Advanced Artificial Intelligence

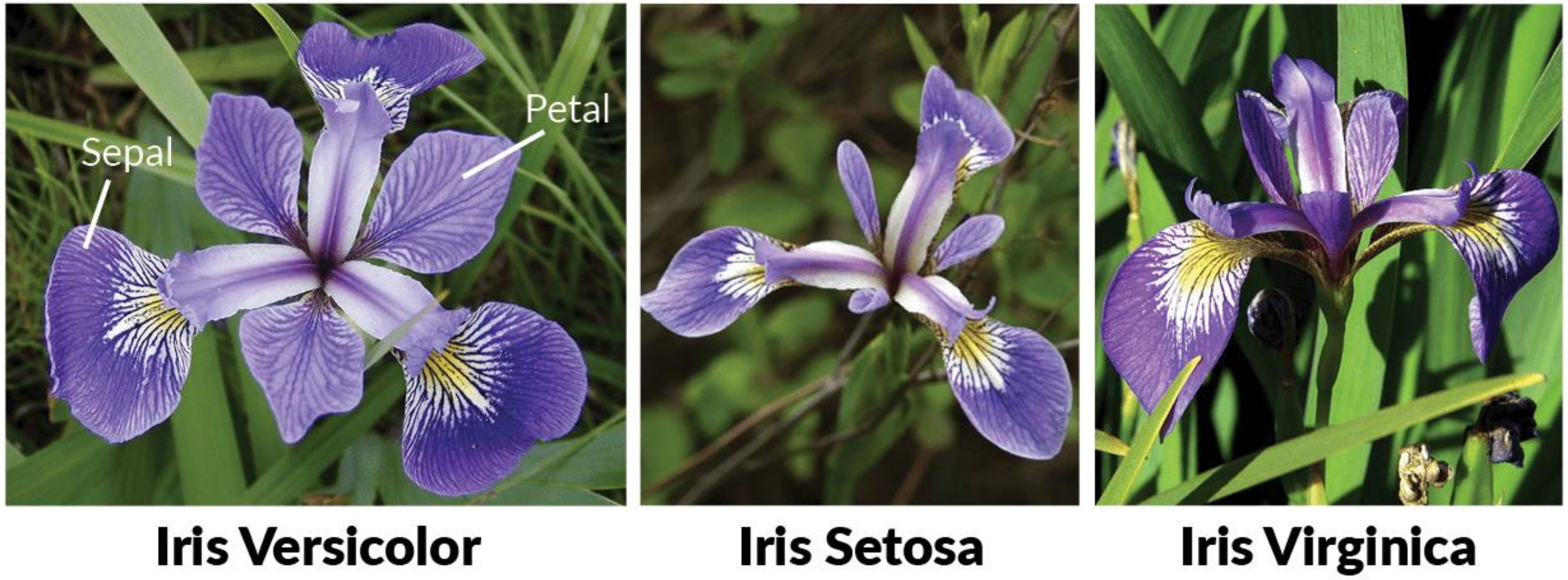
Lab 07

Outline

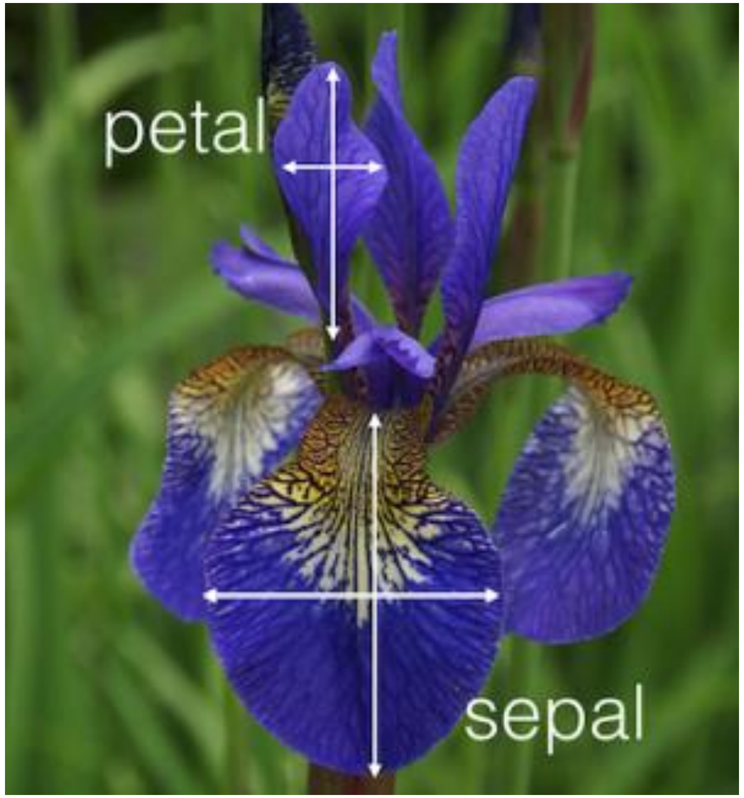
- A concrete problem
- Implementation of different supervised learning algorithms
- Exercise

A concrete problem

Could you classify iris flowers using different supervised learning algorithms?



