T05

October 13, 2025

0.1 KNN

We have mostly focused on parametric models, like $p(y \mid x, \theta)$, where θ is a fixed-dimensional vector of parameters. The parameters are estimated from a variable-sized dataset, $\mathcal{D} = \{(x_n, y_n) : n = 1 : N\}$, but after model fitting, the data is thrown away.

We also consider various kinds of nonparametric models, that keep the training data around. Thus the memory usage of the model can grow with $|\mathcal{D}|$. We focus on models that can be defined in terms of the similarity between a test input, x, and each of the training inputs, x_n. Alternatively, we can define the models in terms of a dissimilarity or distance function $d(x, x_n)$.

We discuss one of the simplest kind of classifier, known as the **K** nearest neighbor (KNN) classifier. The idea is as follows: to classify a new input x, we find the K closest examples to x in the training set, denoted $N_K(x, \mathcal{D})$, and then look at their labels, to derive a distribution over the outputs for the local region around x. More precisely, we compute

$$p(y = c \mid x, \mathcal{D}) = \frac{1}{K} \sum_{n \in N_K(x, \mathcal{D})} \mathbb{I}(y_n = c)$$

We can then return this distribution, or the majority label. The two main parameters in the model are the size of the neighborhood, K, and the distance metric $d(x, x')$. For the latter, it is common to use the Mahalanobis distance

$$d_{\mathrm{M}}(x, \mu) = \sqrt{(x - \mu)^\top \mathrm{M}(x - \mu)}$$

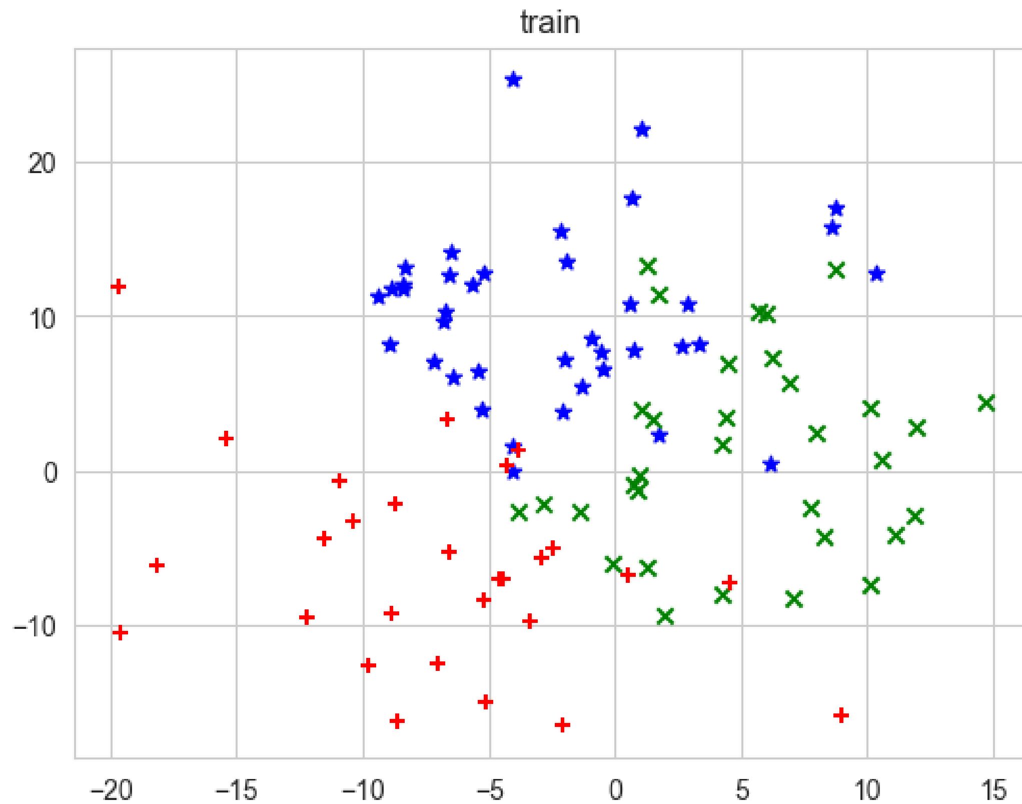
where $\mathbf{M}$ is a positive definite matrix. If $\mathbf{M} = \mathbf{I}$, this reduces to Euclidean distance.

[1]:
```
# Imports

from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.model_selection import cross_val_score


from sklearn.datasets import make_blobs
from IPython import display
from matplotlib import pyplot as plt

import numpy as np
```

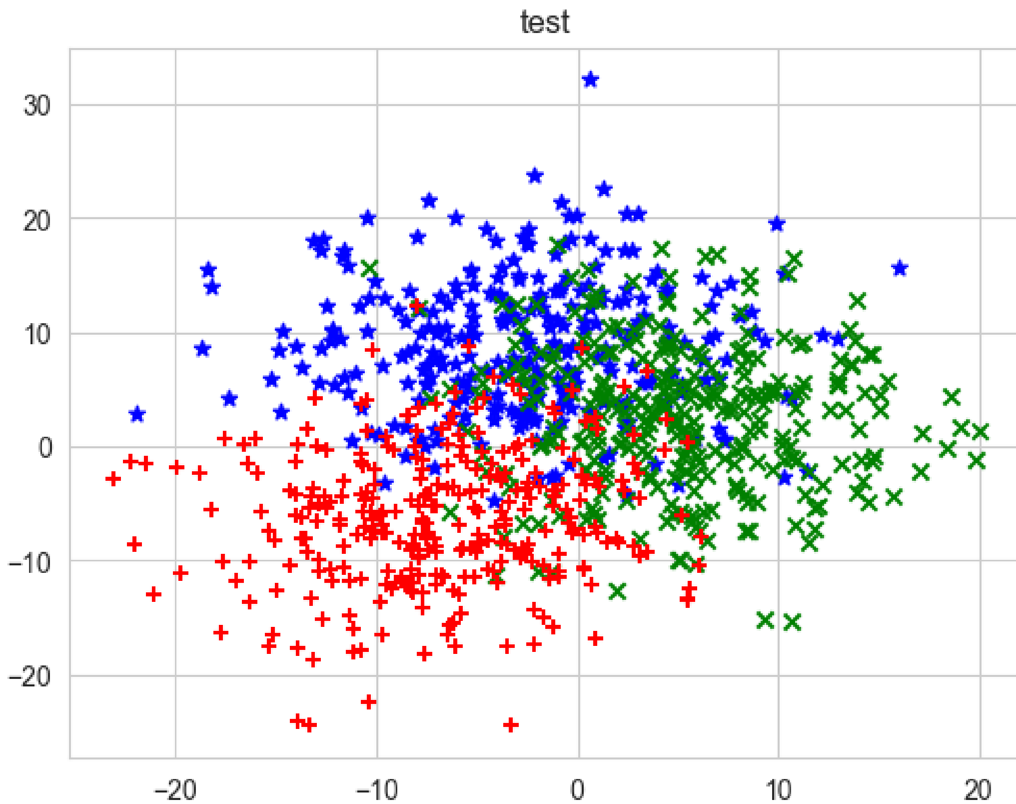
```python
import pathlib
import shutil
import tempfile

from tqdm import tqdm
```

[12]:
```python
# In this notebook we will walk through KNN clustering technique
# Here we generate isotropic Gaussian blobs by using the make_blob function␣
↪from sklearn
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=6,␣
↪random_state=42)
ntrain = 100
x_train = X[:ntrain]
y_train = y[:ntrain]
x_test = X[ntrain:]
y_test = y[ntrain:]
```

