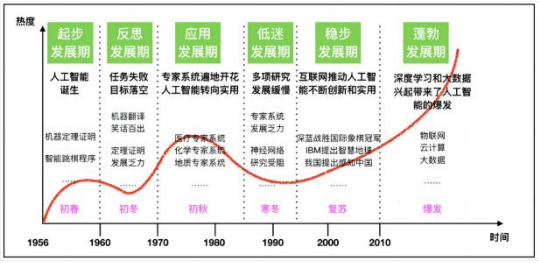
Lec1 intro

Disembodied Intelligence 非具身智能：符号主义（逻辑推理）、联结主义（脑科学）；具身智能：行为主义（感知-行动，控制论）



Machine learning is an application or subset of AI that allows machines to learn from data without being programmed explicitly
Classification, Recognizing patterns, Recommender Systems, Information retrieval, Computer vision, Robotics, Learning to play games, Recognizing anomalies, Spam filtering, fraud detection
Supervised Learning: training set of inputs and outputs, classification (1-of-N output), regression (real-valued output)
Unsupervised Learning: input data, form cluster of extract features
Reinforcement Learning: agent, observation, action, reward; only reward signal, feedback delayed, time matters

Lec3 Linear Regression

model: $y(x) = w_0 + w_1 x = (w_0, w_1, ..., w_d)^T (1, x_1, ..., x_d)$ MSE loss: $l(w) = \frac{1}{2N}\sum_{n=1}^{N}[t^{(n)} - y^{(n)}]^2$ gradient: $\nabla l(w) = -\frac{1}{N}\sum_{n=1}^{N}(t^{(n)} - w^T x^{(n)})x^{(n)}$

least square approach, $= -\frac{1}{N}\sum_{n=1}^{N} X^T(t - Xw) = 0$, so $w = (X^T X)^{-1} X^T t$

gradient descent: $w \leftarrow w - \lambda \nabla l(w)$

Algorithm 1. Stochastic gradient descent (SGD)

1. Initialize w (e.g., randomly)
2. **for** $i = 1$ to n_epoch **do**
3. Randomly shuffle and pick one sample $(x^{(n)}, t^{(n)})$ in the training set
4. Update: $w \leftarrow w + \lambda\left[t^{(n)} - y(x^{(n)})\right]\mathbf{x}^{(n)}$ (error)
5. **end for**

Algorithm 2. Batch gradient descent (BGD)

1. Initialize w (e.g., randomly)
2. **for** $i = 1$ to n_epoch **do**
3. Update: $w \leftarrow w + \lambda\frac{1}{N}\sum_{n=1}^{N}\left[t^{(n)} - y(x^{(n)})\right]\mathbf{x}^{(n)}$
4. **end for**

Algorithm 3. Mini-Batch gradient descent (MBGD)

1. Initialize w (e.g., randomly)
2. **for** $i = 1$ to n_epoch **do**
3. shuffle the training set and partition into a number of mini-batches
4. **for** $j = 1$ to floor($\frac{N}{m}$), **do**
5. Update: $w \leftarrow w + \lambda\frac{1}{m}\sum_{n\in\mathcal{B}_j}\left[t^{(n)} - y(x^{(n)})\right]\mathbf{x}^{(n)}$
6. **end for**
7. **end for**

梯度下降噪声大，但易于跳出局部最优。占据内存最小，适合大数据集，适合在线学习。
收敛速度慢，稳定性差，可能不收敛。

迭代速度慢，算力高，占据大量存储
易于收敛，稳定性高

需要调参
利用了并行计算的优势

```
def batch_update(self, X, y):
    if self.tol is not None:
        loss_old = np.inf  # 初始化旧损失值为无穷大
    for iter in range(self.n_iter):
        y_pred = self._predict(X)              # 预测
        loss = self._loss(y, y_pred)           # 计算损失
        self.loss.append(loss)                 # 保存损失值
        if self.tol is not None:
            if np.abs(loss_old - loss) < self.tol:  # 检查收敛
                break
        loss_old = loss                        # 更新旧损失
        grad = self._gradient(X, y, y_pred)    # 计算梯度
        self.W = self.W - self.lr * grad       # 更新权重
```

regularized least square $l(w) = \sum_{n=1}^{N}[t^{(n)} - y(x^{(n)}, w)]^2 + \alpha w^T w$

update $w \leftarrow w + \lambda\{\frac{1}{N}X^T(t - Xw) - \alpha w\}$

min-max normalization $x^* = \frac{x - x_{min}}{x_{max} - x_{min}}$

mean normalization $x^* = \frac{x - \mu}{\sigma}$

```
def random_split(x, y, test_size=0.2, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    indices = np.arange(len(x))
    np.random.shuffle(indices)

    split_index = int(len(x) * (1 - test_size))
    train_indices, test_indices = indices[:split_index], indices[split_index:]

    x_train, y_train = x[train_indices], y[train_indices]
    x_test, y_test = x[test_indices], y[test_indices]

    return x_train, y_train, x_test, y_test
```

```
def preprocess_data_X(self, X):
    # add bias term to X
    m, n = X.shape
    X_ = np.empty([m, n + 1])
    X_[:, 0] = 1
    X_[:, 1:] = X
    return X_
```

```
def predict(self, X):
    # predict y
    X = self.preprocess_data_X(X)
    return X @ self.W
```

```
def calculate_loss(self, y_true, y_pred):
    # MSE loss
    weights = np.where(y_true == 0, 0.4, 0.6)
    loss = weights * (y_true - y_pred)**2
    return np.mean(loss)
```

```
def gradient(self, X, y, y_pred):
    # gradient
    X = self.preprocess_data_X(X)
    weights = np.where(y == 0, 0.4, 0.6)
    grad = (X.T @ ((weights * (y_pred - y)).reshape(-1, 1))).flatten() / y.size
    return grad
```