Iris Dataset



Attributes:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm

Labels:
-- Iris Setosa
-- Iris Versicolour
-- Iris Virginica

Load and Split the dataset

```
from sklearn.datasets import load_iris
data = load_iris()
X = data.data
y = data.target
print(X.shape,y.shape)
print(X[:3])
print(y[:3])
```

```
(150, 4) (150,)
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]]
[0 0 0]
```

```
from sklearn.model_selection import train_test_split
X_train,X_test, y_train, y_test = train_test_split(X,y,
                                    test_size=0.4,random_state=3)

print(X_train.shape,y_train.shape)
print(X_test.shape,y_test.shape)
print(X_train[:3])
print(y_train[:3])
```

```
(90, 4) (90,)
(60, 4) (60,)
[[5.  3.6 1.4 0.2]
 [4.8 3.  1.4 0.1]
 [6.8 3.2 5.9 2.3]]
[0 0 2]
```

Implementation of different Algorithms

I. Linear Model

II. Decision Tree

III. Neural Network

IV. k-Nearest Neighbours

V. Support Vector Machine

Logistic Regression

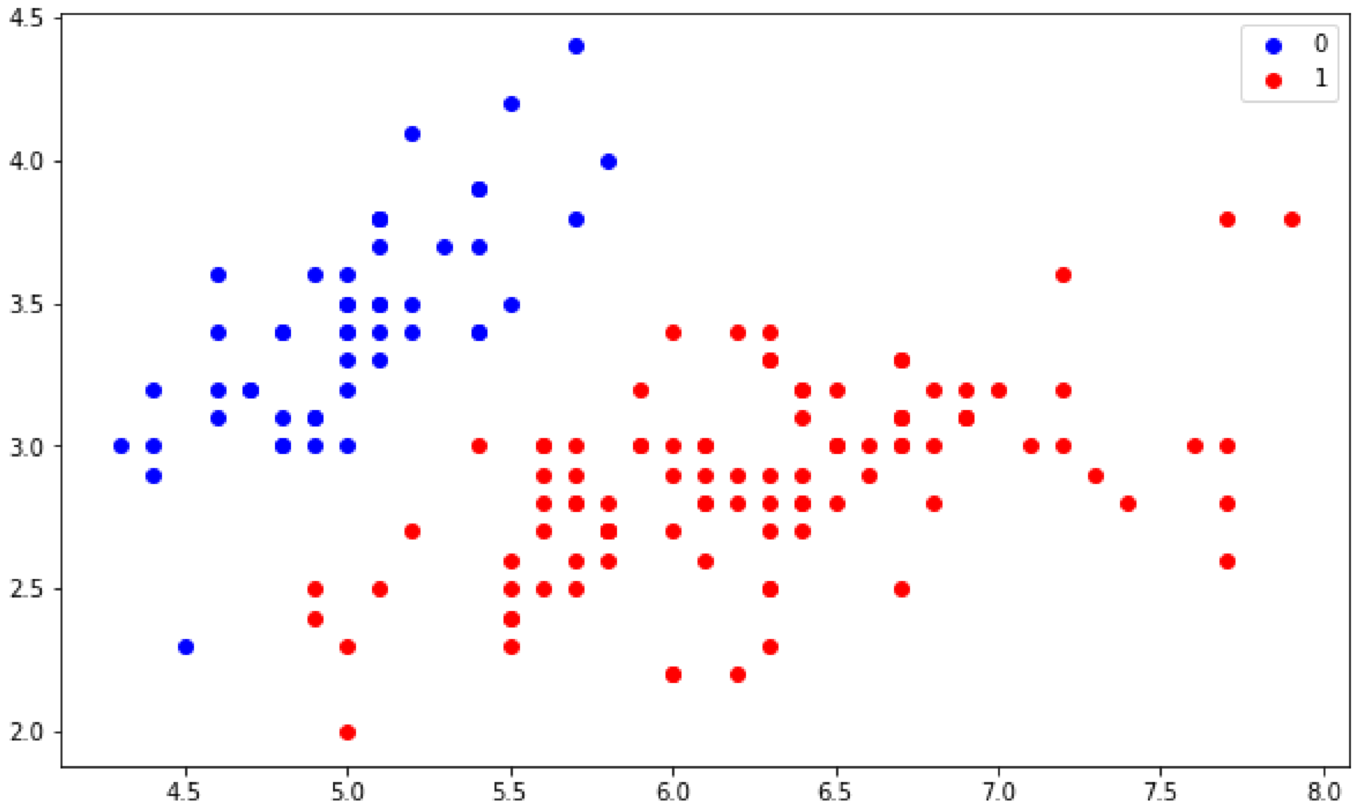
- Logistic regression model: $h_{\boldsymbol{w}}(\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x})$, where $\sigma(z) = \frac{1}{1+e^{-z}}$.

