In [1]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

Resolvi analisar brevemente a base de vinhos do UCI ML. Temos algumas características de vinhos tintos e brancos: atributos químicos e qualidade.

In [2]:

```
df_red = pd.read_csv('winequality-red.csv', sep=';')
df_white = pd.read_csv('winequality-white.csv', sep=';')

df_red['type']='r'
df_white['type']='w'
```

In [3]:

```
df = pd.concat([df_red.sample(n=200),df_white.sample(n=200)])
        .reset_index().drop(columns='index')
```

In [4]:

```
df.describe()
```

Out[4]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	d
count	400.000000	400.000000	400.00000	400.000000	400.000000	400.00000	400.000000	400.0
mean	7.591125	0.392175	0.31720	4.166625	0.065175	24.70000	91.207500	0.9
std	1.514327	0.176994	0.16443	3.902097	0.036972	16.05114	57.153747	0.0
min	4.900000	0.120000	0.00000	0.900000	0.021000	1.00000	7.000000	0.9
25%	6.600000	0.250000	0.24000	1.800000	0.041000	12.00000	35.000000	0.9
50%	7.200000	0.350000	0.31000	2.300000	0.057000	22.00000	94.000000	0.9
75%	8.200000	0.512500	0.41000	5.100000	0.080250	34.00000	132.000000	0.9
max	13.300000	1.005000	1.00000	18.900000	0.422000	87.00000	256.000000	1.0

In [5]:

```
df.columns
```

Out[5]:

```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'densit
y',
       'pH', 'sulphates', 'alcohol', 'quality', 'type'],
      dtype='object')
```

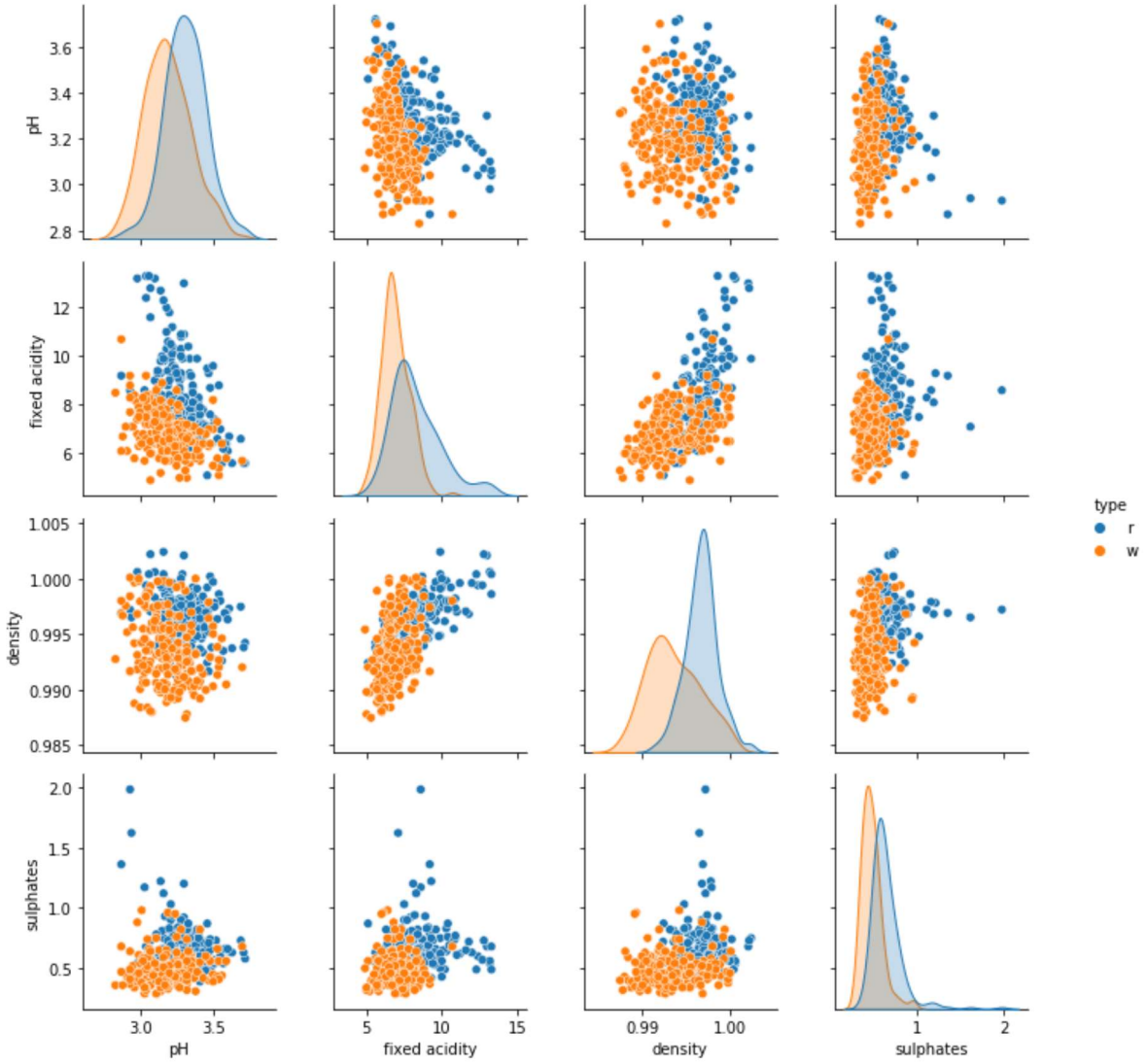
In [6]:

```
columns = ['pH', 'fixed acidity', 'density', 'sulphates']
```