Lec4 Classification

output: 1/-1, decision rule: $y = \text{sign}(w^T\begin{bmatrix}1\\x\end{bmatrix})$

loss function: -y_pred*y if y_pred*y <0 else 0
gradient: -y*x if y_pred*y <=0 else 0
non perfect: model too simple, noises in inputs, features too simple, mis-labellings

accuracy $A = \frac{TP+TN}{TP+TN+FP+FN}$ recall $R = \frac{TP}{TP+FN}$ Precision $P = \frac{TP}{TP+FP}$ f1 score $F1 = 2\frac{P \times R}{P+R}$

```
def evaluate_model(model, X_test, Y_test):
    predictions = model.predict(X_test)
    predictions = [1 if x >= 0.5 else 0 for x in predictions]
    TP = sum((predictions[i] == 1) and (Y_test[i] == 1) for i in range(len(Y_test)))
    TN = sum((predictions[i] == 0) and (Y_test[i] == 0) for i in range(len(Y_test)))
    FP = sum((predictions[i] == 1) and (Y_test[i] == 0) for i in range(len(Y_test)))
    FN = sum((predictions[i] == 0) and (Y_test[i] == 1) for i in range(len(Y_test)))
    print("TP:", TP, " TN:", TN, " FP:", FP, " FN:", FN)
    accuracy = (TP + TN) / (TP + TN + FP + FN)
    precision = TP / (TP + FP) if (TP + FP)!= 0 else 0
    recall = TP / (TP + FN) if (TP + FN)!= 0 else 0
    f1_score = 2 * precision * recall / (precision + recall) if (precision + recall)!= 0 else 0
    print("Accuracy:", accuracy, "Precision:", precision, "Recall:", recall, "F1 Score:", f1_score)
```

```
def mbgd_update(self, X, y, X_test, y_test):
    # Mini-batch gradient descent
    n_samples = X.shape[0]
    self.W = np.random.normal(0, 0.1, X.shape[1])

    for epoch in range(self.n_iter):
        # Shuffle data
        indices = np.random.permutation(n_samples)
        X_shuffled, y_shuffled = X[indices], y[indices]

        for start in range(0, n_samples, self.batch_size):
            # Mini-batch data
            end = start + self.batch_size
            X_batch, y_batch = X_shuffled[start:end], y_shuffled[start:end]
            # Compute predictions and gradients
            y_pred = X_batch @ self.W
            grad = self._gradient_batch(X_batch, y_batch, y_pred)
            # Update weights
            self.W -= self.lr * grad
        self.lr *= 0.999

        # Calculate loss for the epoch
        train_loss = self._loss_batch(y, X @ self.W)
        test_loss = self._loss_batch(y_test, X_test @ self.W)
        self.loss.append(train_loss)
        self.test_loss.append(test_loss)
```

```
def _loss_batch(self, y, y_pred):
    # Weighted hinge loss for a batch with L2 regularization
    weights = np.where(y == 1, self.positive_weight, 1 - self.positive_weight)
    hinge_loss = np.maximum(0, -y * y_pred) * weights
    reg_loss = self.alpha * np.sum(self.W[1:] ** 2)  # Exclude bias term from regularization
    return hinge_loss.mean() + reg_loss
```

```
def _gradient_batch(self, X, y, y_pred):
    # Gradient of weighted hinge loss for a batch with L2 regularization
    weights = np.where(y == 1, self.positive_weight, 1 - self.positive_weight)
    misclassified = y_pred * y < 0
    gradient = -(X[misclassified].T @ (weights[misclassified] * y[misclassified])) / X.shape[0]
    gradient[1:] += 2 * self.alpha * self.W[1:]  # Apply L2 regularization (exclude bias)
    return gradient
```

Lec 4 Logistic Regression

output: 0/1, decision rule: sigmoid $\sigma(z) = \frac{1}{1+\exp(-z)}$

model $y = \sigma(w^T x)$ $p(C=1|x) = \frac{1}{1+\exp(-w^T x)}$, $p(C=0|x) = \frac{\exp(-w^T x)}{1+\exp(-w^T x)}$

maximizing the likelihood $(\max\_w) L(w) = \prod_{i=1}^{N}(p(C=1|x^{(i)}))^{t^{(i)}}(1 - p(C=1|x^{(i)}))^{1-t^{(i)}}$

loss function $l_{\log}(w) = -\sum_{i=1}^{N} t^{(i)} \log y(x^{(i)}, w) - \sum_{i=1}^{N}(1 - t^{(i)})(1 - \log y(x^{(i)}, w))$

gradient descent $w \leftarrow w - \lambda\nabla l(w)$ $\nabla l(w) = -X^T(t - y)$

regularization $(\min\_w) l(w) = -\log(p(w)\prod_i p(t^{(i)}|x^{(i)}; w))\, p(w) = \prod_{j=1}^{d+1}\frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{w_j^2}{2\sigma^2})$ $\nabla l(w) += \alpha w$

