Week 3

Linear Regression

Baseline Linear Regression Model

💡 DummyRegressor

```
from sklearn.dummy import DummyRegressor

dummy_regr = DummyRegressor(strategy="mean")
dummy_regr.fit(X_train, y_train)
dummy_regr.predict(X_test)
dummy_regr.score(X_test, y_test)
```

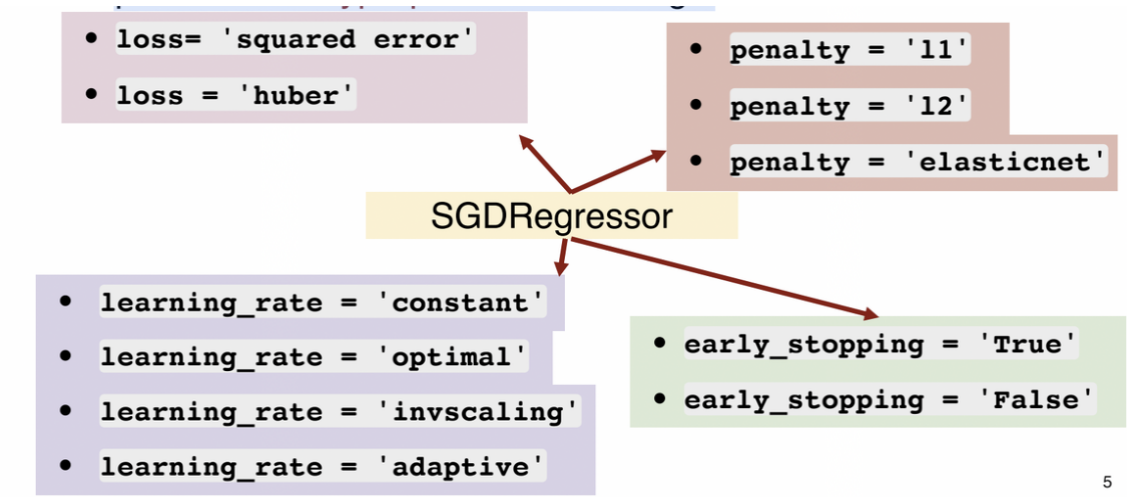
SGDRegressor Estimator

- Implements **stochastic gradient descent**
- Used for **large training set-up** (> 10k samples)
- senstive to feature scaling

```
from sklearn.linear_model import SGDRegressor
linear_regressor = SGDRegressor(random_state=42)
```

Hyperparameters

Provides greater control on optimization process through provision for hyperparameter settings.



- **Loss:**
  - **loss= 'squared error'**: Ordinary least squares

- **loss = 'huber':** Huber loss for robust regression

- **Regularisation:**
  - **Penalty = 'l2'**: L2 norm penalty on coef_ . This is default setting.
  - **penalty = 'l1'**: L1 norm penalty on coef_. This leads to sparse solutions.
  - **penalty = 'elasticnet'**: Convex combination of L2 and L1;
- **Learning Rate:**
  - **invscaling:** (default) The learning rate in t^{th} iteration or time step is calculated as:

$$\eta^t = \frac{\eta_0}{t^{power_t}}$$

  - **constant**
  - **adaptive:**
    - When the stopping criterion is reached, the learning rate is divided by 5, and the training loop continues.
    - The algorithm stops when the learning rate goes below 10 .
  - **optimal:**
    - Used as a default setting for classification problems.
    - The learning rate in t^{th} iteration or time step is calculated as:

$$\eta^t = \frac{1}{\alpha(t_0 + t)}$$

    - Here
      - α is a regularization rate.
      - t is the time step (there are a total of n_samples*n_iter time steps)
      - t_0 is determined based on a heuristic proposed by Léon Bottou such that the expected initial updates are comparable with the expected size of the weights (this assuming that the norm of the training samples is approx. 1).
- **Stopping Criteria:** SGDRegressor provides two stopping criteria to stop the algorithm when a given level of convergence is reached:
  - **early_stopping = True**

        - The input data is split into a training set and a validation set based on the validation_fraction parameter.
        - The model is fitted on the training set, and the stopping criterion is based on the prediction score (using the scoring method) computed on the validation set.
    - **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.