- Iterative solution: $w_i \leftarrow w_i - \alpha \frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \mathrm{w}_i}$
  - $\frac{\partial \mathcal{L}(\boldsymbol{w})}{\partial \omega_i} = -\sum_{n=1}^{N}\left[y^{(n)} - h_{\boldsymbol{w}}\left(\boldsymbol{x}^{(n)}\right)\right] \cdot h_{\boldsymbol{w}}\left(\boldsymbol{x}^{(n)}\right) \cdot \left[1 - h_{\boldsymbol{w}}\left(\boldsymbol{x}^{(n)}\right)\right] \cdot x_i^{(n)}$,
  - α: learning rate, positive.

Classification optimization:

$min_{\boldsymbol{w}}\, \mathcal{L}(\boldsymbol{w}) = \frac{1}{2N}\sum_{n=1}^{N}[y^{(n)} - h_{\boldsymbol{w}}(\boldsymbol{x}^{(n)})]^2$.

2-class classification




```
iris = load_iris()

X = iris.data
y = (iris.target != 0) * 1
```

Important functions

```python
def __loss(self, h, y):
    return (-y * np.log(h) - (1 - y) * np.log(1 - h)).mean()

def fit(self, X, y):
    if self.fit_intercept:
        X = self.__add_intercept(X)

    # weights initialization
    self.theta = np.zeros(X.shape[1])

    for i in range(self.num_iter):
        z = np.dot(X, self.theta)
        h = self.__sigmoid(z)
        gradient = np.dot(X.T, (h - y)) / y.size
        self.theta -= self.lr * gradient

        z = np.dot(X, self.theta)
        h = self.__sigmoid(z)
        loss = self.__loss(h, y)

        if(self.verbose ==True and i % 10000 == 0):
            print(f'loss: {loss} \t')
```

Logistic Regression

```
model = LogisticRegressionScratch(lr=0.1,num_iter=1000)

model.fit(X_train,y_train)

print(accuracy_score(model.predict(X_test), y_test))
```

```
1.0
```

Implementation of different Algorithms

I. Linear Model

II. **Decision Tree**

III. Neural Network

IV. k-Nearest Neighbours

V. Support Vector Machine

Entropy

- Entropy: $\mathcal{H}(Y) \triangleq -\sum_k p(y_k) \log_2 p(y_k)$.

- Larger entropy, more uncertainty.
  - High entropy: $Y \sim$ uniform or flat distribution → less predictable
  - Low entropy: $Y \sim$ peak/valley distribution → more predictable

```python
import numpy as np
from collections import Counter


def entropy(s):
    counts = np.bincount(s)
    percentages = counts / len(s)

    entropy = 0
    for pct in percentages:
        if pct > 0:
            entropy += pct * np.log2(pct)
    return -entropy
```

Information gain

- Conditional entropy:

$$\mathcal{H}(Y|X) \triangleq \sum_j p(X = x_j) \cdot \mathcal{H}(Y|X = x_j).$$

- Information gain: Decrease in entropy after splitting

$$IG(X) = \mathcal{H}(Y) - \mathcal{H}(Y|X)$$

  - X: input feature,
  - Y: classification label.

```
def information_gain(parent, left_child, right_child):
    num_left = len(left_child) / len(parent)
    num_right = len(right_child) / len(parent)

    gain = entropy(parent) - (num_left * entropy(left_child)
                + num_right * entropy(right_child))
    return gain
```

Best split

- Take the best t from $\{t\}$: Denote $X \sim Est$,
  - (1) Define $\mathcal{H}(Y|X{:}\,t) = p(X < t) \cdot \mathcal{H}(Y|X < t) + p(X \geq t) \cdot \mathcal{H}(Y|X \geq t)$;
  - (2) Compute $IG(Y|X{:}\,t_i) = \mathcal{H}(Y) - \mathcal{H}(Y|X{:}\,t_i)$ for $\forall t_i$;
  - (3) Choose $t^* = \arg\max_{t_i} IG(Y|X{:}\,t_i)$