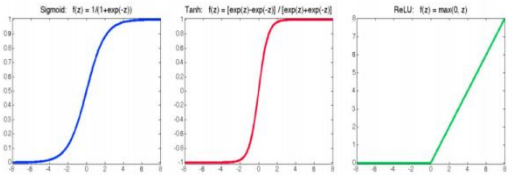
validation set: for tuning hyper-parameters
cross validation: leave-p-out, requires C_N^p for a set of N examples
k-fold cross-validation: k-1 sub samples training data
easily extended to multiple classes, natural probabilistic view of predictions, quick to train, fast at classification, good for simple data, resistant to over-fitting, can interpret. linear decision boundary.

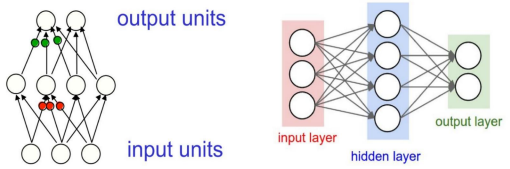
```
def _loss(y, y_pred, epsilon=1e-5):
    # Weighted cross entropy loss
    weights = np.where(y == 1, 0.5, 0.5)
    loss = -weights * (y * np.log(y_pred + epsilon) + (1 - y) * np.log(1 - y_pred + epsilon))
    return np.mean(loss)
```

```
def _gradient(self, X, y, y_pred):
    # Weighted gradient for cross entropy loss
    weights = np.where(y == 1, 0.6, 0.5)
    reg_term = self.alpha * self.W  # Regularization term
    weighted_diff = weights * (y_pred - y)
    return (weighted_diff @ X) / y.size + reg_term
```

Lec5 MLP

name	function	derivative
Sigmoid	$\sigma(z) = \frac{1}{1+\exp(-z)}$	$\sigma(z)\cdot(1-\sigma(z))$
Tanh	$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$	$1/\cosh^2(z)$
ReLU	$\text{ReLU}(z) = \max(0, z)$	$\begin{cases}1, & \text{if } z > 0\\ 0, & \text{if } z \le 0\end{cases}$





> Two different visualizations of a 2-layer neural network. In this example: 3 input units, 4 hidden units and 2 output units

N-layer neural network has N-1 layers of hidden units, one output layer
forward pass: performs inference

hidden layer $h_j(x) = f(v_{j0} + \sum_{i=1}^{D} x_i v_{ji})$ output layer $o_k(x) = g(w_{k0} + \sum_{j=1}^{J} h_j(x) w_{kj})$

```
def forward(self, inputs):
    # Forward pass with Dropout for hidden layers
    self.z_values = []
    self.activations = [inputs]
    for i in range(self.num_layers - 2):
        z = np.dot(self.activations[-1], self.weights[i]) + self.biases[i]
        self.z_values.append(z)
        activation = self.activation(z)
        activation = self.dropout(activation)  # Apply dropout
        self.activations.append(activation)
    # Output layer with Softmax
    z = np.dot(self.activations[-1], self.weights[-1]) + self.biases[-1]
    self.z_values.append(z)
    activation = self.softmax(z)
    self.activations.append(activation)
    return self.activations[-1]
```

backward pass: performs learning

Assuming the error function is mean-squared error (MSE), on a single training example n, we have

$$\frac{\partial E}{\partial o_k^{(n)}} = o_k^{(n)} - t_k^{(n)} := \delta_k^o$$

- The gradient descent update rule is given by:

$$w_{ki} \leftarrow w_{ki} - \eta\frac{\partial E}{\partial w_{ki}} = w_{ki} - \eta\sum_{n=1}^{N}(o_k^{(n)} - t_k^{(n)})o_k^{(n)}(1 - o_k^{(n)})x_i^{(n)}$$

mean squared loss

$$E(w) = \frac{1}{2}\sum_n (t^{(n)} - o^{(n)})^2$$

$$\delta_k^o = \frac{\partial E}{\partial o_k} = (o_k - t_k)$$

$$\delta_j^h = \sum_k \delta_k^o \cdot w_{kj}$$

$$\frac{\partial E}{\partial w_{kj}} = \delta_k^o \cdot h_j$$

$$\frac{\partial E}{\partial v_{ji}} = \delta_j^h \cdot x_i$$

$$w_{kj} = w_{kj} - \eta\frac{\partial E}{\partial w_{kj}}$$

$$v_{ji} = v_{ji} - \eta\frac{\partial E}{\partial v_{ji}}$$

cross entropy loss

$$E(w) = -\left[t^{(n)}\log o^{(n)} + (1 - t^{(n)})\log(1 - o^{(n)})\right]$$

$$\delta_k^o = \frac{\partial E}{\partial o_k} = \frac{o_k - t_k}{o_k(1 - o_k)}$$

$$\delta_k^z = \frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial o_k}\cdot\frac{\partial o_k}{\partial z_k} = \delta_k^o \cdot o_k(1 - o_k)$$

$$\delta_j^h = \frac{\partial E}{\partial h_j} = \frac{\partial E}{\partial o_k}\cdot\frac{\partial o_k}{\partial z_k}\cdot\frac{\partial z_k}{\partial h_j} = \sum_k \delta_k^z \cdot w_{kj}$$

$$\delta_j^u = \frac{\partial E}{\partial u_j} = \frac{\partial E}{\partial o_k}\cdot\frac{\partial o_k}{\partial z_k}\cdot\frac{\partial z_k}{\partial h_j}\cdot\frac{\partial h_j}{\partial u_j} = \delta_j^h \cdot h_j(1 - h_j)$$