    In both cases, the criterion is evaluated once by epoch, and the algorithm stops when the criterion does not improve n_iter_no_change times in a row.
    The improvement is evaluated with absolute tolerance tol.
    The algorithm stops in any case after a maximum number of iteration max_iter

Shuffle training data after each epoch

```
from sklearn.linear_model import SGDRegressor
linear_regressor = SGDRegressor(shuffle=True)
```

Learning Rate

- constant - Constant learning rate is used throughout the training. eta0 = 1e-2
- adaptive
    - learning rate is kept to initial value as long as the training loss decreases.
    - When the stopping criterion is reached, the learning rate is divided by 5, and the training loop continues.
    - The algorithm stops when the learning rate goes below 10^{-6}
- invscaling

Defaults:

- Learning rate = invscaling
- eta0 = 1e-2
- power_t = 0.25

Learning rate reduces after every iteration:

$$eta = eta0 / pow(t, power_t)$$

Epochs

- Set **max_iter** to desired #epochs.
- The default value is 1000.

> 💡 SGD converges after observing approximately training samples. Thus, a reasonable first guess for the number of iterations for sampled training set is max_iter = np.ceil(10^6 /n)

Stopping Criteria

Option 1: tol, n_iter_no_change, max_iter

```
from sklearn.linear_model import SGDRegressor
linear_regressor = SGDRegressor(loss='squared_error',
                                max_iter=500,
                                tol=1e-3,
                                n_iter_no_change=5)
```

The SGDRegreesor stops

- when the training loss does not improve (loss > best_loss - tol) for n_iter_no_change consecutive epochs
- else after a maximum number of iteration max_iter.

Option 2: early_stopping, validation_fraction

```
from sklearn.linear_model import SGDRegressor
linear_regressor = SGDRegressor(loss='squared_error',
                                early_stopping=True
                                max_iter=500,
                                tol=1e-3,
                                validation_fraction=0.2,
                                n_iter_no_change=5)
```

Set aside validation_fraction percentage records from training set as validation set. Use score method to obtain validation score.

The SGDRegreesor stops when

- validation score does not improve by at least tol for n_iter_no_change consecutive epochs.
- else after a maximum number of iteration max_iter.

Averaged SGD

Averaged SGD updates the weight vector to average of weights from previous updates.

Option #1: Averaging across all updates average=True

```
from sklearn.linear_model import SGDRegressor
linear_regressor = SGDRegressor(average=True)
```

Option #2: Set average to int value

Averaging begins once the total number of samples seen reaches average

```
from sklearn.linear_model import SGDRegressor
linear_regressor = SGDRegressor(average=10)
```

Averaged SGD works best with a larger number of features and a higher eta0

Initializing SGD with weight vector of the previous run

```
from sklearn.linear_model import SGDRegressor
linear_regressor = SGDRegressor(warm_start=True)
```

Monitoring SGD loss iteration after iteration

```
sgd_reg = SGDRegressor(max_iter=1, tol=-np.infty, warm_start=True,
                       penalty=None, learning_rate="constant", eta0=0.0005)

for epoch in range(1000):
    sgd_reg.fit(X_train, y_train) # continues where it left off
    y_val_predict = sgd_reg.predict(X_val)
    val_error = mean_squared_error(y_val, y_val_predict)
```

Model Inspection

Access the weights of trained Linear Regression model

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_m x_m = \mathbf{w}^T \mathbf{x}$$

Weights

```
linear_regressor.coef_
```

Intercept

```
linear_regressor.intercept_
```

Model Inference

Making predictions on new data

Step 1: Arrange data for prediction in a feature matrix of shape (#samples, #features) or in sparse matrix format.

Step 2: Call predict method on linear regression object with feature matrix as an argument.

```
linear_regressor.predict(X_test)
```

Model Evaluation

General Steps

STEP 1: Split data into train and test

STEP 2: Fit linear regression estimator on training set.

