Logistic Regression

In this notebook, we will learn how to apply Logistic regression for predicting the compressive strength of concrete.

The attached dataset is taken from the [UC Irvine Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength)](https://archive.ics.uci.edu/ml/datasets/Concrete+Compressive+Strength).

To run this code, you will need the following python packages:

- numpy
- pandas
- scikit-learn

In [1]:

```
import numpy as np
import pandas as pd
```

In [2]:

```
# First, we load the dataset using pandas
df = pd.read_excel("Concrete_Data.xlsx")
```

In [3]:

```
# next, we will split the dataframe into a training and testing splits with a 70% / 30% ratio
from sklearn.model_selection import train_test_split

df_train, df_test = train_test_split(df, test_size=0.3, random_state=42) # Random is fixed for re
producability
```

In [4]:
```
df_train
```

Out[4]:

	Cement (component 1)(kg in a m^3 mixture)	Blast Furnace Slag (component 2) (kg in a m^3 mixture)	Fly Ash (component 3)(kg in a m^3 mixture)	Water (component 4)(kg in a m^3 mixture)	Superplasticizer (component 5) (kg in a m^3 mixture)	Coarse Aggregate (component 6) (kg in a m^3 mixture)	Fine Aggregate (component 7) (kg in a m^3 mixture)	Age (day)	Concrete compressive strength(MPa, megapascals)
196	194.68	0.0	100.52	165.62	7.48	1006.4	905.90	28	25.724350
631	325.00	0.0	0.00	184.00	0.00	1063.0	783.00	7	17.540269
81	318.80	212.5	0.00	155.70	14.30	852.1	880.40	3	25.200348
526	359.00	19.0	141.00	154.00	10.91	942.0	801.00	3	23.639177
830	162.00	190.0	148.00	179.00	19.00	838.0	741.00	28	33.756745
...	...	...	...	...	...	...	...	...	...
87	286.30	200.9	0.00	144.70	11.20	1004.6	803.70	3	24.400556
330	246.83	0.0	125.08	143.30	11.99	1086.8	800.89	14	42.216615
466	190.34	0.0	125.18	166.61	9.88	1079.0	798.90	100	33.563692
121	475.00	118.8	0.00	181.10	8.90	852.1	781.50	28	68.299493
860	314.00	0.0	113.00	170.00	10.00	925.0	783.00	28	38.458971

721 rows × 9 columns

In [5]:
```
df_train.describe()
```

Out[5]:

	Cement (component 1)(kg in a m^3 mixture)	Blast Furnace Slag (component 2)(kg in a m^3 mixture)	Fly Ash (component 3)(kg in a m^3 mixture)	Water (component 4)(kg in a m^3 mixture)	Superplasticizer (component 5) (kg in a m^3 mixture)	Coarse Aggregate (component 6)(kg in a m^3 mixture)	Fine Aggregate (component 7)(kg in a m^3 mixture)	Age (day)	Concrete compressive strength(MPa, megapascals)
count	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000	721.000000
mean	284.409681	74.971886	52.006588	181.805576	6.125337	973.798128	771.636297	46.049931	36.152573
std	108.361334	87.717335	63.707358	21.159956	6.046367	78.509208	80.125492	61.650743	16.803402
min	102.000000	0.000000	0.000000	121.750000	0.000000	801.000000	594.000000	1.000000	2.331808
25%	192.000000	0.000000	0.000000	165.620000	0.000000	932.000000	724.300000	14.000000	23.890343
50%	277.000000	22.000000	0.000000	185.700000	6.000000	968.000000	778.450000	28.000000	35.076402
75%	362.600000	145.000000	117.540000	192.000000	10.100000	1040.000000	821.000000	56.000000	46.247292
max	540.000000	359.400000	195.000000	247.000000	32.200000	1145.000000	992.600000	365.000000	82.599225

In [6]:
```
# Now we will extract the models input and targets from both the training and testing dataframes
def extract_Xy(df):
    df_numpy = df.to_numpy()
    return df_numpy[:, :-1], df_numpy[:, -1]

X_train, y_train = extract_Xy(df_train)
X_test, y_test = extract_Xy(df_test)

y_median = np.median(y_train)
print("Median value of the target:", y_median)

# Since we will treat this as a classification task, we will assume that
# the concrete is "strong" (y = True) if its compressive ratio is higher than the median
# otherwise, it is assumed to be "weak" (y = False)
y_train = y_train > y_median
y_test = y_test > y_median

# Now ~50% of the samples should be considered "strong" and the rest are considered "weak"
print(f"Percentage of 'strong' samples: {y_train.mean() * 100} %")

# Also, lets standardize the data since it improves the training process
X_mean = X_train.mean(axis=0)
X_std = X_train.std(axis=0)
X_train = (X_train - X_mean)/(1e-8 + X_std)
X_test = (X_test - X_mean)/(1e-8 + X_std)
```

```
Median value of the target: 35.076402024
Percentage of 'strong' samples: 49.930651872399444 %
```