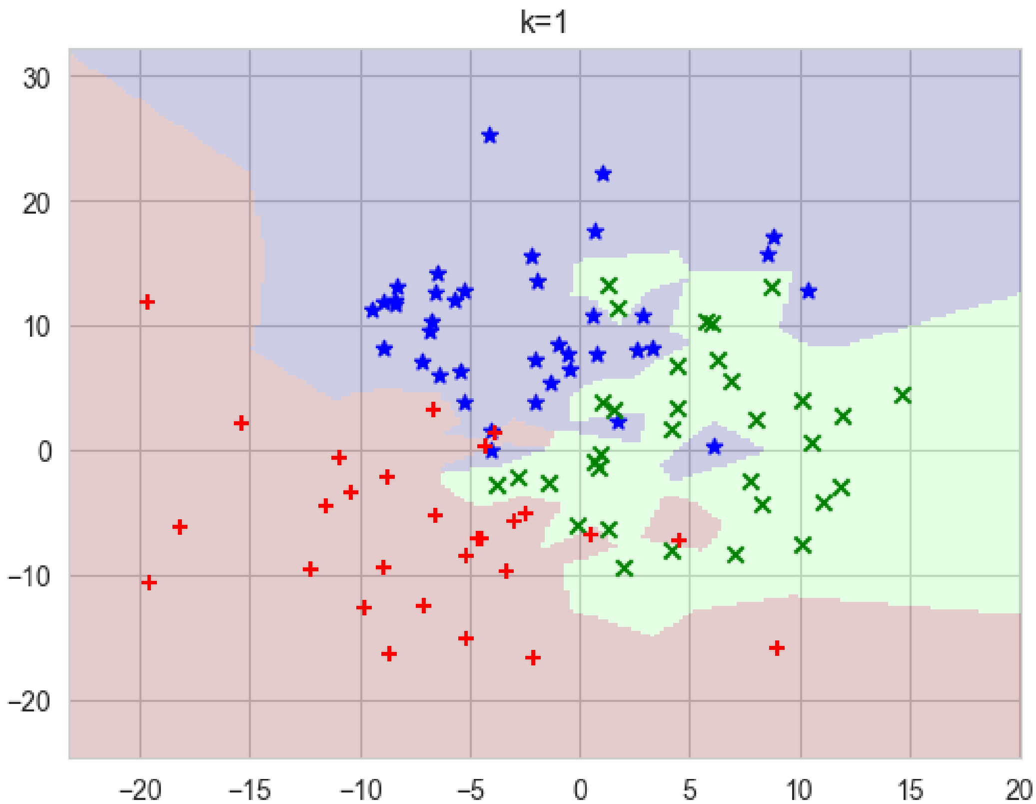
[13]:
```python
# Plotting the generated training dataset by class in a scatter plot
plt.figure()
y_unique = np.unique(y_train)
markers = "*x+"
colors = "bgr"
for i in range(len(y_unique)):
    plt.scatter(x_train[y_train == y_unique[i], 0], x_train[y_train ==␣
↪y_unique[i], 1], marker=markers[i], c=colors[i])
plt.title("train")

plt.show()
```

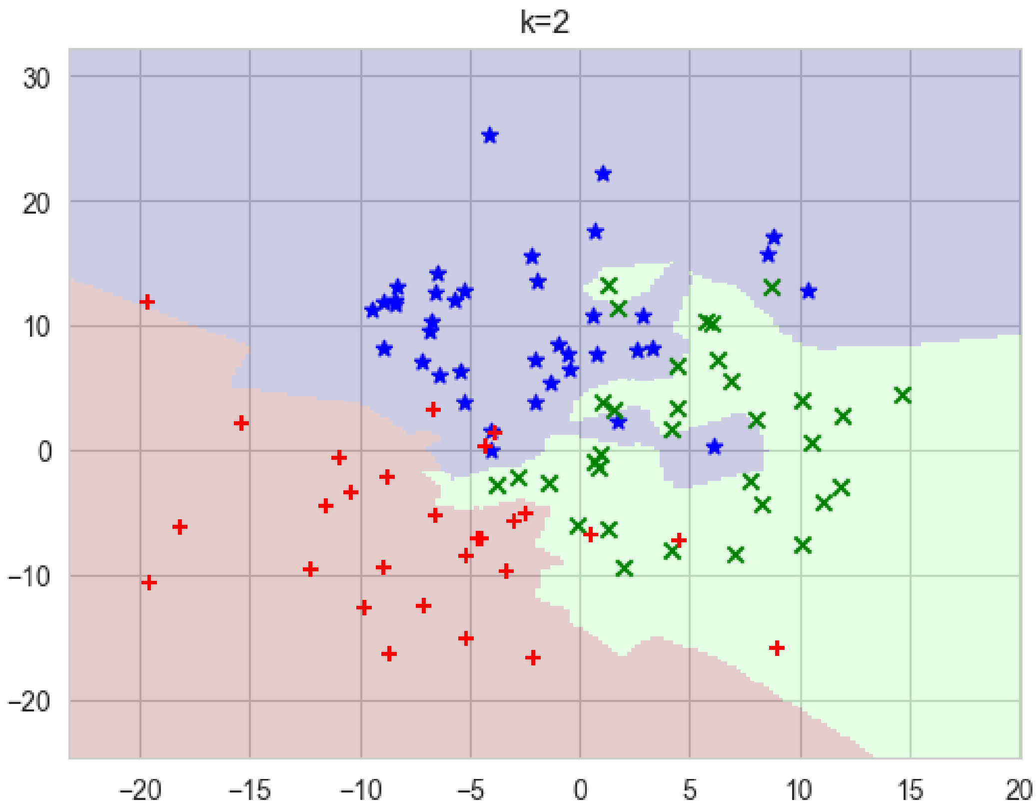


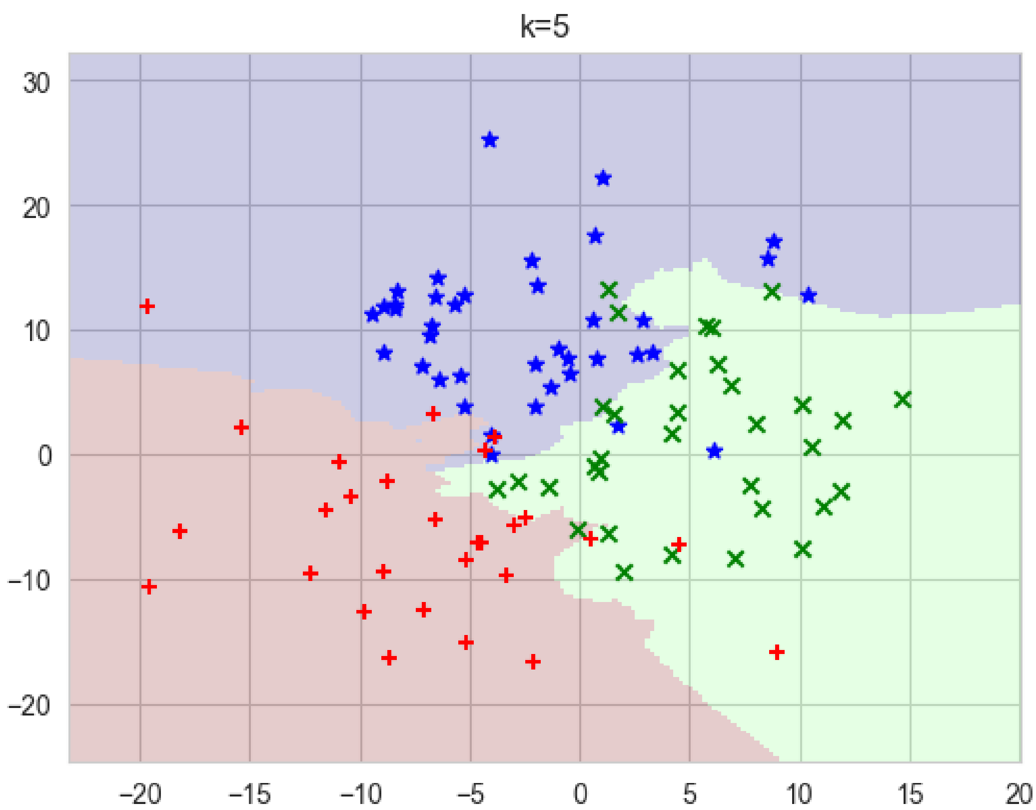
[14]:
```
# Plotting the generated test dataset by class in a scatter plot
plt.figure()
for i in range(len(y_unique)):
    plt.scatter(x_test[y_test == y_unique[i], 0], x_test[y_test == y_unique[i],␣
 ↪1], marker=markers[i], c=colors[i])
plt.title("test")

plt.show()
```