STEP 3: Calculate training error (a.k.a. empirical error)

STEP 4: Calculate test error (a.k.a. generalization error)

Evaluating a trained Linear Regression model

```
linear_regressor.score(X_test, y_test)
```

The score returns R or coefficient of determination

Evaluation Metrics

sklearn.metrics

- mean_absolute_error
- mean_squared_error
- r2_score (Same as score) - range $(-\inf, 1]$
- mean_squared_log_error (Useful for targets with exponential growths)
- mean_absolute_percentage_error (Sensitive to relative error)
- median_absolute_error (robust to outliers)

Evaluating regression model on worst case error

```
from sklearn.metrics import max_error
test_error = max_error(y_test, y_predicted)
```

It does not support multi-output regression.

Scores and Errors

- **Score:** higher the better
- **Error:** lower the better

Convert error metric to score metric by adding neg_ suffix.

Function	Scoring
metrics.mean_absolute_error	neg_mean_absolute_error
metrics.mean_squared_error	neg_mean_squared_error
metrics.mean_squared_error	neg_root_mean_squared_error
metrics.mean_squared_log_error	neg_mean_squared_log_error
metrics.median_absolute_error	neg_median_absolute_error

Robust Evaluation - Cross Validation

Cross-validation performs robust evaluation of model performance

- by repeated splitting and
- providing many training and test errors

This enables us to **estimate variability in generalization** performance of the model.

sklearn implements the following cross validation iterators

- KFold
- RepeatedKfold
- LeaveOneOut
- ShuffleSplit

Kfold

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import linear_regression

lin_reg = linear_regression()

score = cross_val_score(lin_reg, X, y, cv=5)
#or
kfold_cv = KFold(n_splits=5, random_state=42)
score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

- Uses KFold cross validation iterator, that divides training data into 5 folds.
- In each run, it uses 4 folds for training and 1 for evaluation.

LeaveOneOut

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import LeaveOneOut
from sklearn.linear_model import linear_regression

lin_reg = linear_regression()

loocv = LeaveOneOut()
score = cross_val_score(lin_reg, X, y, cv=loocv)
#or
n = X.shape[0]
kfold_cv = KFold(n_splits=n)
score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

ShuffleSplit

```
from sklearn.linear_model import linear_regression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import ShuffleSplit

lin_reg = linear_regression()

shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
score = cross_val_score(lin_reg, X, y, cv=shuffle_split)
```

It is also called random permutation based cross validation strategy.

- Generates user defined number of train/test splits.
- It is robust to class distribution.

In each iteration, it shuffles order of data samples and then splits it into train and test.

Obtaining test scores from different folds

```
from sklearn.model_selection import cross_validate
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=40, test_size=0.3, random_state=0)
cv_results = cross_validate(
        regressor, data, target, cv=cv, scoring="neg_mean_absolute_error")
```

The results are stored in python dictionary with the following keys:

- fit_time

- score_time
- test_score
- estimator
- train_score

Obtain trained estimators and scores on training data during cross validation?

- For trained estimator, set return_estimator = True
- For scores on training set, set return_train_score = True

```
from sklearn.model_selection import cross_validate
from sklearn.model_selection import ShuffleSplit
cv = ShuffleSplit(n_splits=40, test_size=0.3,random_state=0)
cv_results = cross_validate(
                            regressor, data, target,
                            cv=cv,
                            scoring=["neg_mean_absolute_error",
                            "neg_mean_squared_error"],
                            return_train_score=True,
                            return_estimator=True)
```

The estimators can be accessed through estimator key of the dictionary returned by cross_validate

Study effect of #samples on training and test errors

STEP 1: Instantiate an object of learning_curve class with estimator, training data, size, cross validation strategy and scoring scheme as arguments.

```
from sklearn.model_selection import learning_curve
results = learning_curve(
                lin_reg, X_train, y_train, train_sizes=train_sizes, cv=cv,
                scoring="neg_mean_absolute_error")
train_size, train_scores, test_scores = results[:3]
# Convert the scores into errors
train_errors, test_errors = -train_scores, -test_scores
```

STEP 2: Plot training and test scores as function of the size of training sets. And make assessment about model fitment: under/overfitting or right fit.

Underfitting/Overfitting Diagnosis

STEP 1: Fit linear models with different number of features.

STEP 2: For each model, obtain training and test errors.

STEP 3: Plot #features vs error graph - one each for training and test errors.

STEP 4: Examine the graphs to detect under/overfitting.