- Use: $IG^*(Est) = IG(Y|X{:}\,t^*) = \max_{t_i} IG(Y|X{:}\,t_i)$.

```python
# For every dataset feature
for f_idx in range(n_cols):
    X_curr = X[:, f_idx]
    # For every unique value of that feature
    for threshold in np.unique(X_curr):
        # Construct a dataset and split it to the left and right parts
        df = np.concatenate((X, y.reshape(1, -1).T), axis=1)
        df_left = np.array([row for row in df if row[f_idx] <= threshold])
        df_right = np.array([row for row in df if row[f_idx] > threshold])

        # Do the calculation only if there's data in both subsets
        if len(df_left) > 0 and len(df_right) > 0:
            # Obtain the value of the target variable for subsets
            y = df[:, -1]
            y_left = df_left[:, -1]
            y_right = df_right[:, -1]

            # Caclulate the information gain and save the split parameters
            # if the current split if better then the previous best
            gain = self._information_gain(y, y_left, y_right)
            if gain > best_info_gain:
                best_split = {
                    'feature_index': f_idx,
                    'threshold': threshold,
                    'df_left': df_left,
                    'df_right': df_right,
                    'gain': gain
                }
                best_info_gain = gain
return best_split
```

Build the tree

Learning Decision Trees

- Start from empty tree.
- Split on next best feature based on *information gain*.
- Repeat

```python
# Check to see if a node should be leaf node
if n_rows >= self.min_samples_split and depth <= self.max_depth:
    # Get the best split
    best = self._best_split(X, y)
    # If the split isn't pure
    if best['gain'] > 0:
        # Build a tree on the left
        left = self._build(
            X=best['df_left'][:, :-1],
            y=best['df_left'][:, -1],
            depth=depth + 1
        )
        right = self._build(
            X=best['df_right'][:, :-1],
            y=best['df_right'][:, -1],
            depth=depth + 1
        )
        return Node(
            feature=best['feature_index'],
            threshold=best['threshold'],
            data_left=left,
            data_right=right,
            gain=best['gain']
        )
# Leaf node - value is the most common target value
return Node(
    value=Counter(y).most_common(1)[0][0]
)
```

Performance

```
from dt import DecisionTree

model = DecisionTree()
model.fit(X_train,y_train)
preds = model.predict(X_test)
print(accuracy_score(y_test, preds))
```

```
0.95
```