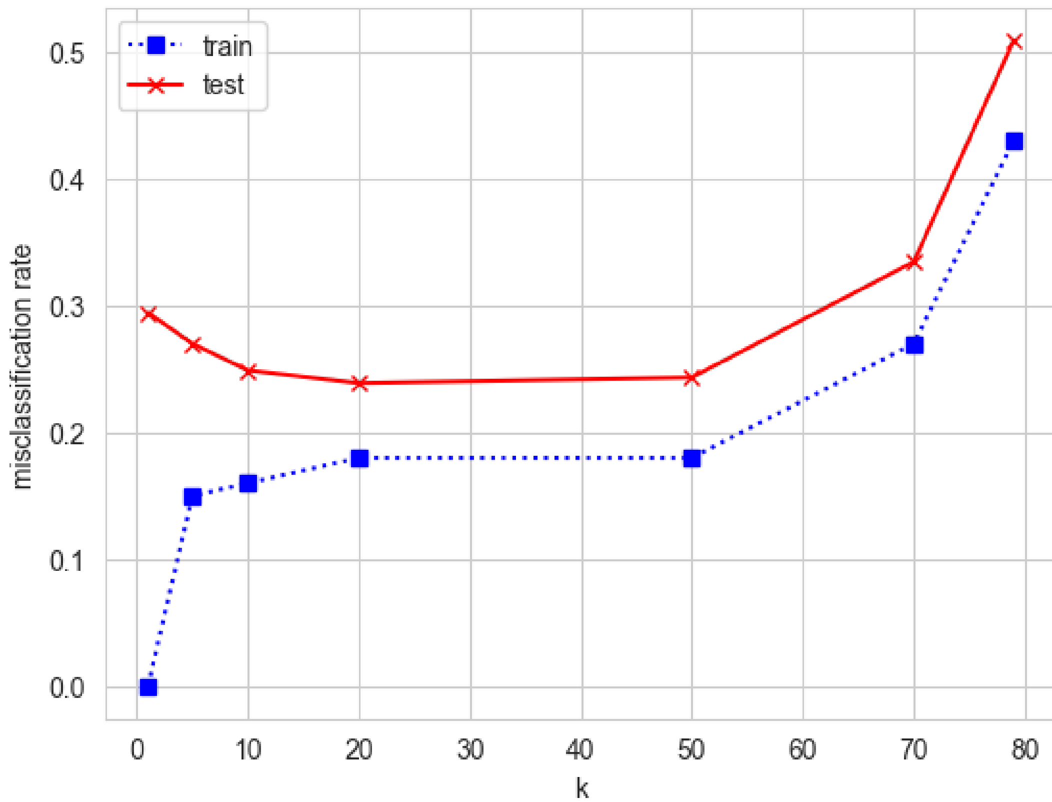
[15]:
```
x = np.linspace(np.min(x_test[:, 0]), np.max(x_test[:, 0]), 200)
y = np.linspace(np.min(x_test[:, 1]), np.max(x_test[:, 1]), 200)
xx, yy = np.meshgrid(x, y)
xy = np.c_[xx.ravel(), yy.ravel()]

# Train a knn model and use the knn model to predict
for k in [1, 2, 5]:
    knn = KNN(n_neighbors=k)
    knn.fit(x_train, y_train)
    plt.figure()
    y_predicted = knn.predict(xy)

    plt.pcolormesh(xx, yy, y_predicted.reshape(200, 200), cmap="jet", alpha=0.2)
    for i in range(len(y_unique)):
        plt.scatter(
            x_train[y_train == y_unique[i], 0], x_train[y_train == y_unique[i],␣
 ↪1], marker=markers[i], c=colors[i]
        )
    plt.title("k=%s" % (k))

    plt.show()
```


