Redes Neurais Artificiais - Exercício 7

February 6, 2021

Aluno: Victor São Paulo Ruela

[1]:
```
%load_ext autoreload
%autoreload 2

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

from sklearn.metrics import confusion_matrix, accuracy_score,␣
↪mean_squared_error, explained_variance_score
from sklearn.preprocessing import MinMaxScaler
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.utils.validation import check_X_y, check_array, check_is_fitted
from sklearn.cluster import KMeans
from sklearn.metrics.pairwise import euclidean_distances

from mpl_toolkits.mplot3d import Axes3D
```

1 Benchmark RBFs

1.1 Problemas de Classificação

Inicialmente, carrega-se os dados para as funções indicadas no enunciado. Eles foram gerados utilizando o pacote R indicado e exportados para arquivos CSV.

[2]:
```
normals = pd.read_csv('2dnormals.csv')
xor = pd.read_csv('xor.csv')
circle = pd.read_csv('circle.csv')
spirals = pd.read_csv('spirals.csv')

# map classes as -1 or 1
normals['classes'] = normals['classes'].map({2:1, 1:-1})
xor['classes'] = xor['classes'].map({2:1, 1:-1})
circle['classes'] = circle['classes'].map({2:1, 1:-1})
```

```
spirals['classes'] = spirals['classes'].map({2:1, 1:-1})
```

A seguir, é feita a implementação do algoritmo do RBF com centros e raios escolhidos pelo k-means.

[15]:
```
class RBF(BaseEstimator):
    def __init__(self, p, center_estimation='kmeans'):
        self.p = p
        self.center_estimation = center_estimation

    def apply_transformation(self, X):
        check_is_fitted(self, ['cov_', 'centers_'])
        N, n = X.shape
        H = np.zeros((N, self.p))
        for j in range(N):
            for i in range(self.p):
                mi = self.centers_[i, :]
                covi = self.cov_[i] + 0.001 * np.eye(n)
                H[j, i] = self.gaussian_kernel(X[j, :], mi, covi, n)
        return H

    def predict(self, X):
        # check X, y consistency
        X = check_array(X, accept_sparse=True)
        check_is_fitted(self, ['cov_', 'centers_', 'H_', 'coef_'])
        N, _ = X.shape

        H = self.apply_transformation(X)
        H_aug = np.hstack((np.ones((N, 1)), H))
        yhat = H_aug @ self.coef_

        return yhat

    def gaussian_kernel(self, X, m, K, n):
        if n == 1:
            r = np.sqrt(float(K))
            px = (1/(np.sqrt(2*np.pi*r*r)))*np.exp(-0.5 * (float(X-m)/r)**2)
            return px
        else:
            center_distance = (X - m).reshape(-1, 1)
            normalization_factor = np.sqrt(((2*np.pi)**n)*sp.linalg.det(K))
            dist = float(
                np.exp(-0.5 * (center_distance.T @ (sp.linalg.inv(K)) @␣
↪center_distance)))
            return dist / normalization_factor

    def make_centers(self, X):
        if(self.center_estimation == 'kmeans'):
```

```
            kmeans = KMeans(n_clusters=self.p).fit(X)
            self.centers_ = kmeans.cluster_centers_
            # estimate covariance matrix for all centers
            clusters = kmeans.predict(X)
            covlist = []
            _, n = X.shape
            for i in range(self.p):
                xci = X[clusters == i, :]
                covi = np.cov(xci, rowvar=False) if n > 1 else np.asarray(np.
↪var(xci))
                covlist.append(covi)
            self.cov_ = covlist
        else:
            raise ValueError

    def fit(self, X, y):
        # check X, y consistency
        X, y = check_X_y(X, y, accept_sparse=True)
        N, _ = X.shape

        # define centers
        self.make_centers(X)
        # calculate H matrix
        H = self.apply_transformation(X)

        H_aug = np.hstack((np.ones((N, 1)), H))
        self.coef_ = (sp.linalg.inv(H_aug.T @ H_aug) @ H_aug.T) @ y
        self.H_ = H

        return self
```

Em seguida, é criada uma rotina que recebe um conjunto de dados de entrada e desenha a sua superfície de separação conforme a sugestão do enunciado do exercício.

