self.train_indx[idx_1: idx_2])
        self.grad_hist = np.dstack((self.grad_hist, np.atleast_3d(grad_new)))
```

```
        # Update Weights
        weights = weights - self.learn_rate * grad_new
        self.weights_hist = np.dstack((self.weights_hist, np.
↪atleast_3d(weights)))

    def mini_batch_GD(self, learn_rate=0.01, n_batches=8, max_epochs=1e3,␣
↪seed=None):
        """
        Optimize weights using Mini_batch Gradient Descent

        Args:

            learn_rate (float, optional): Learning rate for gradient descent

            n_batches (int, optional): Number of batches for mini-batch␣
↪Gradient descent

            max_epochs (int, optional): Maximum number of iterations over the␣
↪entire training set

            seed (int, optional): seed the the data shuffeling after before␣
↪each epoch

        Returns:

            2d numpy array: Trained model weights
        """
        # Randomly shuffle the dataset's order
        if (seed is not None):
            np.random.seed(seed)

        self.n_batches = n_batches
        np.random.shuffle(self.train_indx)
        self.MAX_EPOCHS = max_epochs

        batch_size = len(self.train_indx) // n_batches

        # Initialize first epoch
        self.initialize(learn_rate, batch_size)

        # Train model using Mini_batches
        while (self.epoch < self.MAX_EPOCHS):
            for i in range(n_batches):
                idx_1 = i * batch_size
                idx_2 = idx_1 + batch_size
```

```
                self.update_weights_GD(idx_1, idx_2, self.weights_hist[:, :,␣
↪-1])

            # Update Predictions and Loss from last epoch
            self.h_pred = self.hypothesis(batch_indices=self.train_indx[:
↪batch_size], weights=self.weights_hist[:, :, -1])
            self.loss_hist = np.append(self.loss_hist, self.loss(self.h_pred,␣
↪self.train_indx[:batch_size]))

            # Increment epoch and shuffle data incides for next epoch
            self.epoch += 1
            np.random.shuffle(self.train_indx)

            # Record minimum loss so far if early stopping is used
            if self.early_stop == True:
                # Check if stop criteria threshold is met
                if (self.epoch - self.best_epoch) > self.early_stop_thresh:
                    self.is_converged = True
                    return self.weights_hist[:, :, -1]

                val_preds = self.hypothesis(batch_indices=self.valid_indx,␣
↪weights=self.weights_hist[:, :, -1])
                current_loss = self.loss(val_preds, self.valid_indx)
                if self.min_val_loss > current_loss:
                    self.min_val_loss = current_loss
                    self.best_epoch = self.epoch


        return self.weights_hist[:, :, -1]

    def fit(self, X, y, early_stop=False, early_stop_thresh=5, **kwargs):
        """Fit model to training data"""

        # X, and y are 2D Numpy arrays.
        # Add a column of ones for theta_0
        self.X = np.hstack((np.ones((len(X), 1)), X))
        self.y = y


        # One-hot encode target variable
        self.y_ohe = self.encode_target(y)
        all_indices = np.array([i for i in range(len(self.X))])

        self.early_stop = early_stop

        # Split the data to validation set to check early stop
        if self.early_stop == True:
```

```
            validation_mask = np.array([np.random.choice([0, 1], p=[0.7, 0.3]) for i in range(len(y))], dtype='bool')
            self.valid_indx = all_indices[validation_mask]
            self.train_indx = all_indices[~validation_mask]

            self.min_val_loss = float("inf")           # Value of minimum loss
            self.best_epoch = 0                        # Epoch with minimum loss
            self.early_stop_thresh = early_stop_thresh # Maximum number of epochs to perform without improvement in loss
            self.is_converged = False

        else:
            self.train_indx = all_indices

        self.weights = self.mini_batch_GD(**kwargs)

    def predict(self, X):
        """Returns predictions"""
        X = np.hstack((np.ones((X.shape[0], 1)), X))
        proba = self.hypothesis(X, self.weights, list(range(X.shape[0])))
        return np.argmax(proba, axis=1)

    def predict_proba(self, X):
        """Returns predictions for each class as probabilities"""
        X = np.hstack((np.ones((X.shape[0], 1)), X))
        return self.hypothesis(X, self.weights, list(range(X.shape[0])))