k=1
30
20
10
0
−10
−20
−20
−15
−10
−5
0
5
10
15
20

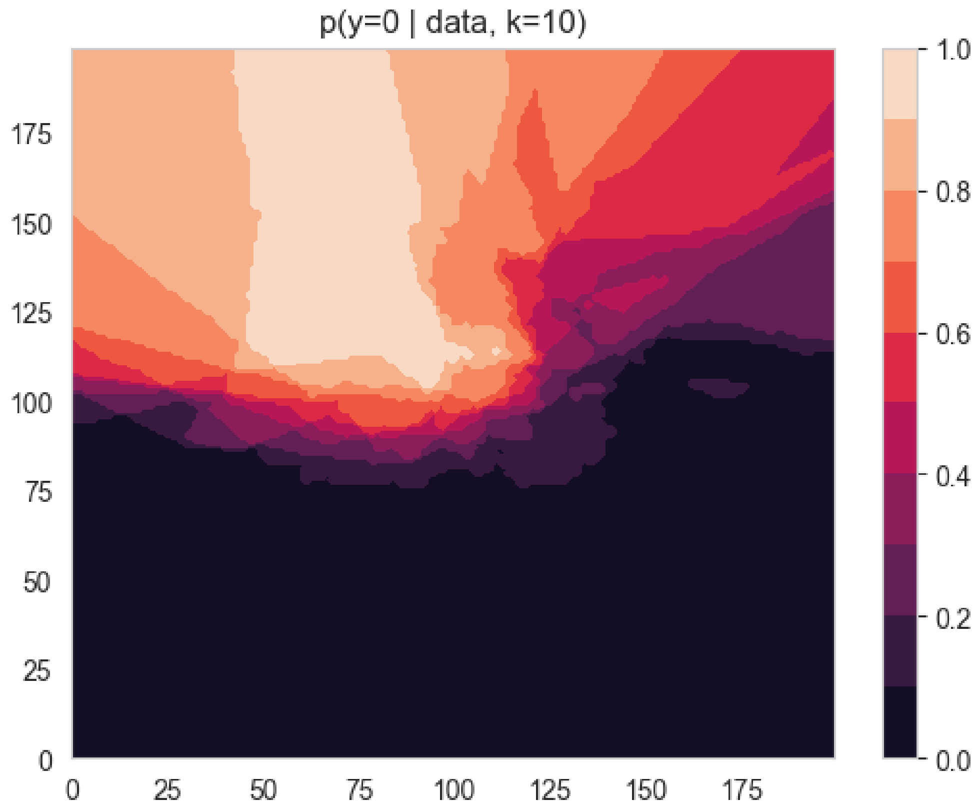

k=2
30
20
10
0
−10
−20
−20
−15
−10
−5
0
5
10
15
20

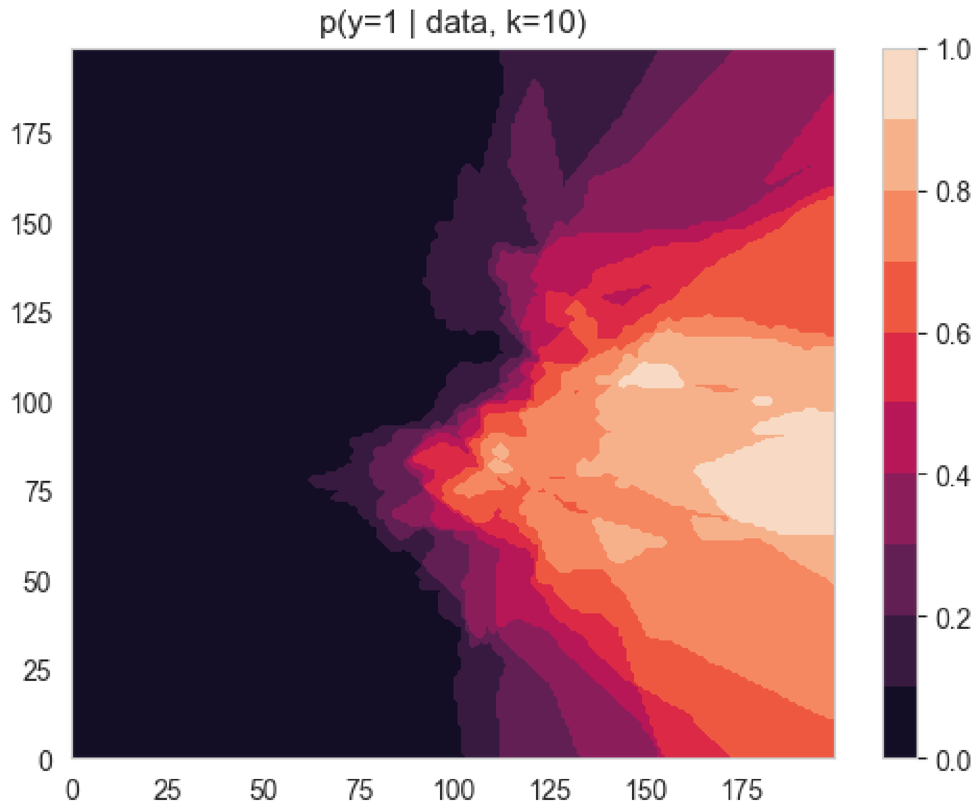


[16]:
```
# plot train err and test err with different k
# ks = [int(n) for n in np.linspace(1, ntrain, 10)]
ks = [1, 5, 10, 20, 50, 70, 79]
train_errs = []
test_errs = []
for k in ks:
    knn = KNN(n_neighbors=k)
    knn.fit(x_train, y_train)
    train_errs.append(1 - knn.score(x_train, y_train))
    test_errs.append(1 - knn.score(x_test, y_test))
plt.figure()
plt.plot(ks, train_errs, "bs:", label="train")
plt.plot(ks, test_errs, "rx-", label="test")
plt.legend()
plt.xlabel("k")
plt.ylabel("misclassification rate")

plt.show()
```

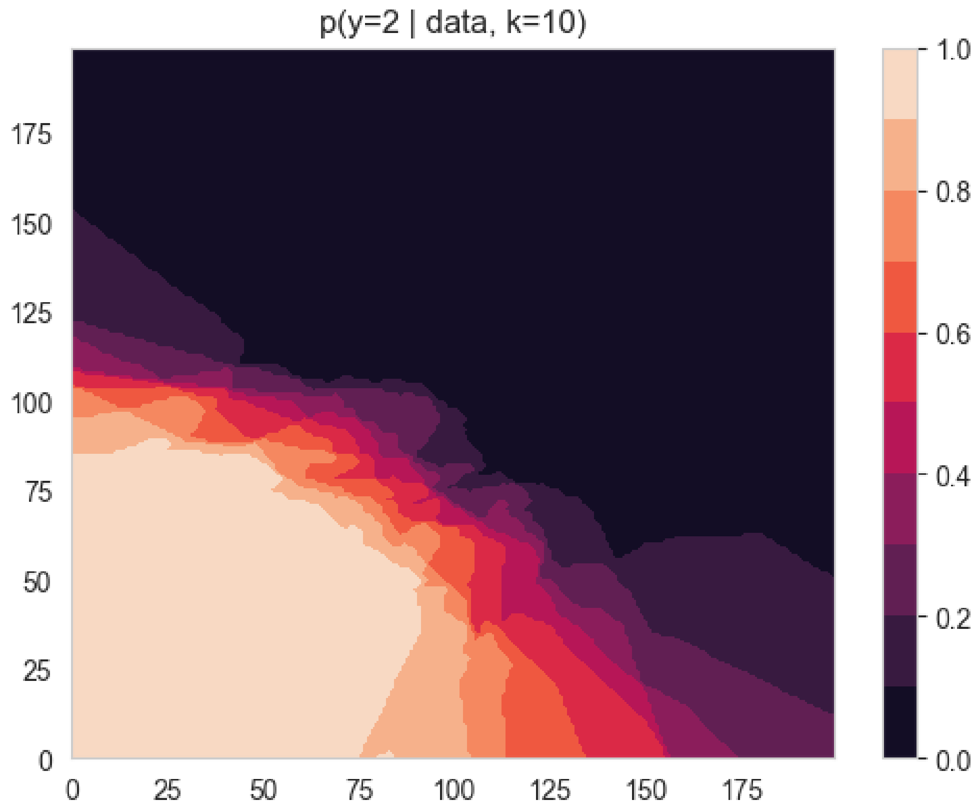


[17]:
```
# draw hot-map to show the probability of different class
knn = KNN(n_neighbors=10)
knn.fit(x_train, y_train)
xy_predic = knn.predict_proba(xy)
levels = np.arange(0, 1.01, 0.1)
for i in range(3):
    plt.figure()
    plt.contourf(xy_predic[:, i].ravel().reshape(200, 200), levels)
    plt.colorbar()
    plt.title("p(y=%s | data, k=10)" % (i))
plt.show()
```


p(y=0 | data, k=10)
175
150
125
100
75
50
25
0
0
25
50
75
100
125
150
175
1.0
0.8
0.6
0.4
0.2
0.0



p(y=1 | data, k=10)
0
25
50
75
100
125
150
175
0
25
50
75
100
125
150
175
0.0
0.2
0.4
0.6
0.8
1.0



0.2 Multiclass Logistic Regression

0.3 Let's first recall logistic regression for classification!

- Training set $\mathcal{D} = \{\mathbf{x}_i, y_i\}_{i=1}^N$, where $y_i \in \{0, 1\}$.
- Probabilistic model

$$p(y \mid \mathbf{x}, \beta) = \mathrm{Ber}\left(y \mid \sigma\left(\beta^\top \mathbf{x}\right)\right)$$