Implementation of different Algorithms

I. Linear Model

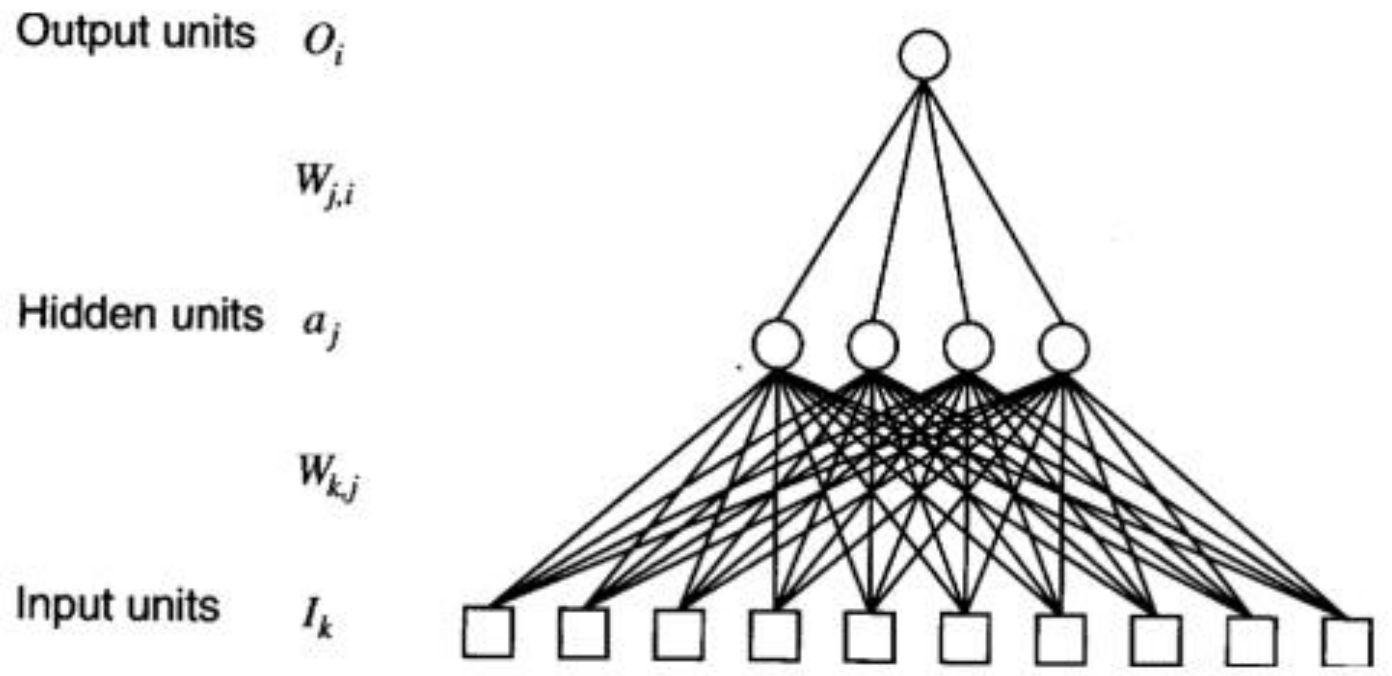
II. Decision Tree

III. **Neural Network**

IV. k-Nearest Neighbours

V. Support Vector Machine

Multilayer Neural Network



Loss Function

- Loss function for an example: $\ell_2(\boldsymbol{w}) = \frac{1}{2}||y - o(\boldsymbol{x})||^2$
  - $(\boldsymbol{x}, y)$: a training example;
  - $o(\boldsymbol{x})$: estimated output for inputs $\boldsymbol{x}$.
- Partial derivative for any w: 'chain rule'

$$\frac{\partial}{\partial \omega}\ell_2(\boldsymbol{w}) = \frac{\partial}{\partial \omega}\frac{1}{2}\sum_i (y_i - o_i)^2 = -\sum_i (y_i - o_i)\frac{\partial o_i}{\partial \omega} = \cdots$$

- Back propagation (反向传播/逆传播) to train ANN:
  - Gradient descent for $w_{j,i}$ from hidden to output: $\omega_{j,i} \leftarrow \omega_{j,i} - \alpha \cdot \frac{\partial}{\partial \omega_{j,i}}\ell_2(\boldsymbol{w})$
  - Gradient descent for $w_{k,j}$ from input to hidden: $\omega_{k,j} \leftarrow \omega_{k,j} - \alpha \cdot \frac{\partial}{\partial \omega_{k,j}}\ell_2(\boldsymbol{w})$

Configure Models

```python
import torch.nn.functional as F
import torch.nn as nn
from torch.autograd import Variable
class Model(nn.Module):
    def __init__(self, input_dim):
        super(Model, self).__init__()
        self.layer1 = nn.Linear(input_dim, 50)