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial o_k}\cdot\frac{\partial o_k}{\partial z_k}\cdot\frac{\partial z_k}{\partial w_{kj}} = \delta_k^z \cdot h_j$$

$$\frac{\partial E}{\partial v_{ji}} = \frac{\partial E}{\partial o_k}\cdot\frac{\partial o_k}{\partial z_k}\cdot\frac{\partial z_k}{\partial h_j}\cdot\frac{\partial h_j}{\partial u_j}\cdot\frac{\partial u_j}{\partial v_{ji}} = \delta_j^u \cdot x_i$$

```
def backward(self, inputs, targets, epoch):
    m = inputs.shape[0]
    predictions = self.activations[-1]
    delta = predictions - targets

    # Update output layer
    grad_w = np.dot(self.activations[-2].T, delta) / m + 2 * self.l2_lambda * self.weights[-1]
    grad_b = np.sum(delta, axis=0, keepdims=True) / m
    self.update_params(-1, grad_w, grad_b, epoch)

    # Backpropagate through hidden layers
    for i in range(self.num_layers - 2, 0, -1):
        delta = np.dot(delta, self.weights[i].T) * self.activation_derivative(self.z_values[i - 1])
        grad_w = np.dot(self.activations[i - 1].T, delta) / m + 2 * self.l2_lambda * self.weights[i - 1]
        grad_b = np.sum(delta, axis=0, keepdims=True) / m
        self.update_params(i - 1, grad_w, grad_b, epoch)
```

```
def backward(self, inputs, targets, learning_rate):
    # Backpropagation process
    # Calculate the error at the output layer
    output_errors = targets - self.output_layer
    output_delta = output_errors * self.sigmoid_derivative(self.output_layer)

    # Calculate the error for the hidden layer
    hidden_errors = output_delta.dot(self.weights2.T)
    hidden_delta = hidden_errors * self.sigmoid_derivative(self.hidden_layer)

    # Update the weights and biases using the calculated deltas
    self.weights2 += self.hidden_layer.T.dot(output_delta) * learning_rate
    self.bias2 += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
    self.weights1 += inputs.T.dot(hidden_delta) * learning_rate
    self.bias1 += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
```

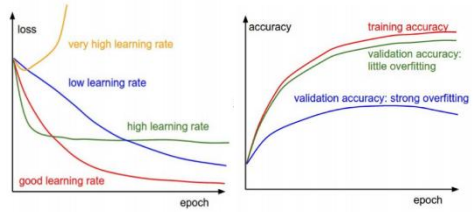
```
Loop until convergence
    for each example n:
      Given input x(n), propagate activity forward (x(n) → h(n) → o(n))(forward pass)
      Propagate gradients backward (backward pass)
      Update each weight (via gradient descent)
```

$$\delta_{\text{output}} = E \cdot \sigma'(z_{\text{output}}) \quad E_{\text{hidden}} = \delta_{\text{output}} \cdot W_2^T$$

$$\delta_{\text{hidden}} = E_{\text{hidden}} \cdot \sigma'(z_{\text{hidden}}) \quad W_2 \leftarrow W_2 + a_{\text{hidden}}^T \cdot \delta_{\text{output}} \cdot \eta$$

$$b_2 \leftarrow b_2 + \sum(\delta_{\text{output}}) \cdot \eta$$

add momentum:
$w_{ki} \leftarrow w_{ki} - v$ and $v \leftarrow \gamma v + \eta\frac{\partial E}{\partial w_{ki}}$



prevent overfitting: model have right capacity (enough to true regularities, not enough to spurious regularities), limit number of hidden units, limit norm of weights, early stopping
weight-decay, keep weights small unless they have big error derivatives, $C = E + \frac{\lambda}{2}\sum_i w_i^2$, $\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$
separate validation set to decide which regularizer to use and how strong to make it
early stopping: start with small, grow until validation worse, capacity is

limited cause no time to grow big

Lec6 knn

non-parametric models: learning amounts to simply storing training data; test instances classified using similar training instances; embodies often have assumptions (output varies smoothly with input, data occupies sub-space of high-dimensional input space)

Euclidean distances $\|\mathbf{x}^{(a)} - \mathbf{x}^{(b)}\|_2 = \sqrt{\sum_{j=1}^{d} \left(x_j^{(a)} - x_j^{(b)}\right)^2}$

large k, better performance, but may end up looking not neighbors, rule of thumb is $k < \sqrt{n}$; normalize is important

e Hamming distance $d(1101\ 1001, 1001\ 1101) = 2$

complexity: O(kdN), use subset, pre-sort (kd-trees), compute only approximate distance, remove redundant data (condensing)

kdtree: k-dimensional tree, similar to binary search tree, for efficiently solvng multi-dimensional space search problems