[16]:
```
def plot_decision_boundary(data, p=5, plot_error=False):
    fig = plt.figure(figsize=(10,4))
    ax1 = fig.add_subplot(1, 2, 1)
    ax2 = fig.add_subplot(1, 2, 2, projection='3d')

    x1 = np.arange(np.min(data['x.1']) - 1, np.max(data['x.1']) + 1, step=0.1)
    x2 = np.arange(np.min(data['x.2']) - 1, np.max(data['x.2']) + 1, step=0.1)

    xx, yy = np.meshgrid(x1, x2)
    # flatten each grid to a vector
    r1, r2 = xx.flatten(), yy.flatten()
    r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
```

```
    # horizontal stack vectors to create x1,x2 input for the model
    grid = np.hstack((r1,r2))

    # extract the data
    X, y = data[['x.1', 'x.2']].to_numpy(), data['classes'].to_numpy()

    # train the model
    model = RBF(p=p).fit(X, y)

    # make predictions for the grid
    yhat = np.sign(model.predict(grid))
    # reshape the predictions back into a grid
    zz = yhat.reshape(xx.shape)
    ax1.contour(xx, yy, zz, colors=['black'])

    t_class0 = data['classes'] == -1
    t_class1 = data['classes'] == 1
    ax1.scatter(data.loc[t_class0, 'x.1'],
                data.loc[t_class0, 'x.2'], color='red')
    ax1.scatter(data.loc[t_class1, 'x.1'], data.loc[t_class1, 'x.2'],␣
↪color='blue')
    ax1.set_xlabel('x1')
    ax1.set_ylabel('x2')
    fig.suptitle(f'Neurônios:{p}\nAccurácia: {100 * np.sum(y == np.sign(model.
↪predict(X)))/len(y)} %')

    surf = ax2.plot_surface(xx, yy, zz, cmap='jet')
    fig.tight_layout()
    fig.show()
```

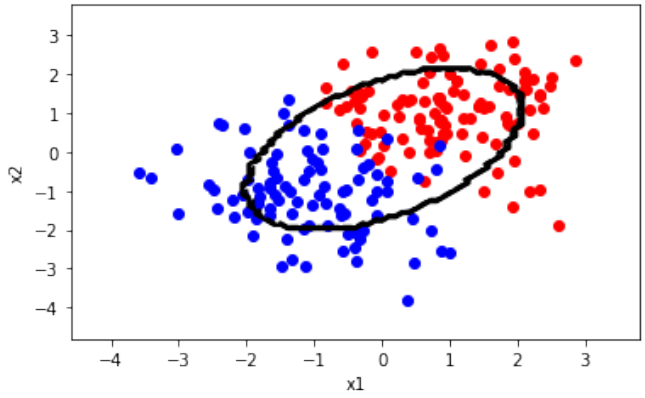
1.2 Análise dos Resultados

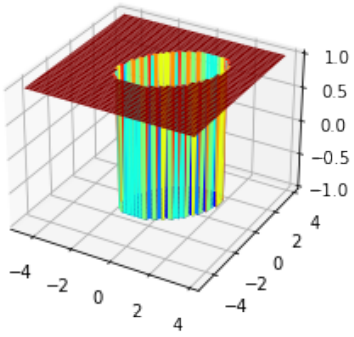
Para cada base de dados, são geradas superfícies de decisão considerando 1, 5, 10 e 30 neurônios. Os resultados são discutidos a seguir.

1.2.1 Base de dados `2dnormals`

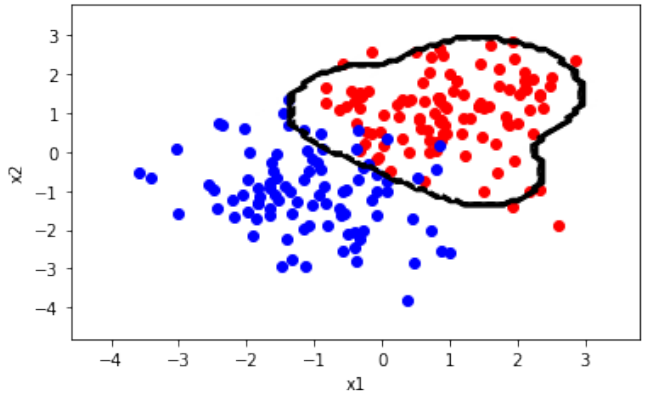
[17]:
```
# define the neurons array:
neurons = [1, 5, 10, 30]
# 2dnormals
for p in neurons:
    plot_decision_boundary(normals, p=p)
```

