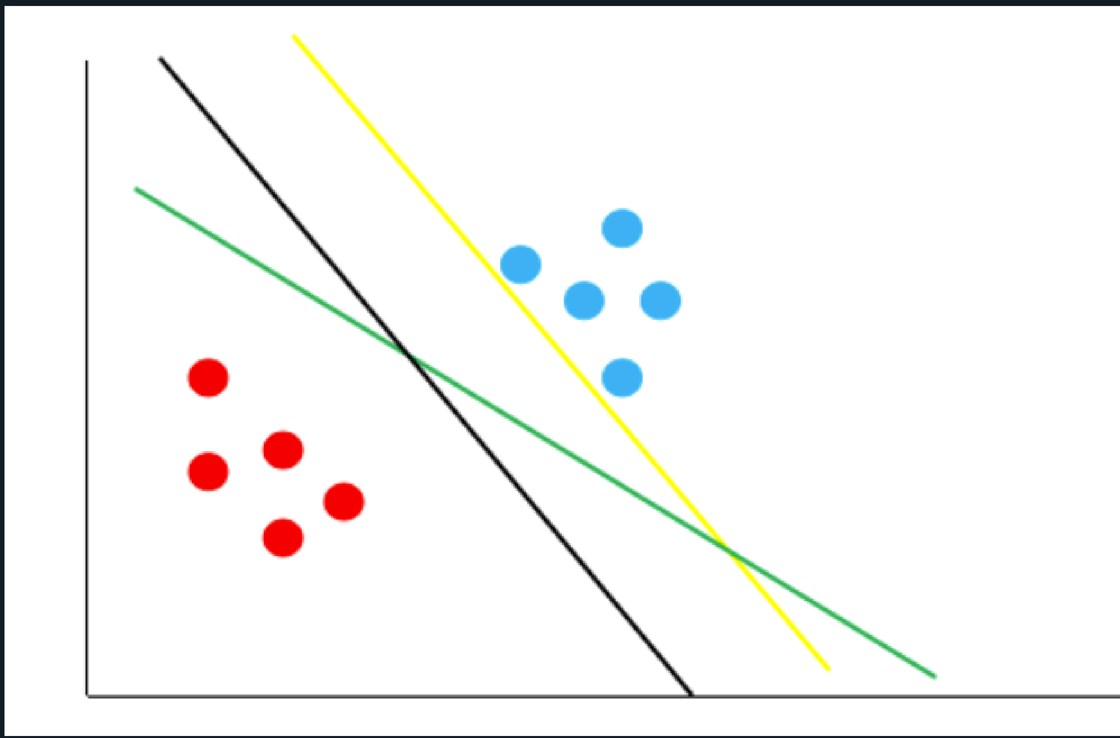
1、 按照支持向量机的思想，下图哪条决策边界的泛化性最好？



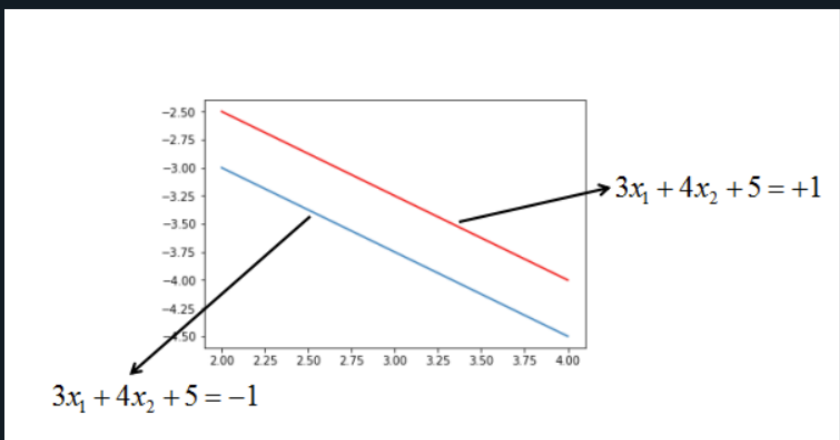
A、绿线

B、黑线

间隔

$$\begin{cases} 3x_1 + 4x_2 + 5 = +1 \\ 3x_1 + 4x_2 + 5 = -1 \end{cases}$$

如下图：



则最大间隔 r 的值为？

A、0.3

B、0.4

C、0.5

D、2

对偶

1、 下面说法正确的是?

- [x] A、支持向量机的最终模型仅仅与支持向量有关。
- [ ] B、支持向量机的最终模型由所有的训练样本共同决定。
- [x] C、支持向量机的最终模型由离决策边界最近的几个点决定。
- [ ] D、训练集越大，支持向量机的模型就一定越准确。

核函数

```
#encoding=utf8

import numpy as np

#实现核函数
def kernel(x,sigma=1.0):
    '''
    input:x(ndarray):样本
    output:x(ndarray):转化后的值
    '''
    #********* Begin *********#
    size = x.shape[0]
    for i in range(size):
        r = x[i,0]-x[i,1]
        x[i,0] = np.exp(r * r.T / (-2*sigma**2))
        x[i,1] = np.exp(r * r.T / (-2*sigma**2))
    #********* End *********#
    return x
```