Olhando a distribuição conjunta de algumas variáveis continuas. Na diagonal temos a distribuição (univariada) delas (estimadas pelo KDE). A cor nos indica de que tipo o vinho é: vermelho (r) ou branco (w).

In [7]:

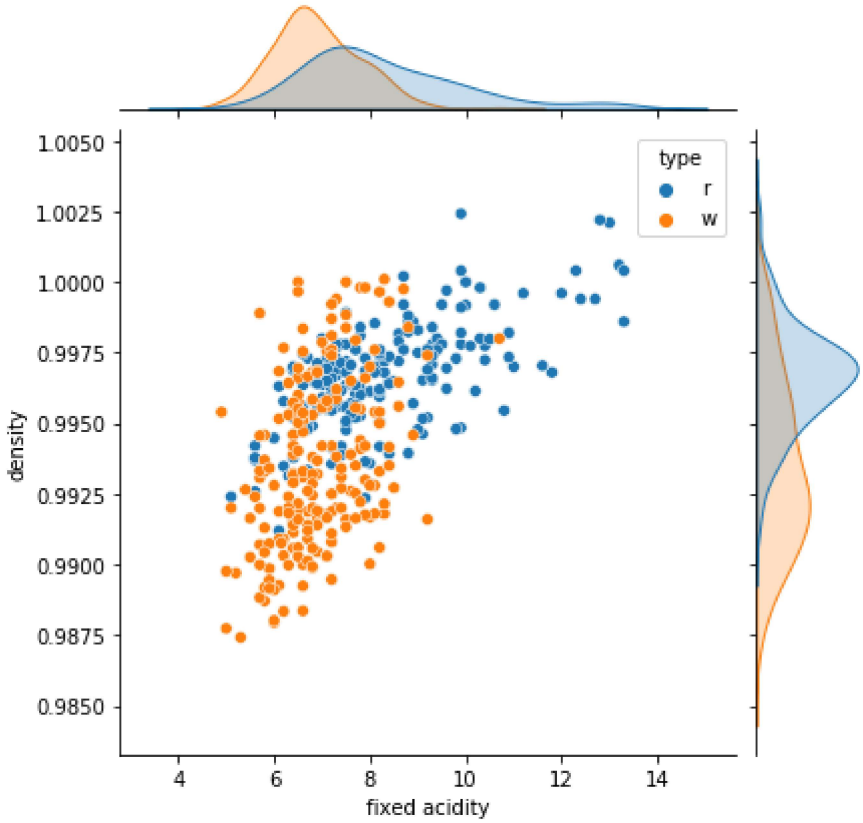
```
sns.pairplot(df[columns+['type']], hue="type", height=2.5)
plt.show()
```



Se quisermos usar duas dessas variáveis para prever o tipo do vinho poderíamos querer olhar para a density e fixed acidity. Um gráfico maior nos permite analisar melhor como essas variáveis se relacionam em cada um dos tipos de vinho.