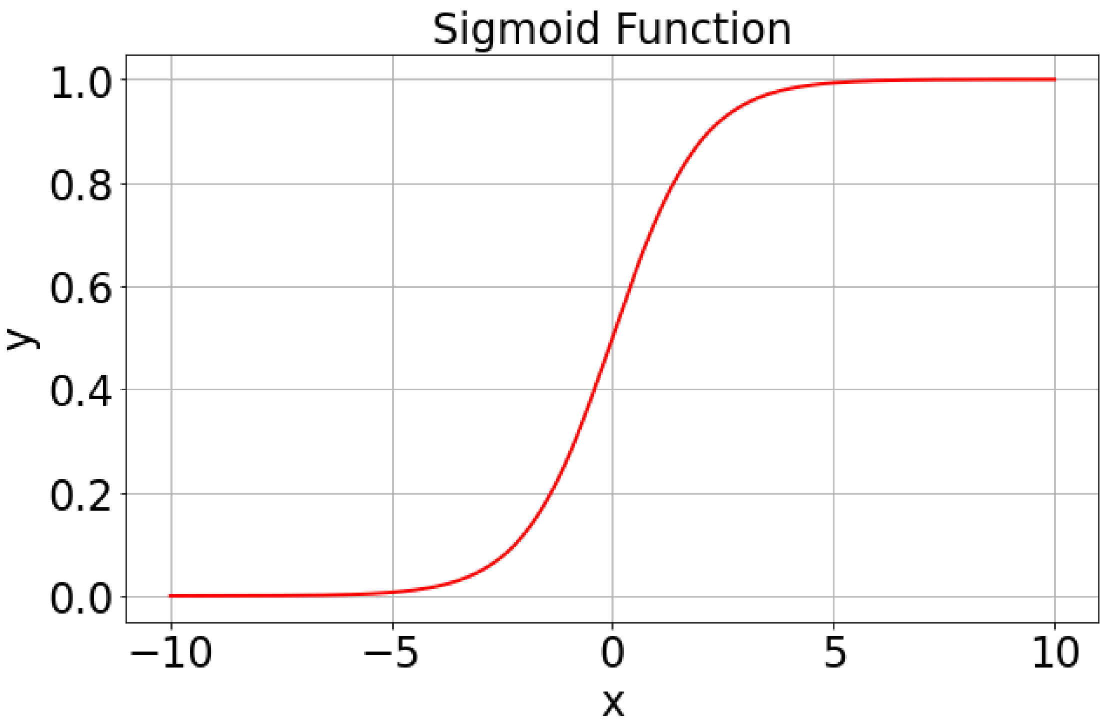
  - $\sigma(z)$ is the sigmoid/logistic/logit function.

$$\sigma(z) = \frac{1}{1 + \exp(-z)} = \frac{e^z}{e^z + 1}$$

  - It maps $\mathbb{R}$ to $(0, 1)$.

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm

# Set random seed
np.random.seed(20221011)
```

```python
# Generate data for the sigmoid function
x = np.sort(np.random.uniform(-10, 10, 1000))
y = 1 / (1 + np.exp(-x))

# Plot the sigmoid function
plt.figure(figsize=(10, 6))
plt.plot(x, y, color='red', linewidth=2)
plt.title('Sigmoid Function', fontsize=24)
plt.xlabel('x', fontsize=24)
plt.ylabel('y', fontsize=24)
plt.tick_params(axis='both', labelsize=24)
plt.grid()
plt.show()
```



0.4 The maximum likelihood estimation

- Recall that, the likelihood is the joint probability function of joint density function of the data.
- Here, we have independent observations $(\mathbf{x}_i, y_i), i = 1, \ldots, n$, each follow the (conditional) distribution

$$P(y_i = 1 \mid \mathbf{x}_i) = \frac{1}{1 + \exp(-\beta^T \mathbf{x}_i)} = 1 - P(y_i = 0 \mid \mathbf{x}_i)$$

- So, the joint probability function is

$$\prod_{i=1,\dots,n;y_i=1} p\left(y_i = 1 \mid \mathbf{x}_i\right) \prod_{i=1,\dots,n;y_i=0} p\left(y_i = 0 \mid \mathbf{x}_i\right)$$

  which can be conveniently written as

$$\prod_{i=1}^{n} \frac{\exp\left(y_i \beta^T \mathbf{x}_i\right)}{1 + \exp\left(\beta^T \mathbf{x}_i\right)}$$

- The likelihood function is the same as the joint probability function, but viewed as a function of β. The log-likelihood function is

$$\ell = \sum_{i=1}^{n} \left[y_i \beta^T x_i - \log\left(1 + \exp\left(\beta^T \mathbf{x}_i\right)\right)\right]$$

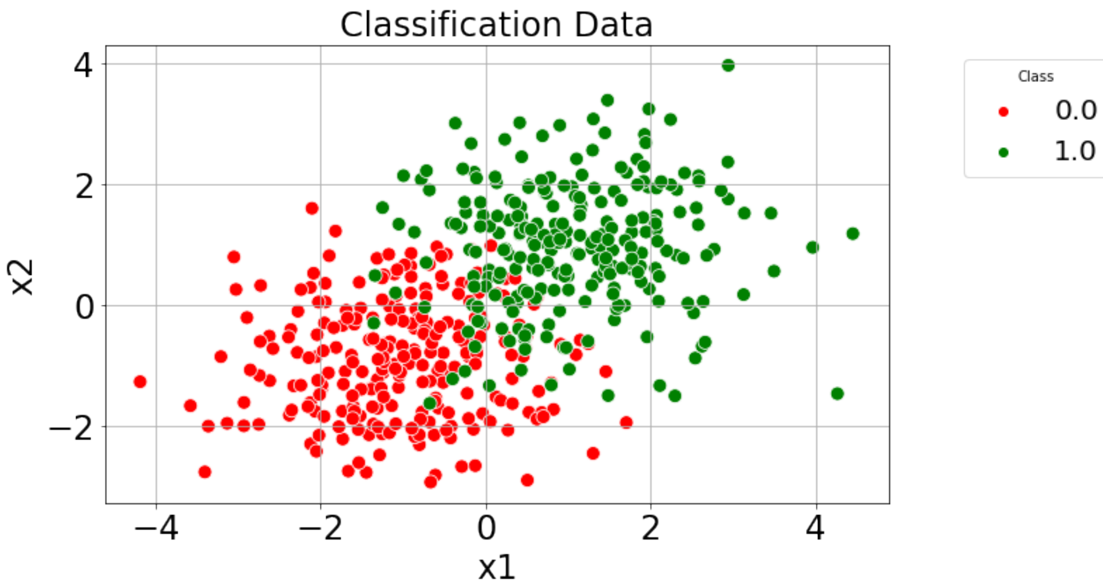
- Unlike linear regression, we can no longer write down the MLE in closed form. Instead, we need to use optimization algorithms to compute it.
  - Gradient descent
  - Newton's method

0.5 Generate data

[ ]:
```
n = 500  # Sample size
p = 2    # Number of features

y = np.concatenate([np.zeros(n // 2), np.ones(n // 2)])
x_class1 = np.random.randn(n // 2, p) - 1
x_class2 = np.random.randn(n // 2, p) + 1
x = np.vstack((x_class1, x_class2))
data = pd.DataFrame(np.column_stack((y, x)), columns=["y", "x1", "x2"])
```

0.6 Visualization

[ ]:
```
# Convert y to categorical
data['y'] = data['y'].astype('category')

# Plot the classification data
plt.figure(figsize=(10, 6))
sns.scatterplot(data=data, x='x1', y='x2', hue='y', palette=['red', 'green'],␣
↪s=100)
plt.title('Classification Data', fontsize=24)
plt.xlabel('x1', fontsize=24)
plt.ylabel('x2', fontsize=24)
plt.tick_params(axis='both', labelsize=24)
plt.legend(title='Class', fontsize=20, bbox_to_anchor = (1.3,1))
plt.grid()
plt.show()
```



0.7 Implementation