软间隔

```
#encoding=utf8
import numpy as np
class SVM:
    def __init__(self, max_iter=100, kernel='linear'):
        '''
        input:max_iter(int):最大训练轮数
```

```
            kernel(str):核函数，等于'linear'表示线性，等于'poly'表示多项
式
        '''
        self.max_iter = max_iter
        self._kernel = kernel
    def init_args(self, features, labels):
        self.m, self.n = features.shape
        self.X = features
        self.Y = labels
        self.b = 0.0
        self.alpha = np.ones(self.m)
        self.E = [self._E(i) for i in range(self.m)]
        self.C = 1.0
    #********* Begin *********#
    def _KKT(self, i):
        y_g = self._g(i)*self.Y[i]
        if self.alpha[i] == 0:
            return y_g >= 1
        elif 0 < self.alpha[i] < self.C:
            return y_g == 1
        else:
            return y_g <= 1

    def _g(self, i):
        r = self.b
        for j in range(self.m):
            r += self.alpha[j]*self.Y[j]*self.kernel(self.X[i],
self.X[j])
        return r

    def kernel(self, x1, x2):
        if self._kernel == 'linear':
            return sum([x1[k]*x2[k] for k in range(self.n)])
        elif self._kernel == 'poly':
            return (sum([x1[k]*x2[k] for k in range(self.n)]) +
1)**2
        return 0

    def _E(self, i):
        return self._g(i) - self.Y[i]

    def _init_alpha(self):
```

```
        index_list = [i for i in range(self.m) if 0 < self.alpha[i]
< self.C]
        non_satisfy_list = [i for i in range(self.m) if i not in
index_list]
        index_list.extend(non_satisfy_list)
        for i in index_list:
            if self._KKT(i):
                continue
            E1 = self.E[i]
            if E1 >= 0:
                j = min(range(self.m), key=lambda x: self.E[x])
            else:
                j = max(range(self.m), key=lambda x: self.E[x])
            return i, j

    def _compare(self, _alpha, L, H):
        if _alpha > H:
            return H
        elif _alpha < L:
            return L
        else:
            return _alpha

    def fit(self, features, labels):
        self.init_args(features, labels)
        for t in range(self.max_iter):
            i1, i2 = self._init_alpha()
            if self.Y[i1] == self.Y[i2]:
                L = max(0, self.alpha[i1]+self.alpha[i2]-self.C)
                H = min(self.C, self.alpha[i1]+self.alpha[i2])
            else:
                L = max(0, self.alpha[i2]-self.alpha[i1])
                H = min(self.C, self.C+self.alpha[i2]-
self.alpha[i1])
            E1 = self.E[i1]
            E2 = self.E[i2]
            eta = self.kernel(self.X[i1], self.X[i1]) +
self.kernel(self.X[i2], self.X[i2]) - 2*self.kernel(self.X[i1],
self.X[i2])
            if eta <= 0:
                continue
```

```
            alpha2_new_unc = self.alpha[i2] + self.Y[i2] * (E2 -
E1) / eta
            alpha2_new = self._compare(alpha2_new_unc, L, H)
            alpha1_new = self.alpha[i1] + self.Y[i1] * self.Y[i2] *
(self.alpha[i2] - alpha2_new)
            b1_new = -E1 - self.Y[i1] * self.kernel(self.X[i1],
self.X[i1]) * (alpha1_new-self.alpha[i1]) - self.Y[i2] *
self.kernel(self.X[i2], self.X[i1]) * (alpha2_new-self.alpha[i2])+
self.b
            b2_new = -E2 - self.Y[i1] * self.kernel(self.X[i1],
self.X[i2]) * (alpha1_new-self.alpha[i1]) - self.Y[i2] *
self.kernel(self.X[i2], self.X[i2]) * (alpha2_new-self.alpha[i2])+
self.b
            if 0 < alpha1_new < self.C:
                b_new = b1_new
            elif 0 < alpha2_new < self.C:
                b_new = b2_new
            else:
                b_new = (b1_new + b2_new) / 2
            self.alpha[i1] = alpha1_new
            self.alpha[i2] = alpha2_new
            self.b = b_new
            self.E[i1] = self._E(i1)
            self.E[i2] = self._E(i2)
    #********* End *********#
    def predict(self, data):
        r = self.b
        for i in range(self.m):
            r += self.alpha[i] * self.Y[i] * self.kernel(data,
self.X[i])
        return 1 if r > 0 else -1
    def score(self, X_test, y_test):
        right_count = 0
        for i in range(len(X_test)):
            result = self.predict(X_test[i])
            if result == y_test[i]:
                right_count += 1
        return right_count / len(X_test)
    def _weight(self):
        yx = self.Y.reshape(-1, 1)*self.X
        self.w = np.dot(yx.T, self.alpha)
        return self.w
```

sklearn

```python
#encoding=utf8
from sklearn.svm import SVC

def svm_classifier(train_data,train_label,test_data):
    '''
    input:train_data(ndarray):训练样本
          train_label(ndarray):训练标签
          test_data(ndarray):测试样本
    output:predict(ndarray):预测结果
    '''
    #********* Begin *********#
    svc = SVC()
    svc.fit(train_data,train_label)
    predict = svc.predict(test_data)
    #********* End *********#
    return predict
```