In [8]:

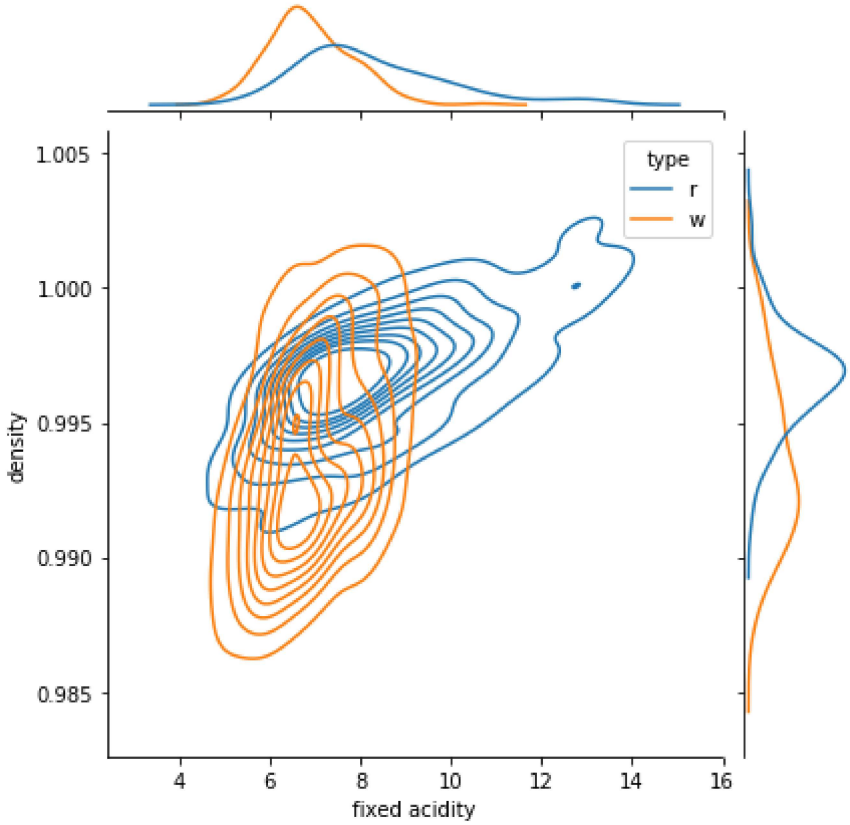
```
sns.jointplot(data=df, x="fixed acidity", y="density", hue="type")
plt.show()
```



Muitas vezes um scatter plot não nos permite entender com precisão a distribuição dos pontos. É o caso aqui pela sobreposição. Neste caso podemos usar o um estimador de densidade multivariado e plotar as curvas de nível. Assim conseguimos ver as diferenças nas distribuições conjuntas em cada um dos tipos de vinho.

In [9]:

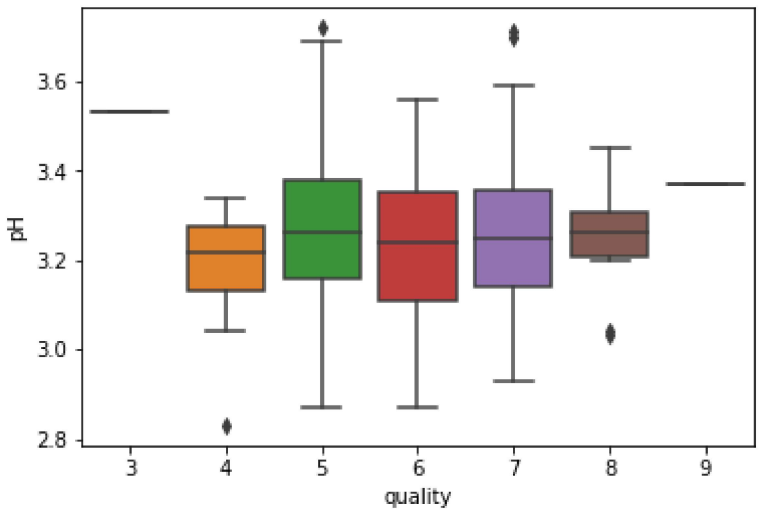
```
sns.jointplot(data=df, x="fixed acidity", y="density", hue="type", kind="kde")
plt.show()
```