1. choose a dimension; 2. sort along that dimension; 3.select the median as partition point, divide; 4. Recursive

```
class KNN:
    解释代码 | 注释代码 | 生成单测 | ×
    def __init__(self, k=3):
        self.k = k
        self.kdtree = None

    解释代码 | 注释代码 | 生成单测 | ×
    def fit(self, X, y):
        self.X_train = X
        self.y_train = y
        self.kdtree = KDTree(X)

    解释代码 | 注释代码 | 生成单测 | ×
    def predict_multiple(self, X):
        predictions = [self.predict_single(x) for x in X]
        return np.array(predictions)

    解释代码 | 注释代码 | 生成单测 | ×
    def predict_single(self, x):
        dist, idx = self.kdtree.query(x, k=self.k, p=2)
        neighbors_labels = [self.y_train[i] for i in idx[0]]
        prediction = max(set(neighbors_labels), key=neighbors_labels.count)
        return prediction
```

Lec7 Decision Tree

pick an attribute, condition on a choice, assign class with majority vote

greedy heuristic, use information theory to find best attribute

minimum entropy $H(X) = -\sum_{x\in X} p(x)\log_2 p(x)$越小越好

conditional entropy $H(Y|X=x) = -\sum_{y\in Y} p(y|x)\log_2 p(y|x)$

expected conditional entropy $H(Y|X) = -\sum_{x\in X}\sum_{y\in Y} p(x,y)\log_2 p(y|x)$

chain rule $H(X,Y) = H(X|Y) + H(Y)$ 必然存在 $H(Y|X) \le H(Y)$

information gain $IG(Y|X) = H(Y) - H(Y|X)$越大越好

$IG(Y|X) = 0$ means X uninformative about Y

$IG(Y|X) = H(Y)$ means X informative about Y

```
def _entropy(self, y):
    # Compute entropy of output -sum(p(Y=y)log2(p(Y=y))), which is a scalar
    count_y = np.bincount(y) # Count the number of each output label
    prob_y = count_y[np.nonzero(count_y)] / y.size # Compute the probability of each output label
    entropy_y = -np.sum(prob_y * np.log2(prob_y)) # Compute the entropy of output
    return entropy_y
def _conditional_entropy(self, feature, y):
    # Compute the conditional entropy according to the formula H(Y|feature) = Sum_{feature_value} p
    # The argument feature represents the input data vector of one specific feature
    feature_values = np.unique(feature)
    h = 0.
    for v in feature_values:
        y_sub = y[feature == v]
        prob_y_sub = y_sub.size / y.size
        h += prob_y_sub * self._entropy(y_sub) # Compute the conditional entropy of feature_value

    return h
def _information_gain(self, feature, y):
    # Compute the information gain according to the formula IG(feature) = H(Y) - H(Y|feature)
    ig_feature = self._entropy(y) - self._conditional_entropy(feature, y)
    return ig_feature
def _select_feature(self, X, y, features_list):
    # Select the feature with the largest information gain
    if features_list:
        gains = np.apply_along_axis(self._information_gain, 0, X[:, features_list], y)
        index = np.argmax(gains)
        if gains[index] > self.gain_threshold:
            return index
def _predict_one(self, x):
    node = self.tree_
    while node.children:
        child = node.children.get(x[node.feature_index])
        if not child:
            break
        node = child
    return node.value
def predict(self, X):
    return np.apply_along_axis(self._predict_one, axis=1, arr=X)
```

```
def _build_tree(self, X, y, features_list):
    # Build a decision tree recuresively.
    # The default output should be the label with the maximum counting
    node = DecisionTree.Node()
    labels_count = np.bincount(y)
    node.value = np.argmax(np.bincount(y))
    # Check whether the labels are the same
    if np.count_nonzero(labels_count) !=1:
        # Select the feature with the largest information gain
        index = self._select_feature(X, y, features_list)
        if index is not None:
            # Remove this feature from the features list
            node.feature_index = features_list.pop(index)
            # Divide the training set according to this selected feature
            # Then use the subset of training examples in each branch to create a sub-tree
            feature_values = np.unique(X[:, node.feature_index])
            for v in feature_values:
                # Obtain the subset of training examples
                idx = X[:, node.feature_index] == v
                X_sub, y_sub = X[idx], y[idx]
                # Build a sub-tree
                node.children[v] = self._build_tree(X_sub, y_sub, features_list.copy())
    return node
```