        self.layer2 = nn.Linear(50, 50)
        self.layer3 = nn.Linear(50, 3)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = F.softmax(self.layer3(x), dim=1)
        return x

model     = Model(X_train.shape[1])
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
loss_fn   = nn.CrossEntropyLoss()
model
```

```
Model(
  (layer1): Linear(in_features=2, out_features=50, bias=True)
  (layer2): Linear(in_features=50, out_features=50, bias=True)
  (layer3): Linear(in_features=50, out_features=3, bias=True)
)
```

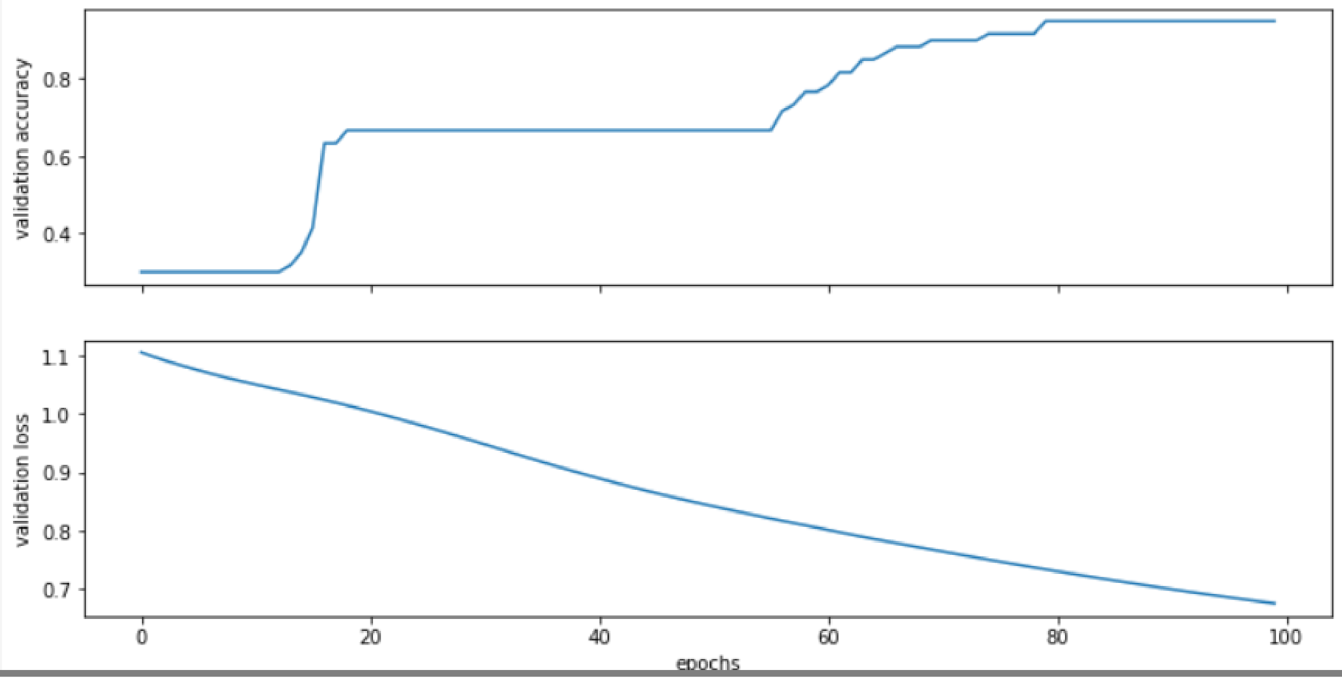
Train the Model

```python
import tqdm

EPOCHS  = 100
X_train = Variable(torch.from_numpy(X_train)).float()
y_train = Variable(torch.from_numpy(y_train)).long()
X_test  = Variable(torch.from_numpy(X_test)).float()
y_test  = Variable(torch.from_numpy(y_test)).long()

loss_list     = np.zeros((EPOCHS,))
accuracy_list = np.zeros((EPOCHS,))

for epoch in tqdm.trange(EPOCHS):
    y_pred = model(X_train)
    loss = loss_fn(y_pred, y_train)
    loss_list[epoch] = loss.item()

    # Zero gradients
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    with torch.no_grad():
        y_pred = model(X_test)
        correct = (torch.argmax(y_pred, dim=1) == y_test).type(torch.FloatTensor)
        accuracy_list[epoch] = correct.mean()
```

```
100%|██████████| 100/100 [00:03<00:00, 30.73it/s]
```