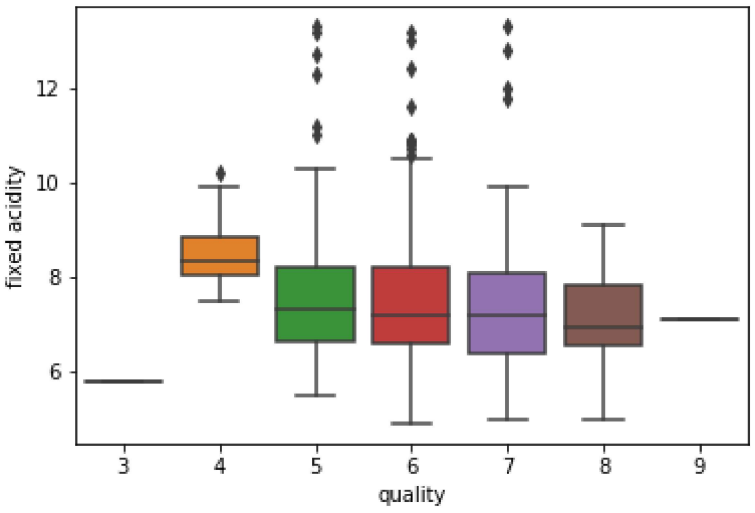


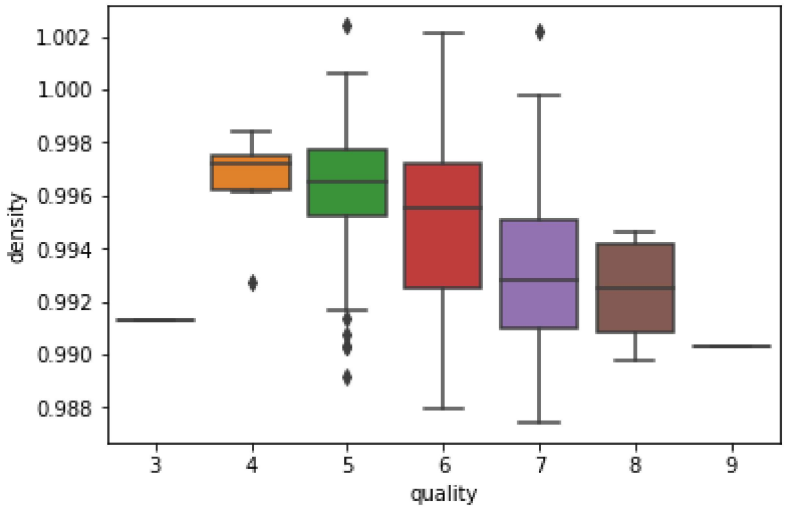
Estamos interessados ainda em entender como alguma das variáveis se relaciona com a qualidade. Neste caso podemos fazer boxplot em que colocamos no eixo x os diferentes valores de qualidade do dataset e no eixo y vemos o boxplot para coluna naquele subconjunto de vinhos com qualidade fixada.

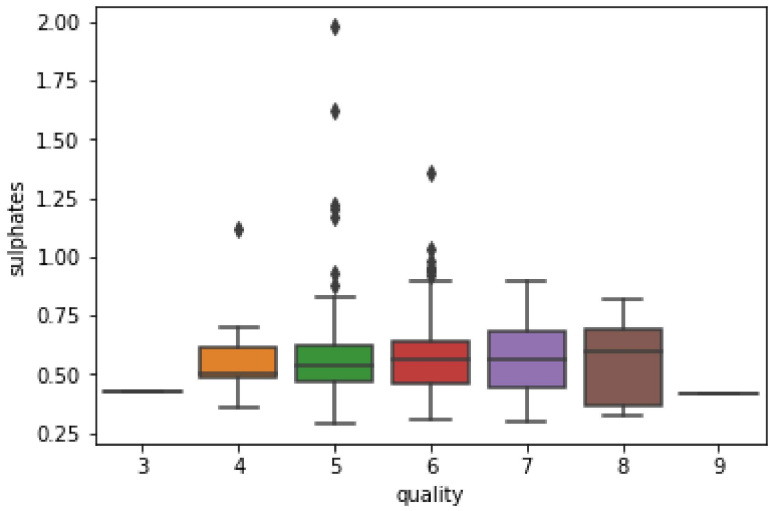
In [10]:

```
for coluna in columns:
    sns.boxplot(data=df, x="quality", y=coluna)
    plt.show()
```






density
1.002
1.000
0.998
0.996
0.994
0.992
0.990
0.988
3
4
5
6
7
8
9
quality


sulphates
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
3
4
5
6
7
8
9
quality