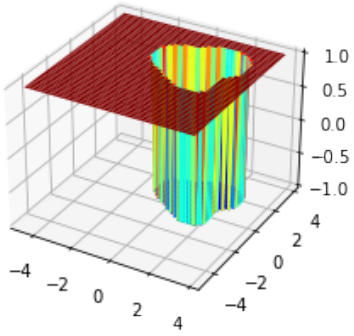
Neurônios:1
Accurácia: 56.0 %



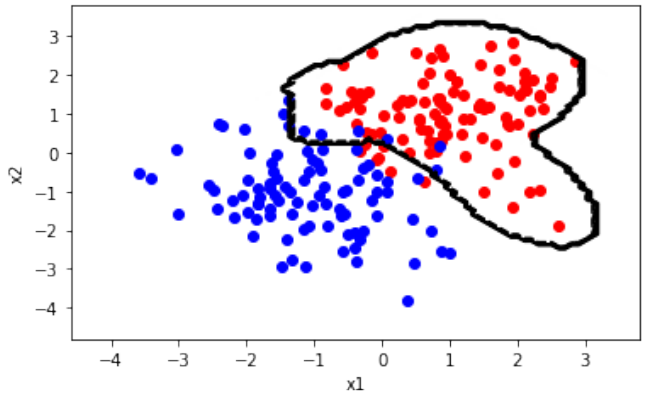


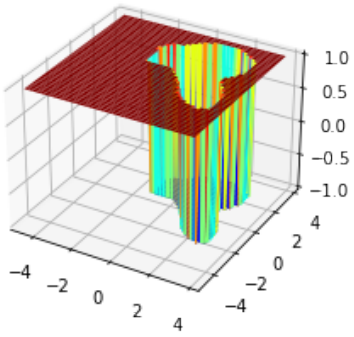
Neurônios:5
Accurácia: 92.5 %

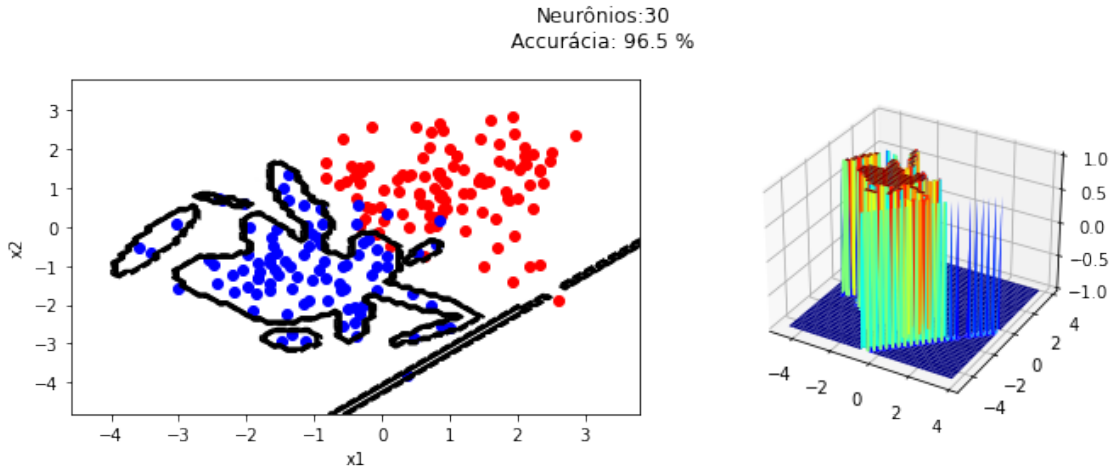




Neurônios:10
Accurácia: 93.5 %



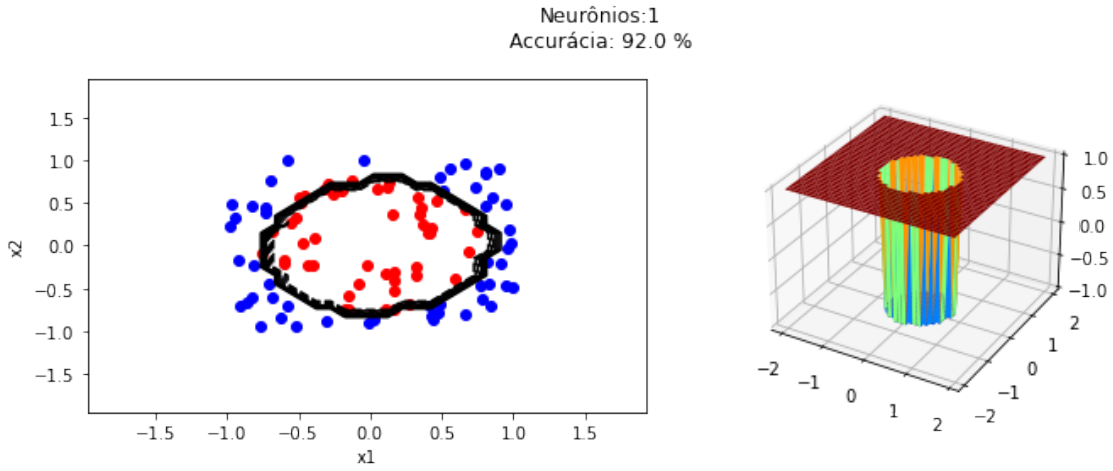




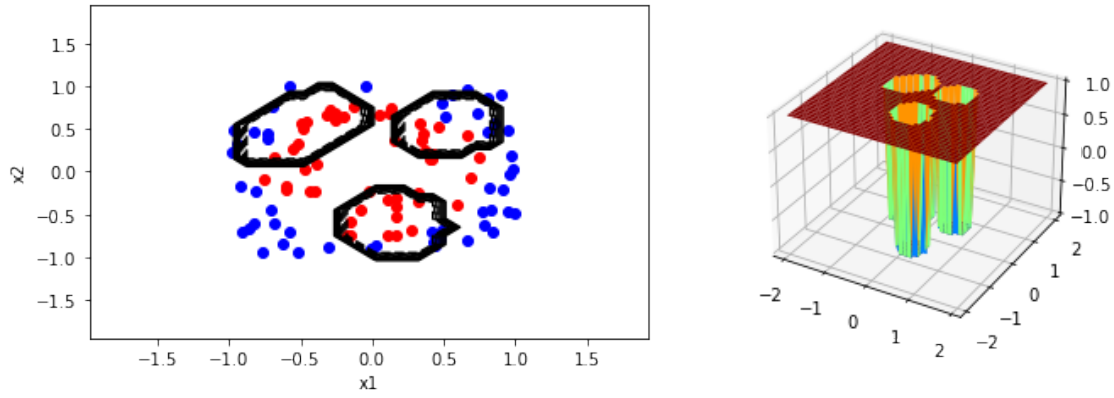
Através dos resultados acima, é possível notar que a soluçõe deste problema foi dificultada pela uso do RBF, uma vez que um modelo linear seria capaz de