```
[ ]: def mylogistic(y, x, method="Hessian", maxIter=500):
         n, p = x.shape
         y = y.reshape(-1,1)
         x = np.column_stack((np.ones(n), x))  # Add intercept
         XtX = np.dot(x.T, x)

         beta_old = np.zeros((p + 1, 1))
         prob = 1 / (1 + np.exp(-np.dot(x, beta_old)))

         for iter in range(maxIter):
             if method == "Hessian":
                 W = prob * (1 - prob)
                 temp = np.sqrt(W) * x
                 XWX = np.dot(temp.T, temp)
                 # Newton-Raphson update
                 invH = np.linalg.inv(XWX)
                 beta = beta_old + np.dot(invH, (np.dot(x.T, (y - prob))))
             else:
                 # This is a method using the upper bound of Hessian
                 # Because prob*(1-prob) <= 0.25
                 # We replace prob*(1-prob) by 0.25
                 z = 0.25 * np.dot(x, beta_old) + (y - prob)
                 beta = np.linalg.solve(0.25 * XtX, np.dot(x.T, z))

             if np.max(np.abs(beta_old - beta)) / np.sqrt(np.sum(beta**2)) < 1e-6:
```

```
            break

        prob = 1 / (1 + np.exp(-np.dot(x, beta)))
        prob = np.clip(prob, 0.001, 0.999)

        beta_old = beta

    se = np.sqrt(np.diag(invH))
    return {'prob': prob, 'beta': beta, 'se': se, 'Iter': iter + 1}
```

[ ]:
```
my_fit = mylogistic(y=(data.iloc[:,0].values).astype('int'), x=data[['x1',␣
↪'x2']].values, method="Hessian")
print("Estimated coefficients:", my_fit['beta'].flatten())
print("Iterations:", my_fit['Iter'])
print("Standard errors:", my_fit['se'])
```

```
Estimated coefficients: [-0.08002258  2.17128394  1.88221187]
Iterations: 10
Standard errors: [0.17984921 0.25011241 0.2247756 ]
```

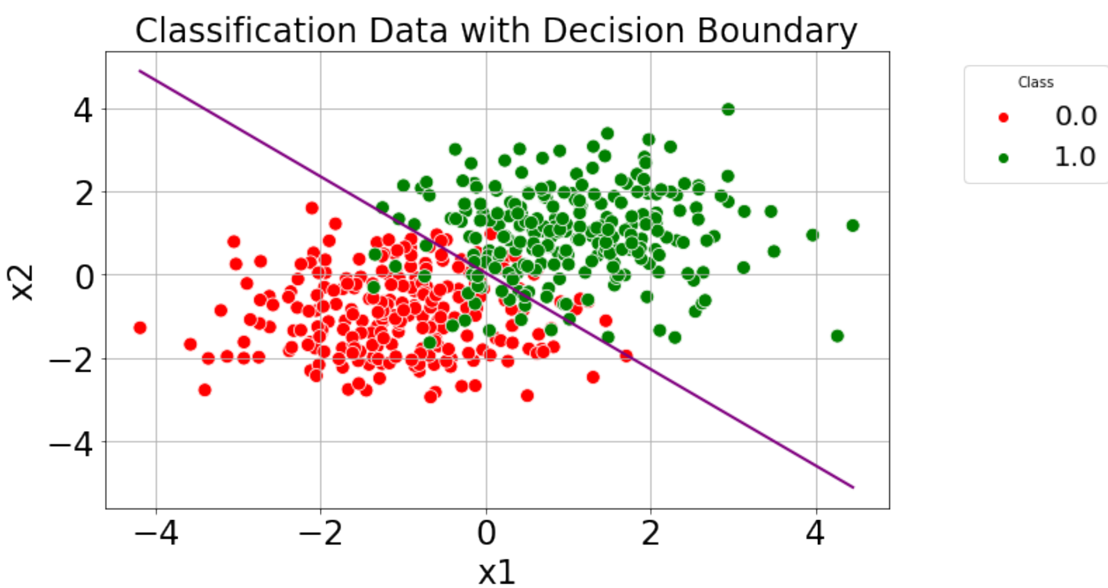
0.8 Built-in function

[ ]:
```
glm_fit = sm.Logit(data['y'].cat.codes, sm.add_constant(data[['x1', 'x2']])).
↪fit()
print(glm_fit.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.204026
         Iterations 8
                           Logit Regression Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  500
Model:                          Logit   Df Residuals:                      497
Method:                           MLE   Df Model:                            2
Date:                Sun, 13 Oct 2024   Pseudo R-squ.:                  0.7057
Time:                        21:00:02   Log-Likelihood:                -102.01
converged:                       True   LL-Null:                       -346.57
Covariance Type:            nonrobust   LLR p-value:                 6.147e-107
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0799      0.180     -0.445      0.656      -0.432       0.272
x1             2.1602      0.253      8.534      0.000       1.664       2.656
x2             1.8735      0.227      8.265      0.000       1.429       2.318
==============================================================================
```

[ ]:
```
plt.figure(figsize=(10, 6))
```

```
sns.scatterplot(data=data, x='x1', y='x2', hue='y', palette=['red', 'green'],␣
    ↪s=100)
x_vals = np.linspace(data['x1'].min(), data['x1'].max(), 100)
y_vals = (-my_fit['beta'][0] - my_fit['beta'][1] * x_vals) / my_fit['beta'][2]
plt.plot(x_vals, y_vals, color='purple', linewidth=2)
plt.title('Classification Data with Decision Boundary', fontsize=24)
plt.xlabel('x1', fontsize=24)
plt.ylabel('x2', fontsize=24)
plt.tick_params(axis='both', labelsize=24)
plt.legend(title='Class', fontsize=20, bbox_to_anchor = (1.3,1))
plt.grid()
plt.show()
```



0.9 Multiclass logistic regression

We now extend the two-class logistic regression approach to the setting of $K > 2$ classes. This extension is known as multiclass logistic regression or multinomial logistic regression.

To do this, we first select a single class to serve as the **baseline** (why?); without loss of generality, we select the K-th class for this role. Then

$$\Pr(Y = k \mid X = x) = \frac{e^{\beta_{k0}+\beta_{k1}x_1+\cdots+\beta_{kp}x_p}}{1 + \sum_{l=1}^{K-1} e^{\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{lp}x_p}}$$

for $k = 1, \dots, K-1$, and

$$\Pr(Y = K \mid X = x) = \frac{1}{1 + \sum_{l=1}^{K-1} e^{\beta_{l0}+\beta_{l1}x_1+\cdots+\beta_{lp}x_p}}.$$

It is not hard to show that for $k = 1, \dots, K-1$,

$$\log\left(\frac{\Pr(Y=k \mid X=x)}{\Pr(Y=K \mid X=x)}\right) = \beta_{k0} + \beta_{k1}x_1 + \cdots + \beta_{kp}x_p$$

Notice that the log odds between any pair of classes is linear in the features.

0.10 A toy example

0.10.1 Generate data

[ ]:
```
np.random.seed(20241013)
N = 1000  # Sample size
P = 2     # Number of features
K = 3     # Number of classes

# Generate features and coefficients
X = np.random.randn(N, P)  # Features, N * P
b = np.random.normal(3, 1, (P, K - 1))  # Coefficients, P * (K - 1)
```

[ ]:
```
X
```

```
array([[ 1.81899607,  0.27176549],
       [-0.18405267, -0.57797361],
       [ 0.58327899, -0.32941768],
       …,
       [-1.46430942, -1.35279912],
       [ 1.05492677,  0.41762662],
       [-0.10525239, -0.77753856]])
```

[ ]:
```
b
```