LDA e QDA no API do sklearn

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn as sk
import matplotlib

from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.utils import as_float_array, check_X_y, check_array
from sklearn.utils.validation import check_is_fitted

from sklearn.metrics import accuracy_score

print("sklearn versão:", sk.__version__)
print("numpy versão:", np.__version__)
print("pandas versão:", pd.__version__)
print("matplotlib versão:", matplotlib.__version__)
```

```
sklearn versão: 0.23.2
numpy versão: 1.16.2
pandas versão: 1.0.5
matplotlib versão: 3.0.3
```

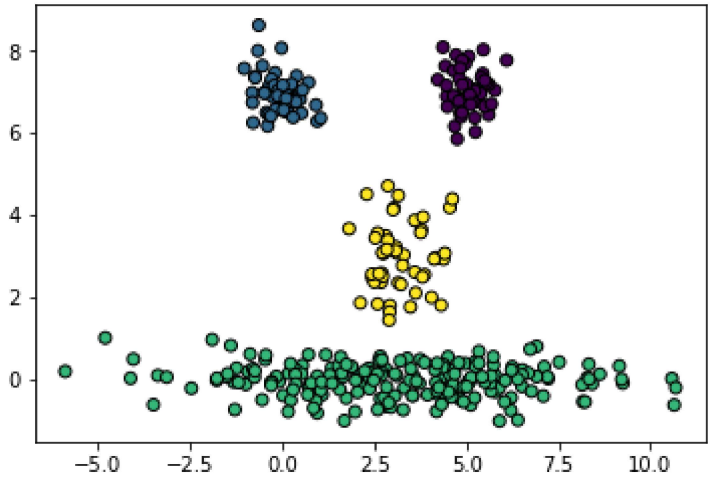
In [2]:

```
df = pd.read_csv('uma_base.csv')

X = np.array(df.drop(columns='c'))
y = np.array(df['c'])
```

In [3]:

```
plt.scatter(X[:,0],X[:,1], edgecolors='k', c=[*map(lambda x: int(x[-1]), y),])
plt.show()
```



In [5]:

```
# "título" sem _ e cada palavra nova começa com maiusculo

# funções
# público. sem _ na frente
# modo protegido. nao aparece de primeira mas acessável quando da lda._ -> _ na frente
# modo privado. não acessivel -> __ na frente

# mesma coisa vale para os atributos da classe!
# em outras linguagens declaro na mão: publico tal privado tal

# no is_fitted posso passar atributos que ele pode verificar se ja foram calculados
# ['cov_', 'inv_cov_', 'mean_'])

class DiscriminantAnalysis(BaseEstimator, ClassifierMixin):

    def __init__(self, tipo='Linear'):
        self.tipo = tipo

    def fit(self, X, y):

        # verificando se a entrada faz sentido
        check_X_y(X, y)
        X = as_float_array(X)

        self.possible_values_ = np.unique(y)
        self.count_values_ = [(y == value).sum() for value in self.possible_values_]

        # calculando a matriz de covariância de cada classe
        cov_matrix = []
        for classe in self.possible_values_:
            cov_matrix.append(np.cov(np.array(X[y==classe]).T))

        # se usamos o tipo linear, agrupamos em um valor só
        if self.tipo == 'Linear':
            cov_matrix = [(tamanho-1)*matrix for matrix, tamanho in
                          zip(cov_matrix,self.count_values_)]
            self.cov_ = [sum(cov_matrix)/(X.shape[0]
                        -len(self.possible_values_))]*len(self.possible_values_)
            self.inv_cov_ = [np.linalg.inv(matriz) for matriz in self.cov_]

        if self.tipo == 'Quadrática':
            self.cov_ = cov_matrix
            self.inv_cov_ = [np.linalg.inv(matriz) for matriz in self.cov_]

        # calculando a média dos atributos em cada classe
        self.mean_ = []
        for classe in self.possible_values_:
            self.mean_.append(np.mean(X[y==classe],axis=0))
        return self

    def _calc_discrim(self, X):

        # calcula os discriminantes de cada classe
        g_ = []
        for i, tam_classe in enumerate(self.count_values_):
            if self.tipo == 'Linear':
                g_.append(np.matmul(np.matmul(X,self.inv_cov_[i]),self.mean_[i].T)
                          - 0.5*np.matmul(np.matmul(self.mean_[i].T,self.inv_cov_[i]),
                                          self.mean_[i])
```