Logistic Regression via Scikit-Learn

In [7]:
```
from sklearn.linear_model import LogisticRegression
```

In [8]:
```
%%time
# We use time to compute the training time of our model
model = LogisticRegression(random_state=0, penalty="none").fit(X_train, y_train)
```

```
CPU times: user 12.9 ms, sys: 1.29 ms, total: 14.2 ms
Wall time: 14.2 ms
```

In [9]:
```
from sklearn.metrics import accuracy_score

y_train_predict = model.predict(X_train)
print(f"Training Accurracy: {accuracy_score(y_train, y_train_predict) * 100}%")
y_test_predict = model.predict(X_test)
print(f"Testing Accurracy: {accuracy_score(y_test, y_test_predict) * 100}%")
```

```
Training Accurracy: 85.5755894590846%
Testing Accurracy: 84.14239482200647%
```

Logistic Regression from Scratch

In [10]:
```
def sigmoid(x):
    #TODO: Implement sigmoid (hint: use np.exp)
    return 1/(1 + np.exp(-x))
```

In [12]:
```
# Sanity checks
print(f"{sigmoid(-1e2) = }") # This should be almost equal 0
print(f"{sigmoid(   0) = }") # This should be exactly 0.5
print(f"{sigmoid(+1e2) = }") # This should be almost equal 1
```

```
sigmoid(-1e2) = 3.7200759760208356e-44
sigmoid(   0) = 0.5
sigmoid(+1e2) = 1.0
```

In [13]:
```
def our_accuracy_score(true, predicted):
    #TODO: Implement an accuracy metric so that is can be used instead of Sklearn's accuracy scor
e
    #Note: both true and predicted will be boolean numpy array
    return np.mean(true == predicted)
```

In [14]:
```
# Sanity checks
print(f"{our_accuracy_score( np.array([True,  True]), np.array([True,  True]) ) = }") # Should be 1
print(f"{our_accuracy_score( np.array([True, False]), np.array([True,  True]) ) = }") # Should be 0.5
print(f"{our_accuracy_score( np.array([True, False]), np.array([True, False]) ) = }") # Should be 1
print(f"{our_accuracy_score( np.array([False, True]), np.array([True, False]) ) = }") # Should be 0
```

```
our_accuracy_score( np.array([True,  True]), np.array([True,  True]) ) = 1.0
our_accuracy_score( np.array([True, False]), np.array([True,  True]) ) = 0.5
our_accuracy_score( np.array([True, False]), np.array([True, False]) ) = 1.0
our_accuracy_score( np.array([False, True]), np.array([True, False]) ) = 0.0
```

In [118]:
```
#IMPORTANT: You can only use numpy here. Do not use any premade algorithms (e.g. Scikit-Learn's L
ogistic Regression)
class OurLogisticRegression:
    def __init__(self, lr: int, epochs: int, probability_threshold: float = 0.5, random_state = N
one):
        self.lr = lr # The learning rate
        self.epochs = epochs # The number of training epochs
        self.probability_threshold = probability_threshold # If the output of the sigmoid functio
n is > probability_threshold, the prediction is considered to be positive (True)
                                                            # otherwise, the prediction is conside
red to be negative (False)
        self.random_state = random_state # The random state will be used set the random seed for
 the sake of reproducability

    def _prepare_input(self, X):
        # Here, we add a new input with value 1 to each example. It will be multipled by the bias
        ones = np.ones((X.shape[0], 1), dtype=X.dtype)
        return np.concatenate((ones, X), axis=1)

    def _prepare_target(self, y):
        # Here, we convert True to +1 and False to -1
        #TODO (Optional): You can modify your function if you wish to used other values for the p
ositive and negative classes
        return np.where(y, 1, -1)

    def _initialize(self, num_weights: int, stdev: float = 0.01):
        # Here, we initialize the weights using a normally distributed random variable with a sma
ll standard deviation
        self.w = np.random.randn(num_weights) * stdev

    def _gradient(self, X, y):
        #TODO: Compute and return the gradient of the weights (self.w) wrt to the loss given the
 X and y arrays
        return np.dot(X.T, (sigmoid(np.dot(X, self.w)) - y)) / X.shape[0]

    def _update(self, X, y):
        #TODO: Implement this function to apply a single iteration on the weights "self.w"
        #Hint: use self._gradient
        self.w = self.w - self.lr * self._gradient(X, y)

    def fit(self, X, y):
        np.random.seed(self.random_state) # First, we set the seed
```

```
        X = self._prepare_input(X) # Then we prepare the inputs
        y = self._prepare_target(y) # and prepare the targets too
        self._initialize(X.shape[1]) # and initialize the weights randomly
        for _ in range(self.epochs): # Then we update the weights for a certain number of epochs
            self._update(X, y)
        return self # Return self to match the behavior of Scikit-Learn's LinearRegression fit()

    def predict(self, X):
        X = self._prepare_input(X)
        #TODO: Implement the rest of this function (Note: It should return a boolean array)
        return sigmoid(np.dot(X, self.w)) >= self.probability_threshold
```

