Logistic Regression

```python
class LogisticRegression(object):

    def __init__(self, learning_rate=0.1, max_iter=100, regularization='l2', lambda_ = 10 , tolerance = 1e-4):
        self.learning_rate  = learning_rate
        self.max_iter       = max_iter
        self.regularization = regularization
        self.lambda_        = lambda_
        self.tolerance      = tolerance
        self.loss_log       = []

    def fit(self, X, y, verbose = False):
        no_samples = X.shape[0]
        no_features = X.shape[1]

        self.W = np.random.rand(no_features + 1) # random initialization, +1 for bias

        extra_feature_with_value_1 = np.ones((no_samples, 1))
        X = np.concatenate((extra_feature_with_value_1, X), axis = 1)  # match dimension with W. Optimizes the bias calculation

        self.loss_log = []

        for iteration in range(self.max_iter):
            Z = np.matmul(X,  self.W)

            y_hat = self.__sigmoid(Z)

            errors = y_hat - y

            if self.regularization:
              cost = (-1.0/no_samples) * np.sum( y*np.log(y_hat) + (1.0 - y)*np.log(1.0-y_hat)) + (1.0/no_samples)* self.lambda_ * np.matmul(self.W, np.transpose(self.W))
            else:
              cost = (-1.0/no_samples) * np.sum( y*np.log(y_hat) + (1.0 - y)*np.log(1.0-y_hat))
            self.loss_log.append(cost)

            if verbose:
                print(f'Iteration {iteration} Loss: {cost}') # For printing loss of every epoch

            if self.regularization is not None:
                delta_grad = (1./no_samples) * (np.matmul(np.transpose(errors), X)+ self.lambda_ * self.W)
            else:
                delta_grad = (1./no_samples) * (np.matmul(np.transpose(errors), X))

            self.W -= self.learning_rate * delta_grad

        return self

    def predict_proba(self, X):
        no_samples = X.shape[0]
        no_features = X.shape[1]

        samples = np.reshape(X, (no_samples, no_features))
        weights = np.reshape(self.W[1:], (no_features, 1))

        samples = np.matrix(samples)
        weights = np.matrix(weights)

        wtx = np.matmul(samples, weights)

        z = wtx + self.W[0]

        probabilities = self.__sigmoid(z)
        #return self.__sigmoid((X @ self.W[1:]) + self.W[0])
        return probabilities

    def predict(self, X):
        return np.round(self.predict_proba(X))

    def __sigmoid(self, z):
        return (1.0 / (1.0 + np.exp(-z)))

    def get_params(self):
        try:
            params = dict()
            params['intercept'] = self.W[0]
            params['coef'] = self.W[1:]
            return params
        except:
            raise Exception('Fit the model first!')
```