```
array([[1.98920896, 3.84405831],
       [3.36578599, 2.57251504]])
```

[ ]:
```
f = np.exp(X @ b)  # N * (K - 1)
prob = f / (1 + np.sum(f, axis=1, keepdims=True))  # Prob of Class 1 and 2, N *
↪(K - 1)
prob = np.hstack((prob, 1 - np.sum(prob, axis=1, keepdims=True)))  # Prob of
↪Class 1, 2, and 3, N * K
```

[ ]:
```
f
```

```
array([[9.30435604e+01, 2.18954226e+03],
       [9.91169619e-02, 1.11430413e-01],
       [1.05286433e+00, 4.03387121e+00],
       …,
       [5.72184114e-04, 1.10666777e-04],
       [3.32518904e+01, 1.68941211e+02],
       [5.92258382e-02, 9.02812076e-02]])
```

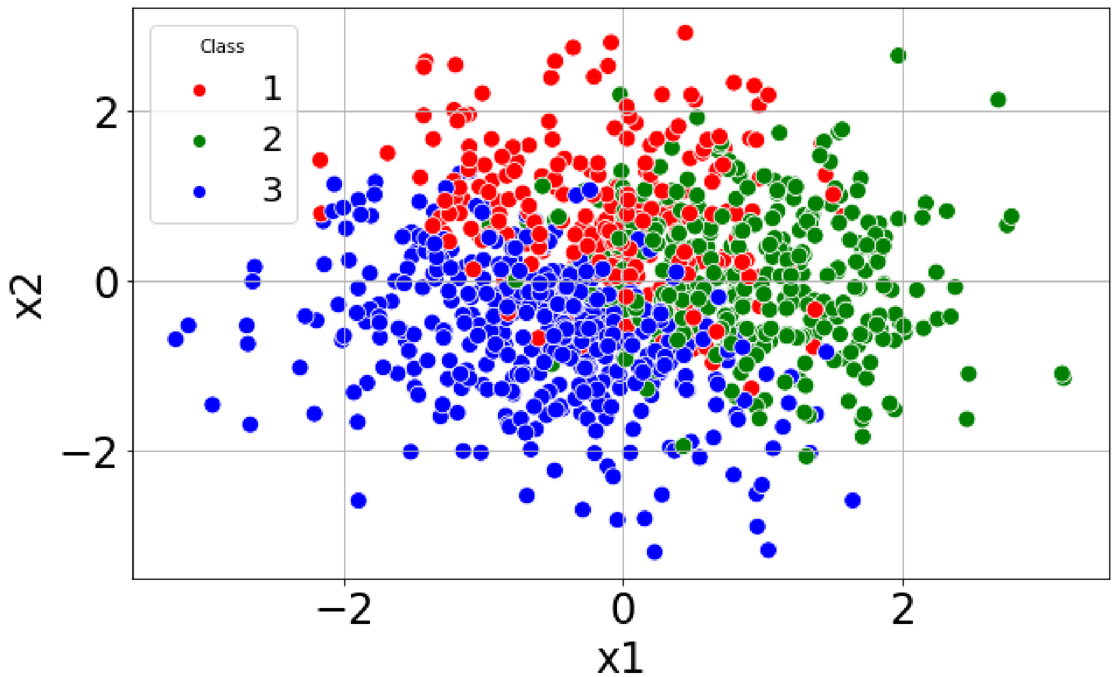
[ ]:
```
prob
```

```
array([[4.07444990e-02, 9.58817593e-01, 4.37907781e-04],
       [8.18778050e-02, 9.20496092e-02, 8.26072586e-01],
       [1.72976848e-01, 6.62731473e-01, 1.64291679e-01],
       …,
       [5.71793664e-04, 1.10591260e-04, 9.99317615e-01],
       [1.63646749e-01, 8.31431825e-01, 4.92142692e-03],
       [5.15228144e-02, 7.85390641e-02, 8.69938121e-01]])
```

[ ]:
```
y = np.array([np.random.multinomial(1, p) for p in prob])
y
```

```
array([[0, 1, 0],
       [0, 0, 1],
       [0, 0, 1],
       …,
       [0, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])
```

[ ]:
```
# Visualization

# Encode to 1, 2, and 3
def class_encode(x):
    if x[0] == 1:
        return 1
    elif x[1] == 1:
        return 2
    else:
        return 3

y_encode = np.apply_along_axis(class_encode, 1, y)
y_encode = pd.Categorical(y_encode, categories=[1, 2, 3])

# Plotting
plt.figure(figsize=(10, 6))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y_encode, palette=['red', 'green',␣
↪'blue'], s=100)
plt.xlabel('x1', fontsize=24)
plt.ylabel('x2', fontsize=24)
plt.tick_params(axis='both', labelsize=24)
plt.legend(title='Class', fontsize=20)
plt.grid()
plt.show()
```



0.10.2 Now let's fit multiclass logistic regression!

```
[ ]: def multLogReg(X, y, b_init=None, lambda_=0, tol=1e-4, inner_iter=100,␣
     ↪outer_iter=100):
         # Data dimension
         N, P = X.shape
         K = y.shape[1]

         # Start solving

         X = np.hstack((np.ones((N, 1)), X)) # Add intercept
         XtX = X.T @ X
         Lambda = np.diag([0] + [lambda_] * P)

         # Initialize coefficients
         if b_init is None:
             b = np.zeros((P + 1, K - 1)) # Allow different initialization of beta0
         else:
             b = b_init

         b_old = b.copy()
         f = np.exp(X @ b_old)
         prob = f / (1 + np.sum(f, axis=1, keepdims=True))
         # prob = 1 / (1 + np.exp(-np.dot(x, beta_old)))
```

```
    for out in range(outer_iter):  # Outer iteration
        for l in range(K - 1):
            for inner in range(inner_iter):  # Inner iteration
                b_tmp = b[:, l].copy()  # Coefficients of l-th class
                z = 0.25 * X @ b_tmp + (y[:, l] - prob[:, l])  # Use last class␣
↪as baseline
                b[:, l] = np.linalg.solve(0.25 * XtX + Lambda, X.T @ z)

                if np.max(np.abs(b_tmp - b[:, l])) / np.linalg.norm(b[:, l]) <␣
↪tol:
                    break

                f[:, l] = np.exp(X @ b[:, l])
                prob = f / (1 + np.sum(f, axis=1, keepdims=True))

                # Safe guard
                prob = np.clip(prob, 0.001, 0.999)

        if np.max(np.abs(b_old - b)) / np.linalg.norm(b) < tol:
            break

        b_old = b.copy()