obter uma excelente generalização. O uso de um único neurônion não foi capaz de classificar com qualidade os padrões, sendo necessário 5 para se obter um superfício de separação com boa generalização. Nota-se que o aumento do número de neurônios leva a um possível overfitting sobre os dados, resultando em uma superfície de separação bastante irregular.

1.2.2 Base de dados `circle`

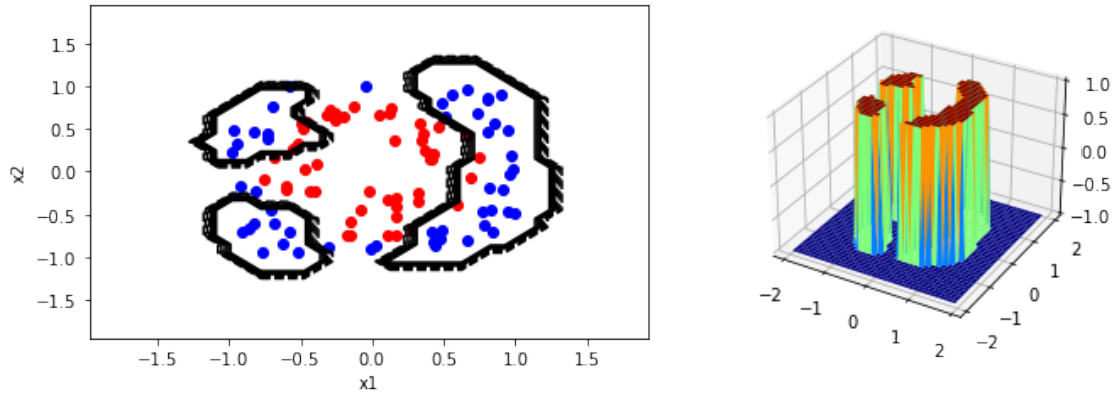
[18]:
```
# circle
for p in neurons:
    plot_decision_boundary(circle, p=p)
```