Lec8 multi-class

1 vs all / 1 vs 1

decision boundary always singly connected and convex

1-of-k encoding: $t = [0\ 1\ 0\ 0]^T$ means label 2

softmax function $\frac{exp(z_k)}{\sum_j exp(z_j)}$

multi-class logistic $E(W) = -\sum_{k=1}^{K} t_k \log[y_k(x)]$ and derivative $\frac{\partial E}{\partial y_k} = -\frac{t_k}{y_k}$

so $\frac{\partial y_k}{\partial z_m} = \delta(k,m)y_k - y_k y_m$

gradient of batch $\nabla E(W) = -X^T(T-Y)$

Lec9 K-means

unsupervised learning: dimensionality reduction, cluster, density estimation

kmeans: initialize, assign, refit

$$\min_{\{m\},\{r\}} J(\{m\},\{r\}) = \min_{\{m\},\{r\}} \sum_{n=1}^{N}\sum_{k=1}^{K} r_k^{(n)} \|m_k - x^{(n)}\|^2$$

$$s.t. \sum_k r_k^{(n)} = 1, \forall n, \text{where } r_k^{(n)} \in \{0,1\}, \forall k, n$$

kmeans++: improve initial, select next center farthest

softkmeans: degree of assignment to each cluster mean based on responsibilities

```
f initialize_centroids(self, X):
  """Initialize centroids using KMeans++."""
  self.centroids = np.array([X[np.random.choice(X.shape[0])]])  # Choose one random point as the first centroid

  for _ in range(1, self.k):
      # Compute distance of each point to the nearest centroid
      distances = np.array([np.min([np.linalg.norm(x - centroid) ** 2 for centroid in self.centroids]) for x in
      probabilities = distances / distances.sum()  # Normalize to create a probability distribution
      cumulative_probabilities = np.cumsum(probabilities)
      r = np.random.rand()
      # Select new centroid based on probability distribution
      for i, p in enumerate(cumulative_probabilities):
          if r < p:
              self.centroids = np.vstack((self.centroids, X[i]))
              break
def predict(self, X):
    """Predict the closest cluster for each data point."""
    distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
    return np.argmin(distances, axis=1)
def fit(self, X, y):
    """Fit the model to the data."""
    self.initialize_centroids(X)
    for n in range(self.max_iters):
        # Assign each point to the nearest centroid
        closest_centroids = np.argmin(np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2), axis=1)
        # Update centroids (with learning rate mechanism)
        new_centroids = np.array([X[closest_centroids == i].mean(axis=0)
                                  if X[closest_centroids == i].size else X[np.random.choice(X.shape[0])] for i in range(self.k)])
        # Apply learning rate to smooth centroid update
        self.centroids = self.centroids * (1 - self.learning_rate) + new_centroids * self.learning_rate
        # Compute inertia (sum of squared distances to centroids)
        inertia = np.sum([np.sum(np.linalg.norm(X[closest_centroids == i] - self.centroids[i], axis=1) ** 2)
                          for i in range(self.k)])
        self.inertia_history.append(inertia)
        # Compute centroid movement (Euclidean distance between old and new centroids)
        centroid_movement = np.sum(np.linalg.norm(new_centroids - self.centroids, axis=1))
        self.centroid_movement_history.append(centroid_movement)
        # If centroid movement is very small, stop the algorithm
        if centroid_movement < 1e-6:
            print(f"Converged at iteration {n}")
            break
    # Map the predicted labels to true labels using the training set
    train_predictions = self.predict(X)
    self.label_mapping = self.map_labels(y, train_predictions)
def update_membership(self, X):
    """Update the membership values for each point"""
    distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
    distances = np.maximum(distances, self.epsilon)  # Avoid division by zero
    membership = 1 / (distances ** 2)
    return membership / np.sum(membership, axis=1, keepdims=True)  # Normalize the membership
def update_centroids(self, X):
    """Update centroids based on membership and a learning rate"""
    alpha = 0.5  # Learning rate
    new_centroids = np.zeros((self.k, X.shape[1]))
    for k in range(self.k):
        weight = self.membership[:, k] ** self.m
        updated_centroid = np.sum(weight[:, np.newaxis] * X, axis=0) / np.sum(weight)
        # Apply learning rate
        new_centroids[k] = alpha * updated_centroid + (1 - alpha) * self.centroids[k]
    return new_centroids
```

```
def calculate_inertia(self, X, centroids):
    """Calculate the inertia (sum of squared distances to the nearest centroid)"""
    inertia = np.sum([np.sum(np.linalg.norm(X[self.membership[:, k] > 0.5] - centroids[k], axis=1) ** 2)
                      for k in range(self.k)])
    return inertia
```

Lec10 PCA

linearly project to low dimensional $x \approx U_{pca} z + a$, U_{pca} is D*M matrix, z is M-dimensional vector

empirical covariance matrix $C = \frac{1}{N}\sum_{n=1}^{N}(x^{(n)} - x)(x^{(n)} - x)^T = U\Sigma U^T$

principal components $z = U_{1:M}^T x$

minimize reconstruction error $J(u,z,b) = \sum_n \|x^{(n)} - \tilde{x}^{(n)}\|^2$

where $\tilde{x}^{(n)} = \sum_{j=1}^{M} z_j^{(n)} u_j + \sum_{j=M+1}^{D} b_j u_j$ is reconstruction sample

minimized when $x^{(n)} \approx \tilde{x}^{(n)} = U_{1:M} z^{(n)} + a, a = U_{M+1:D} b, b = U_{M+1:D}^T x$

```
def fit(self, X):
    # Compute covariance matrix and eigenvalues/eigenvectors
    cov_matrix = np.cov(X.T)
    eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
    # Sort eigenvalues and eigenvectors in descending order
    eigen_pairs = [(np.abs(eigenvalues[i]), eigenvectors[:, i]) for i in range(len(eigenvalues))]
    eigen_pairs.sort(key=lambda k: k[0], reverse=True)
    # Select the top n_components eigenvectors
    self.eigenvalues = eigenvalues
    self.eigenvectors = eigenvectors
    self.projection_matrix = np.hstack([eigen_pairs[i][1][:, np.newaxis] for i in range(self.n_componen
```