    return b
```

[ ]:
```
b_hat = multLogReg(X, y)
b_hat
```

```
array([[-0.03407293,  0.01851146],
       [ 2.07595444,  3.81413738],
       [ 3.42020967,  2.52251442]])
```

Remember that for $k = 1, \dots, K-1$,

$$\log\left(\frac{\Pr(Y=k \mid X=x)}{\Pr(Y=K \mid X=x)}\right) = \beta_{k0} + \beta_{k1}x_1 + \dots + \beta_{kp}x_p.$$

Let $\frac{\Pr(Y=k|X=x)}{\Pr(Y=K|X=x)} = 1$, we can get

$$\beta_{k0} + \beta_{k1}x_1 + \dots + \beta_{kp}x_p = 0.$$

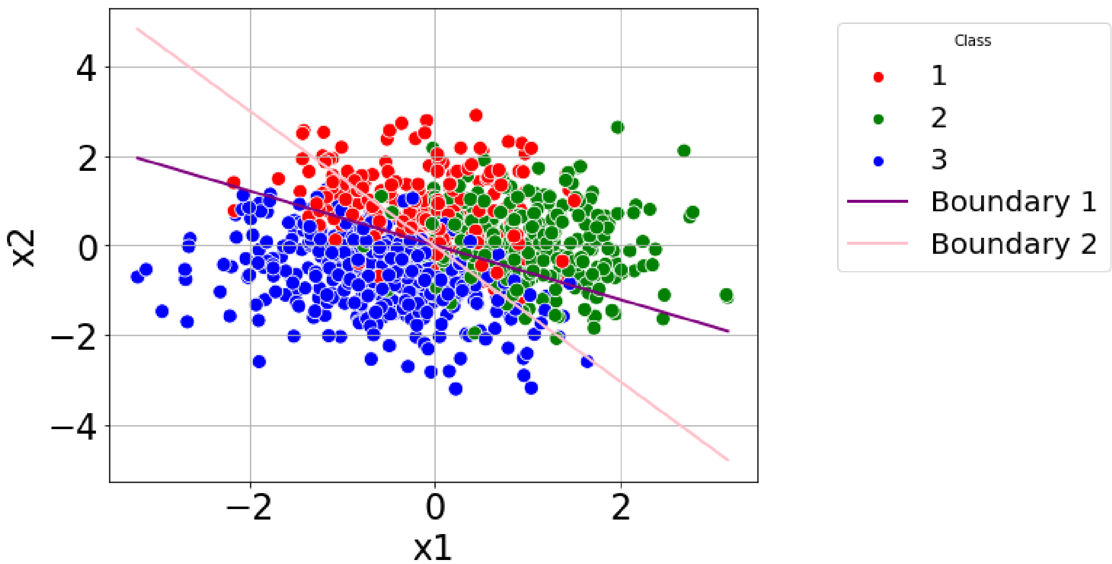
In this case $p = 2$, so

$$\beta_{k0} + \beta_{k1}x_1 + \beta_{k2}x_2 = 0, x_2 = -\frac{\beta_{k0}}{\beta_{k2}} - \frac{\beta_{k1}}{\beta_{k2}}x_1.$$

Suppose we do not know which is the **baseline** class and assume it was chosen at random. (Actually we know it.)

Let's first visualize the known classification line.

[ ]:
```python
# Plotting with decision boundaries
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y_encode, palette=['red', 'green', 'blue'], s=100)
x_vals = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)

# Decision boundaries
for i in range(K - 1):
    y_vals = (-b_hat[0, i] - b_hat[1, i] * x_vals) / b_hat[2, i]
    plt.plot(x_vals, y_vals, linewidth=2, label=f'Boundary {i + 1}', color=['purple', 'pink'][i])

plt.xlabel('x1', fontsize=24)
plt.ylabel('x2', fontsize=24)
plt.tick_params(axis='both', labelsize=24)
plt.legend(title='Class', fontsize=20, bbox_to_anchor = (1.1,1))
plt.grid()
plt.show()
```



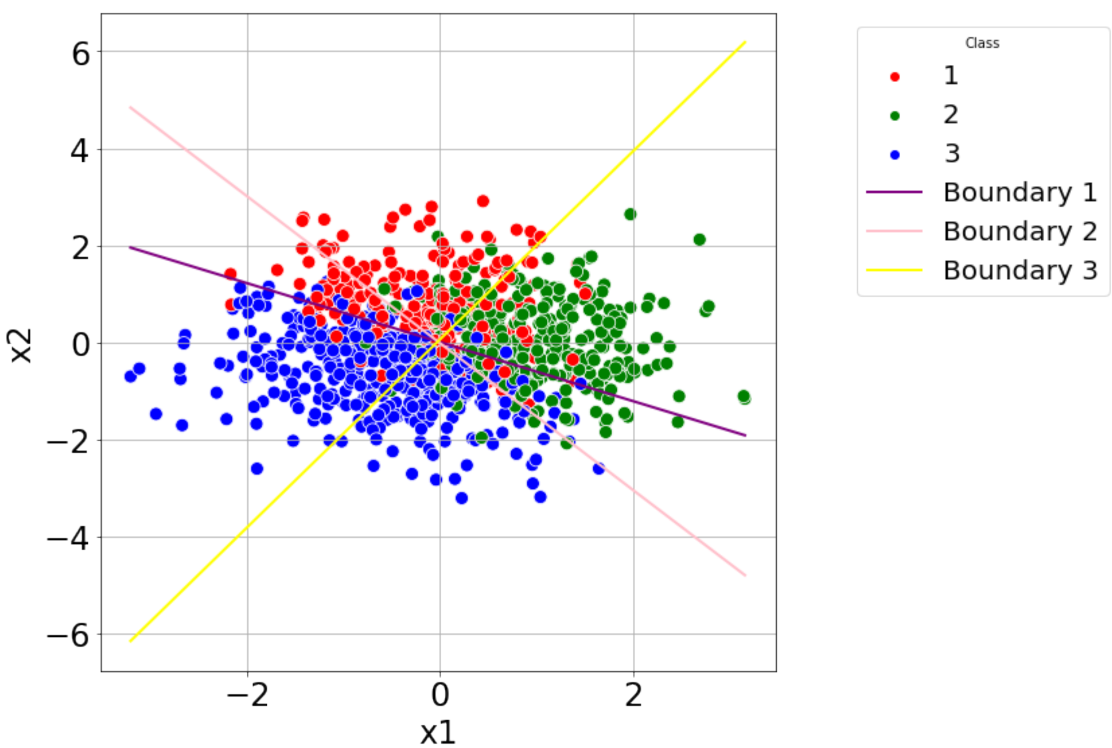
We can see that the purple line is between class 1 and 3, and the pink line is between class 2 and 3. So the **baseline** here is class 3, and

$$\log\left(\frac{\Pr(Y=1\mid X=x)}{\Pr(Y=3\mid X=x)}\right) = -0.03+2.08x_1+3.42x_2,\ \log\left(\frac{\Pr(Y=2\mid X=x)}{\Pr(Y=3\mid X=x)}\right) = 0.02+3.81x_1+2.52x_2.$$

Therefore,

$$\log\left(\frac{\Pr(Y=1\mid X=x)}{\Pr(Y=2\mid X=x)}\right) = (-0.03-0.02)+(2.08-3.81)x_1+(3.42-2.52)x_2.$$

[ ]:
```
# # Full figure with additional decision boundary
plt.figure(figsize=(9, 9))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=y_encode, palette=['red', 'green',␣
↪'blue'], s=100)
x_vals = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)

# Decision boundaries
for i in range(K - 1):
    y_vals = (-b_hat[0, i] - b_hat[1, i] * x_vals) / b_hat[2, i]
    plt.plot(x_vals, y_vals, linewidth=2, label=f'Boundary {i + 1}',␣
↪color=['purple', 'pink'][i])

# Additional boundary between class 1 and class 2
y_vals = (-(b_hat[0, 0] - b_hat[0, 1]) - (b_hat[1, 0] - b_hat[1, 1]) * x_vals) /
↪ (b_hat[2, 0] - b_hat[2, 1])
plt.plot(x_vals, y_vals, linewidth=2, color='yellow', label='Boundary 3')

plt.xlabel('x1', fontsize=24)
plt.ylabel('x2', fontsize=24)
plt.tick_params(axis='both', labelsize=24)
plt.legend(title='Class', fontsize=20, bbox_to_anchor = (1.1,1))
plt.grid()
plt.show()
```



```
[ ]:
```