```
                        + np.log(tam_classe/sum(self.count_values_)))

            if self.tipo == 'Quadrática':
                g_.append(-0.5*np.matmul(np.matmul(X,self.inv_cov_[i]),X.T).diagonal() +
                          np.matmul(np.matmul(self.mean_[i].T,self.inv_cov_[i]),X.T) -
                          0.5*np.matmul(np.matmul(self.mean_[i].T,self.inv_cov_[i]),
                                        self.mean_[i])
                          - 0.5*np.log(np.linalg.det(self.cov_[i])) +
                          np.log(tam_classe/sum(self.count_values_)))
        return g_

    def predict(self, X):

        # verificando se já foi fitado
        check_is_fitted(self,  None)

        # verificando se a entrada faz sentido
        check_array(X)
        X = as_float_array(X)

        # calculando discriminantes
        g_ = self._calc_discrim(X)
        predict_ = []
        # retornando a classe com maior discriminante
        for i in range(X.shape[0]):
            aux = np.array([g_[j][i] for j in range(len(self.possible_values_))])
            predict_.append(self.possible_values_[np.argmax(aux)])
        return np.asarray(predict_)

    def predict_proba(self, X):

        # verificando se já foi fitado
        check_is_fitted(self, None)

        # verificando se a entrada faz sentido
        check_array(X)
        X = as_float_array(X)

        # calculando discriminantes
        g_ = self._calc_discrim(X)
        predict_ = []
        # soma os discriminantes para retornar uma "probabilidade"
        for i in range(X.shape[0]):
            aux = np.array([g_[j][i] for j in range(len(self.possible_values_))])
            predict_.append(aux/np.sum(aux))
        return np.asarray(predict_)
```

Linear

In [6]:

```
lda = DiscriminantAnalysis(tipo='Linear')
lda.fit(X,y)
lda_predict = lda.predict(X)
```

In [7]:

```
accuracy_score(lda_predict,y)
```

Out[7]:

```
0.9971428571428571
```

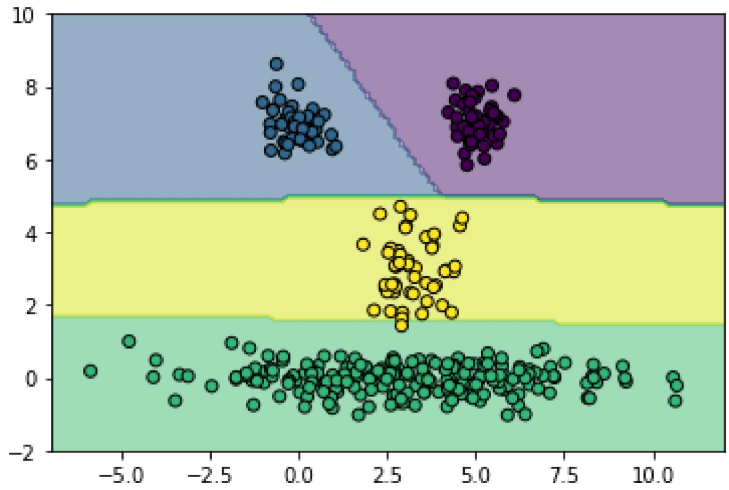
In [8]:

```
resolucao = 100
x_vals = np.linspace(-7, 12, resolucao)
y_vals = np.linspace(-2, 10, resolucao)

Xgrid, Ygrid = np.meshgrid(x_vals, y_vals)
Z = np.array([*map(lambda x: int(x[-1]), lda.predict(
    np.column_stack((Xgrid.reshape(-1,1),Ygrid.reshape(-1,1))))),])
.reshape(resolucao,resolucao)
```

In [9]:

```
plt.contourf(Xgrid, Ygrid, Z, alpha = 0.5)
plt.scatter(X[:,0],X[:,1], edgecolors='k', c=[*map(lambda x: int(x[-1]), y),])
plt.show()
```



Quadrática

In [10]:

```
qda = DiscriminantAnalysis(tipo='Quadrática')
qda.fit(X,y)
qda_predict = qda.predict(X)
```

In [11]:

```
accuracy_score(qda_predict,y)
```

Out[11]:

