1. Random Forest

```
import numpy as np

class DecisionTree:
    def __init__(self, depth=5):
        self.depth = depth
        self.tree = None

    def fit(self, X, y):
        self.tree = self._grow_tree(X, y, depth=self.depth)

    def _grow_tree(self, X, y, depth):
        if depth == 0 or len(set(y)) == 1:
            return np.mean(y)

        best_feature, best_threshold = self._best_split(X, y)
        left_mask = X[:, best_feature] < best_threshold
        right_mask = ~left_mask

        return {
            'feature': best_feature,
            'threshold': best_threshold,
            'left': self._grow_tree(X[left_mask], y[left_mask], depth-1),
            'right': self._grow_tree(X[right_mask], y[right_mask], depth-1)
        }

    def _best_split(self, X, y):
        best_feature, best_threshold, best_score = None, None, float('inf')
        for feature in range(X.shape[1]):
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                left_mask = X[:, feature] < threshold
                right_mask = ~left_mask
                score = np.var(y[left_mask]) * len(y[left_mask]) + np.var(y[right_mask]) *
len(y[right_mask])
                if score < best_score:
                    best_feature, best_threshold, best_score = feature, threshold, score
        return best_feature, best_threshold

    def predict(self, X):
        return np.array([self._traverse_tree(x, self.tree) for x in X])

    def _traverse_tree(self, x, node):
```

```
        if isinstance(node, dict):
            if x[node['feature']] < node['threshold']:
                return self._traverse_tree(x, node['left'])
            else:
                return self._traverse_tree(x, node['right'])
        return node
```

2. Support Vector Machine

```
import numpy as np

class SVM:
    def __init__(self, lr=0.01, lambda_param=0.01, n_iters=1000):
        self.lr = lr
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        y = np.where(y <= 0, -1, 1)
        self.w = np.zeros(X.shape[1])
        self.b = 0

        for _ in range(self.n_iters):
            for i, x in enumerate(X):
                condition = y[i] * (np.dot(x, self.w) - self.b) >= 1
                if condition:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w - np.dot(x, y[i]))
                    self.b -= self.lr * y[i]

    def predict(self, X):
        return np.sign(np.dot(X, self.w) - self.b)
```

3. Hierarchical Clustering using Complete Linkage

```
import numpy as np
import itertools

def complete_linkage(X):
    clusters = {i: [i] for i in range(len(X))}
    distances = {(i, j): np.linalg.norm(X[i] - X[j]) for i, j in itertools.combinations(range(len(X)), 2)}

    while len(clusters) > 1:
        (c1, c2), _ = min(distances.items(), key=lambda x: x[1])
        clusters[min(c1, c2)] += clusters.pop(max(c1, c2))
        distances = {(i, j): max(distances.get((i, j), float('inf')), distances.get((j, i), float('inf')))
                     for i, j in itertools.combinations(clusters.keys(), 2)}
    return clusters
```

4. Apriori Algorithm

```
import itertools

def apriori(transactions, min_support=0.5):
    item_sets = {frozenset([item]) for transaction in transactions for item in transaction}
    n_transactions = len(transactions)
    frequent_sets = []

    while item_sets:
        counts = {item: sum(1 for t in transactions if item.issubset(t)) for item in item_sets}
        item_sets = {item for item, count in counts.items() if count / n_transactions >= min_support}
        frequent_sets.extend(item_sets)
        item_sets = {i | j for i in item_sets for j in item_sets if len(i | j) == len(i) + 1}

    return frequent_sets
```