```
import numpy as np

class SVM:
    def __init__(self, C=1.0, tol=0.001, max_iter=100):
        self.C = C  # Regularization parameter
        self.tol = tol  # Tolerance for stopping criterion
        self.max_iter = max_iter  # Maximum number of iterations
        self.alpha = None  # Lagrange multipliers
        self.b = 0  # Bias term
        self.X = None
        self.y = None

    def fit(self, X, y):
        self.X = X
        self.y = y
        n_samples, n_features = X.shape
        self.alpha = np.zeros(n_samples)

        for _ in range(self.max_iter):
            num_changed_alphas = 0
            for i in range(n_samples):
                E_i = self._predict(X[i]) - y[i]

                if (y[i] * E_i < -self.tol and self.alpha[i] < self.C) or \
                   (y[i] * E_i > self.tol and self.alpha[i] > 0):

                    j = np.random.choice(n_samples)
                    while j == i:
                        j = np.random.choice(n_samples)

                    E_j = self._predict(X[j]) - y[j]
                    alpha_i_old, alpha_j_old = self.alpha[i], self.alpha[j]

                    L, H = self._compute_L_H(self.alpha[i], self.alpha[j], y[i], y[j])
                    if L == H:
                        continue

                    eta = 2.0 * X[i] @ X[j] - X[i] @ X[i] - X[j] @ X[j]
                    if eta >= 0:
                        continue

                    self.alpha[j] -= y[j] * (E_i - E_j) / eta
                    self.alpha[j] = np.clip(self.alpha[j], L, H)

                    if abs(self.alpha[j] - alpha_j_old) < 1e-5:
                        continue

                    self.alpha[i] += y[i] * y[j] * (alpha_j_old - self.alpha[j])
                    b1 = self.b - E_i - y[i] * (self.alpha[i] - alpha_i_old) * X[i] @ X[i] \
                         - y[j] * (self.alpha[j] - alpha_j_old) * X[i] @ X[j]
                    b2 = self.b - E_j - y[i] * (self.alpha[i] - alpha_i_old) * X[i] @ X[j] \
                         - y[j] * (self.alpha[j] - alpha_j_old) * X[j] @ X[j]
                    self.b = 0.5 * (b1 + b2)

                    num_changed_alphas += 1

            if num_changed_alphas == 0:
                break

    def _compute_L_H(self, alpha_i, alpha_j, y_i, y_j):
        if y_i != y_j:
            return max(0, alpha_j - alpha_i), min(self.C, self.C + alpha_j - alpha_i)
        return max(0, alpha_i + alpha_j - self.C), min(self.C, alpha_i + alpha_j)

    def _predict(self, X):
        return np.sum(self.alpha * self.y * self.kernel(X, self.X)) + self.b

    def kernel(self, X1, X2):
        return np.dot(X1, X2.T)

    def predict(self, X):
        return np.sign(self._predict(X))

# Example usage:
X = np.array([[1, 2], [2, 3], [3, 3], [2, 1], [3, 2]])
y = np.array([-1, -1, 1, 1, 1])
svm = SVM(C=1.0, tol=0.001, max_iter=100)
svm.fit(X, y)
predictions = svm.predict(X)
print(predictions)
```

```
1.0
```

Double-click (or enter) to edit