    def accuracy_score(self, X_test, y_test):
        """Returns model accuracy score"""
        return np.sum(self.predict(X_test) == y_test) / len(y_test)

    def show_summary(self):
        """Prints a brief summary after stop criteria is reached"""
        print("Solver summary:")
        print("=" * len("Solver summary:"))

        if self.early_stop == True:
            print("Number of epochs: ", self.best_epoch)
            print("Negative Log Likelihood: ", self.loss_hist[self.best_epoch])
            print("Early stop criteria was reached first: ", self.is_converged)
            print("Train accuracy: ", self.accuracy_score(self.X[self.train_indx, 1:], self.y[self.train_indx]))
            print("Validation accuracy: ", self.accuracy_score(self.X[self.valid_indx, 1:], self.y[self.valid_indx]))
        else:
            print("Number of epochs: ", self.epoch)
            print("Negative Log Likelihood: ", self.loss_hist[-1])
```

```
            print("Train accuracy: ", self.accuracy_score(self.X[self.
↪train_indx, 1:], self.y[self.train_indx]))
```

3 Train with Mini_batch GD

3.1 Without Early stop

[ ]:
```
# Split dataset into validation and training
all_indices = np.array([i for i in range(len(X))])
np.random.shuffle(all_indices)
validation_mask = np.array([np.random.choice([0, 1], p=[0.7, 0.3]) for i in␣
↪range(len(y))], dtype='bool')
train_indx, val_indx = all_indices[~validation_mask],␣
↪all_indices[validation_mask]

X_train, y_train = X[train_indx], y[train_indx]
X_val, y_val = X[val_indx], y[val_indx]

softmax = Softmax()
softmax.fit(X_train, y_train, early_stop=False, learn_rate=0.05)
softmax.show_summary()
print('Validation accuracy: ', softmax.accuracy_score(X_val, y_val))
```

```
Solver summary:
===============
Number of epochs:  1000
Negative Log Likelihood:  0.5375553478186633
Train accuracy:  0.9904761904761905
Validation accuracy:  0.9777777777777777
```

3.2 With Early stop

[ ]:
```
softmax = Softmax()
softmax.fit(X, y, early_stop=True, early_stop_thresh=300, learn_rate=0.05)
softmax.show_summary()
```

```
Solver summary:
===============
Number of epochs:  66
Negative Log Likelihood:  0.37147882094226636
Early stop criteria was reached first:  True
Train accuracy:  0.979381443298969
Validation accuracy:  0.9622641509433962
```

4 Cross Validation

[ ]:
```python
class KfoldCV():
  """Class implements k-fold cross validation"""

  def __init__(self, model):
      self.model = model # ML model object
      self.accuracy_scores=None

  def fit(self, X, y, n_folds=3, **kwargs):
    """Train machine learning model using k_folds cross validation"""

    self.accuracy_scores = []

    # Split Dataset
    split = StratifiedShuffleSplit(n_splits=n_folds, test_size=0.2,␣
 ↪random_state=42)
    for train_index, test_index in split.split(X, y):
        X_train, y_train = X[train_index], y[train_index]
        X_test, y_test = X[test_index], y[test_index]

        self.model.fit(X_train, y_train, **kwargs)
        self.accuracy_scores.append(self.model.accuracy_score(X_test, y_test))

  def predict(self, X):
    """Return model predictions for dataset X"""
    return self.model.predict(X)

  def predict_proba(self, X):
    """Return model probability predictions for dataset X"""
    return self.model.predict_proba(X)

  def show_summary(self):
    self.model.show_summary()

  def accuracy_score(self):
    """Returns average accuracy score from CV"""
    return np.mean(self.accuracy_scores)
```

4.1 CV without early stopping

[ ]:
```python
softmaxCV = KfoldCV(Softmax())
softmaxCV.fit(X, y, learn_rate=0.05, early_stop=False)
print("Cross Validation accuracy: ", softmaxCV.accuracy_score())
```

```
Cross Validation accuracy:  0.9111111111111111
```

4.2 CV with early stopping

[ ]:
```
softmaxCV = KfoldCV(Softmax())
softmaxCV.fit(X, y, learn_rate=0.05, early_stop=True, early_stop_thresh=300) #␣
↪With early stop criteria
print("Cross Validation accuracy: ", softmaxCV.accuracy_score())
```

```
Cross Validation accuracy:  0.9444444444444443
```