Lec10 Auto-encoder

define $z = f(Wx)$ and $\hat{x} = g(Vz)$, and goal is $\min(W,V)\frac{1}{2N}\sum_{n=1}^{N}\|x^{(n)} - \hat{x}^{(n)}\|^2$

```
def forward(self, X):
    A = X
    activations = [A]
    pre_activations = []
    # Forward pass through hidden layers
    for W, b in zip(self.weights[:-1], self.biases[:-1]):
        Z = np.dot(A, W) + b
        A = self.relu(Z)
        pre_activations.append(Z)
        activations.append(A)
    # Decoder output
    Z = np.dot(A, self.weights[-1]) + self.biases[-1]
    activations.append(Z)
    return activations
def backward(self, X, activations):
    m = X.shape[0]  # Number of training examples
    # Backpropagate the error
    dZ = activations[-1] - X  # Derivative of loss with respect to output
    dW = np.dot(activations[-2].T, dZ) / m
    db = np.sum(dZ, axis=0, keepdims=True) / m
    # Update decoder weights
    self.weights[-1] -= self.learning_rate * dW
    self.biases[-1] -= self.learning_rate * db
    # Backpropagate to hidden layers
    for i in range(len(self.hidden_dims)-1, -1, -1):
        dA = np.dot(dZ, self.weights[i+1].T)
        dZ = dA * self.relu_derivative(activations[i+1])  # Derivative of ReLU
        dW = np.dot(activations[i].T, dZ) / m
        db = np.sum(dZ, axis=0, keepdims=True) / m
        # Update hidden layer weights
        self.weights[i] -= self.learning_rate * dW
        self.biases[i] -= self.learning_rate * db
```

Lec11 SVM

supervised learning, binary classification

max-margin classification $(w^T x + b)y \ge 1$ then $\lambda = \frac{2}{w^T w}$

$$J(w,b;\alpha) = \frac{1}{2}\|w\|^2 + \sum_{i=1}^{N}\max_{\alpha_i \ge 0}\alpha_i[1 - (w^T x^{(i)} + b)t^{(i)}]$$

$$\max_{\alpha_i \ge 0}\min_{w,b} J(w,b;\alpha) \le \min_{w,b}\max_{\alpha_i \ge 0} J(w,b;\alpha)$$

$$\frac{\partial J(w,b;\alpha)}{\partial w} = w - \sum_{i=1}^{N}\alpha_i x^{(i)} t^{(i)} = 0 \quad \frac{\partial J(w,b;\alpha)}{\partial b} = -\sum_{i=1}^{N}\alpha_i t^{(i)} = 0$$

final optimization $L = \max_{\alpha_i \ge 0}\{\sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{N} t^{(i)} t^{(j)} \alpha_i \alpha_j (x^{(i)^T} x^{(j)})\}$

只有边界样本 α 非零

Prediction on a new example:.

$$y = \text{sign}\left[b + x \cdot \left(\sum_{i=1}^{N}\alpha_i t^{(i)} x^{(i)}\right)\right] = \text{sign}\left[b + x \cdot \left(\sum_{i\in S}\alpha_i t^{(i)} x^{(i)}\right)\right]$$

optimal solution, for nonzero α_i^*, we have $1 - (w^{*T} x^{(i)} + b^*)t^{(i)} = 0$

so $b^* = t^{(i)} - \sum_{j=1}^{N}\alpha_j^* t^{(j)} (x^{(j)} \cdot x^{(i)})$

data not linearly separable, slack variables

introduce slack variables ξ_i, $\min_{w,b,\xi}\frac{1}{2}\|w\|^2 + \lambda\sum_{i=1}^{N}\xi_i$

subject to $\xi_i \ge 0;\ t^{(i)}(w^T x^{(i)} + b) \ge 1 - \xi_i, \forall i = 1,\ldots,N$

- Define:

$$J(w,b,\xi;\alpha,\mu) = \frac{1}{2}\|w\|^2 + \lambda\sum_{i=1}^{N}\xi_i + \sum_{i=1}^{N}\alpha_i\left[1 - \xi_i - t^{(i)}(w^T x^{(i)} + b)\right] + \sum_{i=1}^{N}\mu_i(-\xi_i)$$