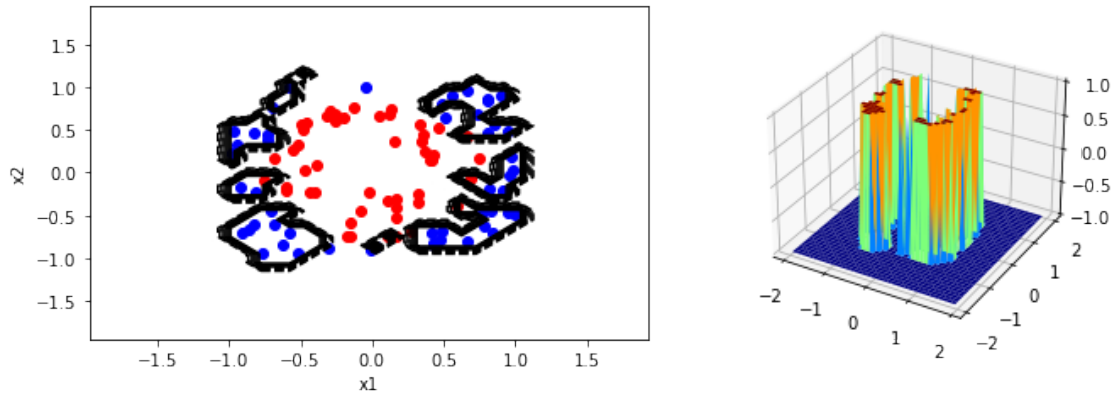
Neurônios:5
Accurácia: 67.0 %



Neurônios:10
Accurácia: 85.0 %



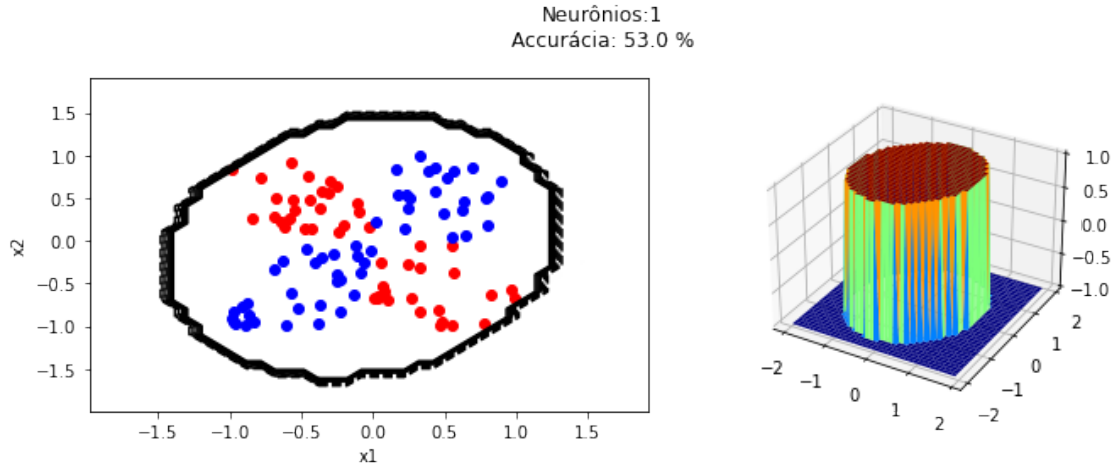
Neurônios:30
Accurácia: 90.0 %

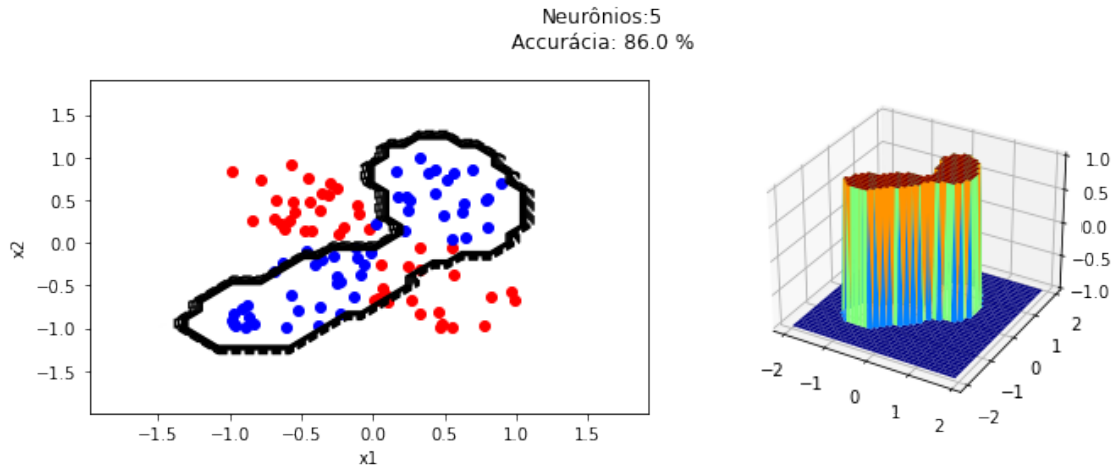


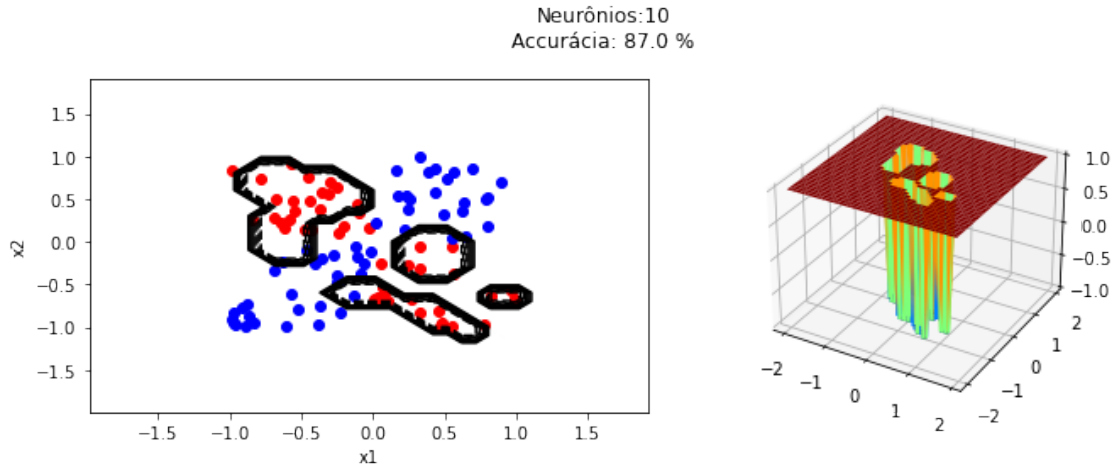
Devido à característica dessa base de dados, um neurônio foi suficiente para obter uma excelente aproximação da superfície de