```
1.0
```

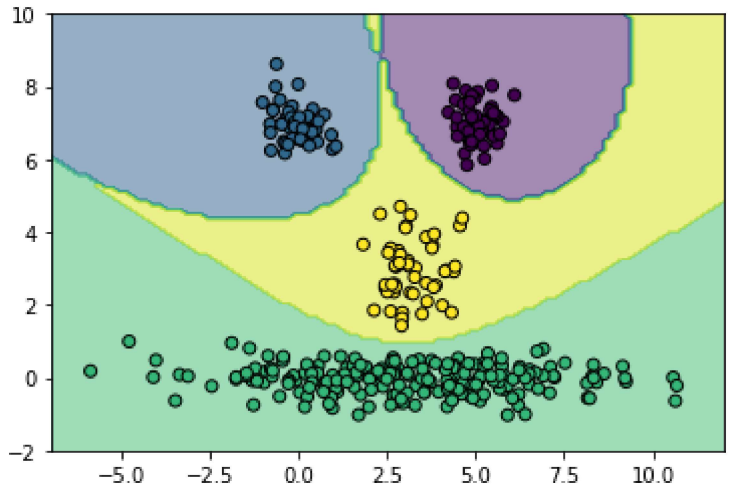
In [12]:

```
resolucao = 100
x_vals = np.linspace(-7, 12, resolucao)
y_vals = np.linspace(-2, 10, resolucao)

Xgrid, Ygrid = np.meshgrid(x_vals, y_vals)
Z = np.array([*map(lambda x: int(x[-1]), qda.predict(
    np.column_stack((Xgrid.reshape(-1,1),Ygrid.reshape(-1,1))))),])
.reshape(resolucao,resolucao)
```

In [13]:

```
plt.contourf(Xgrid, Ygrid, Z, alpha = 0.5)
plt.scatter(X[:,0],X[:,1], edgecolors='k', c=[*map(lambda x: int(x[-1]), y),])
plt.show()
```



In [14]:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDiscriminant
```

In [15]:

```
y_ = np.array([*map(lambda x: int(x[-1]), y),])
```

Teste LDA

In [16]:

```
lda_ = LinearDiscriminantAnalysis(solver='lsqr')
```

In [17]:

```
lda_.fit(X,y_)
```

Out[17]:

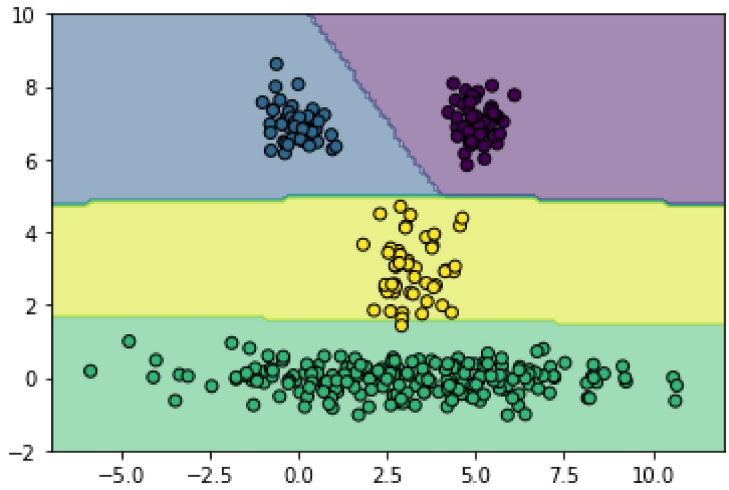
```
LinearDiscriminantAnalysis(solver='lsqr')
```

In [18]:

```
Z_ = lda_.predict(np.column_stack(
    (Xgrid.reshape(-1,1),Ygrid.reshape(-1,1)))).reshape(resolucao,resolucao)
```

In [19]:

```
plt.contourf(Xgrid, Ygrid, Z_, alpha = 0.5)
plt.scatter(X[:,0],X[:,1], edgecolors='k', c=[*map(lambda x: int(x[-1]), y),])
plt.show()
```



Teste QDA

In [20]:

```
qda_ = QuadraticDiscriminantAnalysis(store_covariance=True)
qda_.fit(X,y_)
Z_ = qda_.predict(np.column_stack(
    (Xgrid.reshape(-1,1),Ygrid.reshape(-1,1)))).reshape(resolucao,resolucao)
```

In [21]:

```
plt.contourf(Xgrid, Ygrid, Z_, alpha = 0.5)
plt.scatter(X[:,0],X[:,1], edgecolors='k', c=[*map(lambda x: int(x[-1]), y),])
plt.show()
```