In [82]:

```
# We will use this function to tune the hyper parameters
def validate(lr, epochs):
    validation_size = 0.1 #TODO: Choose a size for the validation set as a ratio from the trainin
g data
    X_tr, X_val, y_tr, y_val = train_test_split(X_train, y_train, test_size=validation_size, rand
om_state=42)
    # We will fit the model to only a subset of the training data and we will use the rest to eva
luate the performance
    our_model = OurLogisticRegression(lr=lr, epochs=epochs, random_state=0).fit(X_tr, y_tr)
    # Then, we evaluate the peformance using the validation set
    return our_accuracy_score(y_val, our_model.predict(X_val))
```

In [124]:

```
lr = 0.001 #TODO: Choose a learning rate to use while testing different values for the number of
 epochs
epochs_values = [10, 100, 1000, 10000, 100000] #TODO: Choose a list of values for the number of e
pochs to test
for epochs in epochs_values:
    accuracy = validate(lr, epochs)
    print(f"In {epochs} epochs, the accuracy reaches {accuracy * 100}% using lr={lr}")
```

```
In 10 epochs, the accuracy reaches 45.20547945205479% using lr=0.001
In 100 epochs, the accuracy reaches 73.97260273972603% using lr=0.001
In 1000 epochs, the accuracy reaches 75.34246575342466% using lr=0.001
In 10000 epochs, the accuracy reaches 71.23287671232876% using lr=0.001
In 100000 epochs, the accuracy reaches 71.23287671232876% using lr=0.001
```

In [129]:
```
epochs = 1000 #TODO: Choose the number of epochs to use while testing different values for the le
arning rate
lr_values = [0.1, 0.5, 0.01, 0.05, 0.001, 0.005, 0.0001, 0.0005, 0.00001, 0.00005] #TODO: Choose
 a list of values for the learning rate to test
for lr in lr_values:
    accuracy = validate(lr, epochs)
    print(f"Using lr={lr}, the accuracy reaches {accuracy * 100}% in {epochs} epochs")
```

```
Using lr=0.1, the accuracy reaches 71.23287671232876% in 1000 epochs
Using lr=0.5, the accuracy reaches 71.23287671232876% in 1000 epochs
Using lr=0.01, the accuracy reaches 71.23287671232876% in 1000 epochs
Using lr=0.05, the accuracy reaches 71.23287671232876% in 1000 epochs
Using lr=0.001, the accuracy reaches 75.34246575342466% in 1000 epochs
Using lr=0.005, the accuracy reaches 71.23287671232876% in 1000 epochs
Using lr=0.0001, the accuracy reaches 73.97260273972603% in 1000 epochs
Using lr=0.0005, the accuracy reaches 76.71232876712328% in 1000 epochs
Using lr=1e-05, the accuracy reaches 45.20547945205479% in 1000 epochs
Using lr=5e-05, the accuracy reaches 71.23287671232876% in 1000 epochs
```

In [140]:
```
%%time
# We use time to compute the training time of our model
#TODO: Select an appropriate learning rate and number of epochs
lr = 0.0005
epochs = 1000
our_model = OurLogisticRegression(lr=lr, epochs=epochs, random_state=0).fit(X_train, y_train)
```

```
CPU times: user 30.9 ms, sys: 0 ns, total: 30.9 ms
Wall time: 29.8 ms
```

In [141]:
```
y_train_predict = our_model.predict(X_train)
print(f"Training Accuracy: {our_accuracy_score(y_train, y_train_predict) * 100}%")
y_test_predict = our_model.predict(X_test)
print(f"Testing Accuracy: {our_accuracy_score(y_test, y_test_predict) * 100}%")
```

```
Training Accuracy: 70.18030513176144%
Testing Accuracy: 67.63754045307444%
```

In [142]:
```
#TODO: Write your conclusion about your implementation's performance and training time
```

The sklearn implementation completely beats my implementation's performance and training time but I think it's not fair to compare both implementations because to the best of my knowledge, sklearn does not implement gradient descent in LogisticRegression()

Bonus

As a bonus, you can implement and test the following:

- Stochastic gradient descent
- Termination conditions (e.g. The gradient check)

Write your conclusion about any results you calculate for your bonus implementations.

IMPORTANT: Do not implement the bonus in the previous cells. You can copy and paste codes from the previous cells and continue your implementation below this cell.