Plot Accuracy and Loss from Training

```python
fig, (ax1, ax2) = plt.subplots(2, figsize=(12, 6), sharex=True)

ax1.plot(accuracy_list)
ax1.set_ylabel("validation accuracy")
ax2.plot(loss_list)
ax2.set_ylabel("validation loss")
ax2.set_xlabel("epochs");
```



Implementation of different Algorithms

I. Linear Model

II. Decision Tree

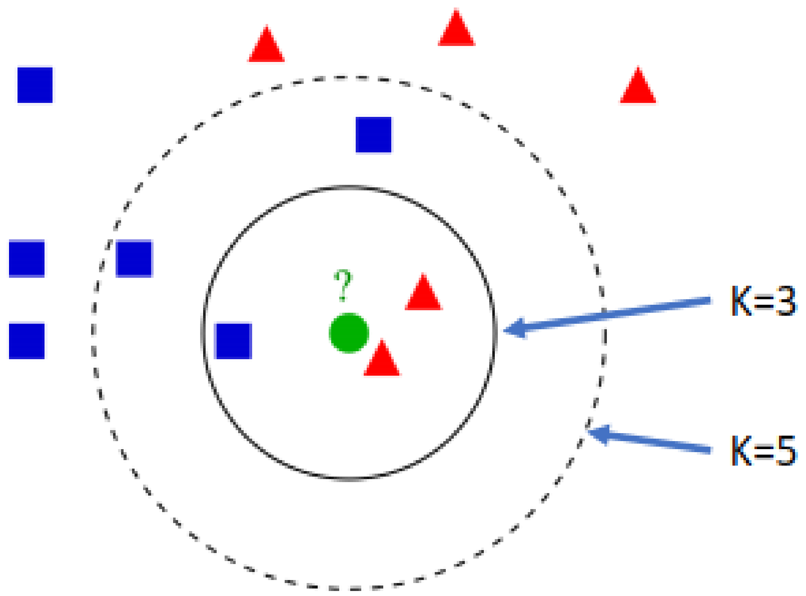
III. Neural Network

IV. **k-Nearest Neighbours**

V. Support Vector Machine

k-Nearest Neighbors

- For classification: find k nearest neighbors of the testing point and take a vote.
- For regression: take mean or median of the k nearest neighbors, or do a local regression on them.



kNN Implementation

```python
def knn(x,X,y,k=5):
    distances = np.sum((x - X)**2,axis=1)**0.5
    k_labels = [y[index] for index in distances.argsort()[:k]]
    return Counter(k_labels).most_common(1)[0][0]
```

[5] ✓ 0.7s

- 对距离排序使用 numpy 中的 argsort 函数，返回的是索引，因此取前 k 个索引使用 [0 : k]
- 使用 collections.Counter 可以统计各个标签的出现次数
- most_common 返回出现次数最多的标签 tuple，例如 [(‘lable1’，2)]，因此 [0][0] 可以取出标签值

kNN Performance

```
corrected = 0
for i in range(X_test.shape[0]):
    y_pred = knn(X_test[i],X_train,y_train)
    if y_pred == y_test[i]:
        corrected+=1
print("acc:",corrected/X_test.shape[0])
```

[6] ✓ 0.1s

```
... acc: 0.9333333333333333
```

kNN Issues

- Advantage:
  - Training is very fast.
  - Learn complex target functions.
  - Do not lose information.