- Then the problem becomes:

$$\max_{\alpha_i \ge 0,\mu_i \ge 0}\min_{w,b,\xi} J(w,b,\xi;\alpha,\mu)$$

- Consdier the KKT conditions:

$$\frac{\partial J}{\partial w} = w - \sum_{i=1}^{N}\alpha_i x^{(i)} t^{(i)} = 0$$

$$\frac{\partial J}{\partial b} = -\sum_{i=1}^{N}\alpha_i t^{(i)} = 0$$

$$\frac{\partial J}{\partial \xi_i} = \lambda - \alpha_i - \mu_i = 0, \quad \forall i = 1,\ldots,N$$

$$max\{\sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{N} t^{(i)} t^{(j)} \alpha_i \alpha_j (x^{(i)} \cdot x^{(j)})\} \text{ subject to } 0 \le \alpha_i \le \lambda, \forall i = 1,\ldots N, \sum_{i=1}^{N}\alpha_i t^{(i)} = 0$$

1. $\alpha_i = 0 \iff t^{(i)}(w^T x^{(i)} + b) \ge 1$ (sample i is on the correct side with $\xi_i = 0$)
2. $0 < \alpha_i < \lambda \iff t^{(i)}(w^T x^{(i)} + b) = 1$ (sample i is a support vector)
3. $\alpha_i = \lambda \iff t^{(i)}(w^T x^{(i)} + b) \le 1$ (sample i is on the wrong side with $\xi_i \ne 0$)

SMO algorithm: sequential minimal optimization

select two alpha at a time, treating others as constants

kernel trick: $K(x1,x2) = \phi(x1)\phi(x2)$, gaussian $\exp(-\frac{\|x1-x2\|^2}{2\sigma^2})$, sigmoid $\tanh(\beta(x1^T x2) + a)$

```
def fit(self, X, y):
    # Data normalization
    X = self.normalize_data(X)
    self.X = X
    self.y = y.reshape(-1, 1).astype(np.double)
    N = X.shape[0]
    # Compute the kernel matrix K (NxN)
    self.K = self.kernel(X, X, self.k)
    # Initialize Lagrange multipliers alpha
    alphas = np.zeros((N, 1))
    for n in range(self.n_iteration):
        decision_values = self.K @ (self.y * alphas)
        gradient = np.ones((N, 1)) - decision_values
        alphas += self.lr * (self.y * gradient)
        alphas = np.clip(alphas, 0, self.C)
        # Loss calculation (hinge loss + regularization)
        loss = np.mean(np.maximum(0, 1 - self.y * decision_values)) + 0.5 * self.C * np.sum(alphas ** 2)
    # Save Lagrange multipliers
    self.alphas = alphas
    self.support_vectors_mask = (self.alphas > 1e-3).flatten()  # Support vector mask
```

```
predict(self, X_test):
# Normalize test data
X_test = self.normalize_data(X_test)
# Get support vectors and their corresponding labels
support_vectors = self.X[self.support_vectors_mask]
support_labels = self.y[self.support_vectors_mask]
support_alphas = self.alphas[self.support_vectors_mask]
# Calculate bias term b
margin_support_vector_index = np.argmax((0 < support_alphas) & (support_alphas < self.C))
margin_support_vector = support_vectors[margin_support_vector_index, np.newaxis]
margin_label = support_labels[margin_support_vector_index]
bias = margin_label - np.sum(support_alphas * support_labels * self.kernel(support_vectors, margin_support_vector))
# Compute decision function (score for each test sample)
decision_scores = np.sum(support_alphas * support_labels * self.kernel(support_vectors, X_test), axis=0) + bias
# Return predicted labels (sign of the decision function)
y = np.sign(decision_scores).astype(int)
return y
```

Lec12 ensemble methods

minimize variance and bias

boosting: Justification prob of wrong = $\binom{N}{k}\epsilon^k(1-\epsilon)^{N-k}$

bagging: bootstrap aggregation, mean of individual estimates, keep train models on re-weighted data

minimize $J_m = \sum_{n=1}^{N} w_n^m [y_m(x^n) \ne t^n]$ unnormalized error rate $\epsilon_m = \frac{J_m}{\sum_n w_n^m}$ classifie quality $\alpha_m = \ln(\frac{1-\epsilon_m}{\epsilon_m})$ update data weights $w_n^{m+1} = w_n^m exp(-\frac{1}{2}t^{(n)}\alpha_m y_m(x^{(n)}))$ final model $y(x) = \text{sign}(\sum_{m=1}^{M}\alpha_m y_m(x))$

Boosting 关注于减少模型的偏差，通过迭代训练和调整样本权重来提高模型性能；而 Bagging 关注于减少模型的方差，通过构建多个独立的弱学习器并聚合结果来提高模型的稳定性。

```
def fit(self, X, y):
    n_samples, n_features = X.shape
    weights = np.ones(n_samples) / n_samples
    self.estimators = []
    for i in range(self.n_estimators):
        clf = Perceptron() # 选择一个弱分类器
        clf.fit(X, y, sample_weight=weights)
        self.estimators.append(clf)
        y_pred = clf.predict(X)
        error = np.sum(weights[y_pred != y]) # 计算错误率
        if error > 0.5: # 如果错误率大于0.5，那么这个分类器就没有用，可以停止
            break
        alpha = np.log((1.0 - error) / (error + 1e-10)) # 计算alpha值
        weights *= np.exp(-alpha * y * y_pred)
        weights /= weights.sum()
def predict(self, X):
    predictions = np.array([clf.predict(X) for clf in self.estimators]).T
    return np.sign(np.sum(predictions * np.array([est.alpha for est in self.estimators]), axis=1))
```