- Disadvantage:
  - Slow at query time.
  - Easily fooled by irrelevant attributes.

Implementation of different Algorithms

I. Linear Model

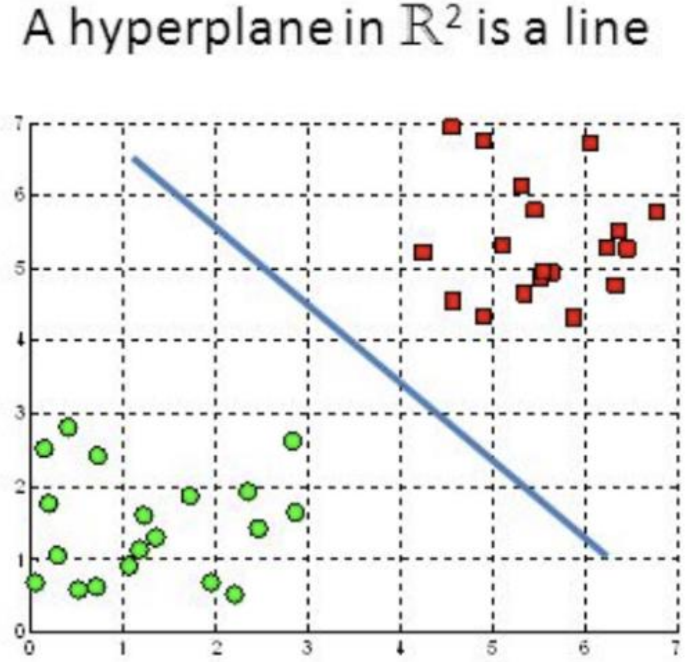
II. Decision Tree

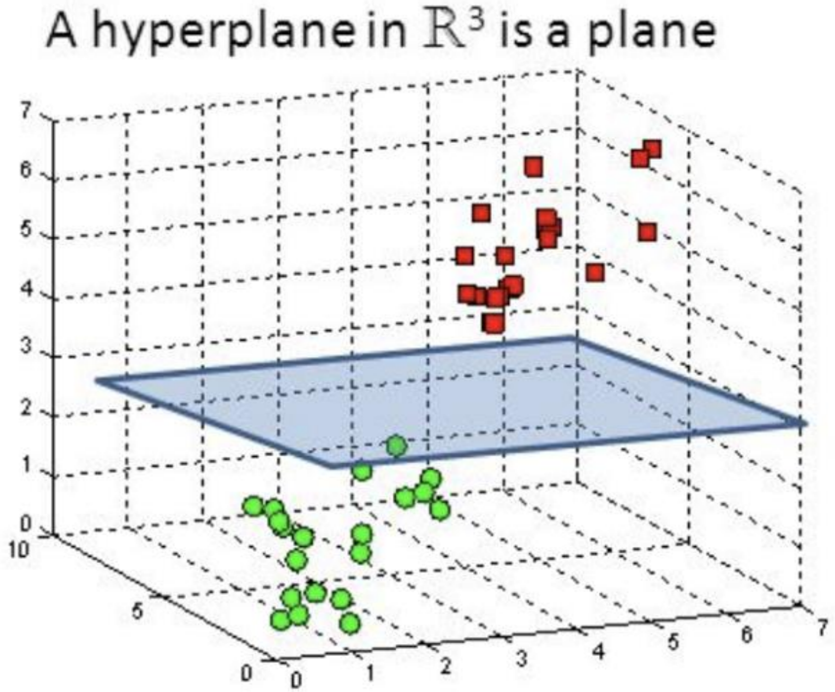
III. Neural Network

IV. k-Nearest Neighbours

V. **Support Vector Machine**

Support Vector Machine





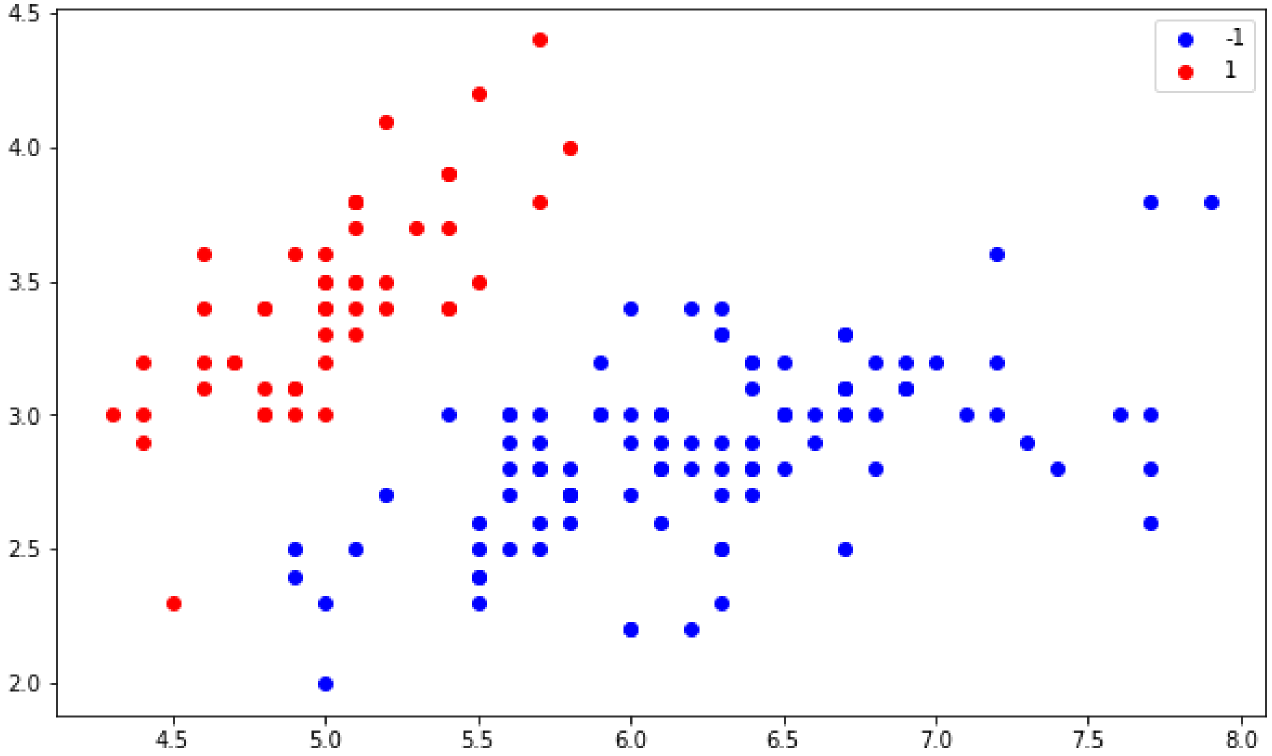
Support Vector Machine

- Training Data: $\mathcal{D} = \left\{\left(\boldsymbol{x}^{(n)}, y^{(n)}\right)\right\}_{n=1}^{N}$.
- Optimization: maximize the margin with the constraints as

$$\max_{\boldsymbol{w}} \frac{2}{||\boldsymbol{w}||^2},$$
$$s.t.\ [\boldsymbol{w} \cdot \boldsymbol{x}^{(n)} + b] \cdot y_i^{(n)} \geq 1$$

- Learning algorithm [3]:
  - Lagrange multiplier with KKT condition ⇒ Dual representation.
  - Gradient descent.

2-class classification




```python
iris = load_iris()

X = iris.data[:,:2]

y = [1 if u==0  else -1
        for u in iris.target]
```

Loss function and Gradients

$$c(x, y, f(x)) = \begin{cases} 0, & \text{if } y * f(x) \geq 1 \\ 1 - y * f(x), & \text{else} \end{cases}$$

$$min_w \lambda \parallel w \parallel^2 + \sum_{i=1}^{n} (1 - y_i \langle x_i, w \rangle)_+$$

$$\frac{\delta}{\delta w_k} \lambda \parallel w \parallel^2 \quad = 2\lambda w_k$$

$$\frac{\delta}{\delta w_k} \left(1 - y_i \langle x_i, w \rangle\right)_+ \quad = \begin{cases} 0, & \text{if } y_i \langle x_i, w \rangle \geq 1 \\ -y_i x_{ik}, & \text{else} \end{cases}$$

Important functions

```
def svm_sgd(X, Y):

    w = np.zeros(len(X[0]))
    eta = 1
    epochs = 100000


    for epoch in range(1,epochs):
        for i, x in enumerate(X):
            if (Y[i]*np.dot(X[i], w)) < 1:
                w = w + eta * ( (X[i] * Y[i]) + (-2  *(1/epoch)* w) )
            else:
                w = w + eta * (-2  *(1/epoch)* w)

    return w

w = svm_sgd(X_train,y_train)
print(w)
```

```
[-10.32077563  17.82557048]
```

Performance

```
results = [1 if u>0 else -1 for u in np.dot(X_test, w)]
from sklearn.metrics import accuracy_score
print(accuracy_score(results,y_test))
```

```
0.9833333333333333
```

Exercises

1. Could you classify iris with multi-class linear regression classifier?

2. Could you classify iris with multi-class SVM classifier?