decisão. Isso mostra um resultado importante da rede RBF, que é a sua melhor adaptabilidade para lidar com problemas onde há um padrão circular nos dados. O uso de mais neurônios claramente leva a um overfitting aos dados, uma vez que a superfície de separação é bem irregular.

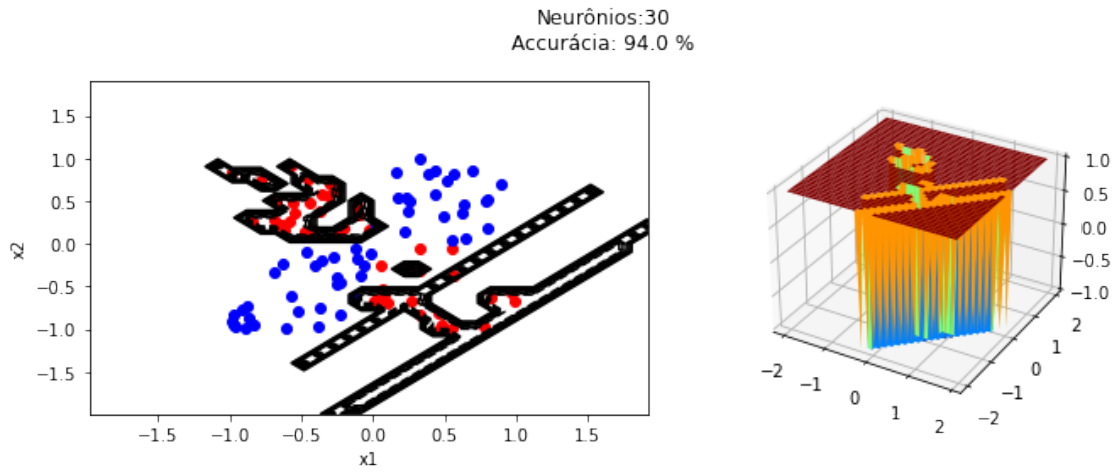
1.2.3 Base de dados `xor`

[19]:
```
# xor
for p in neurons:
    plot_decision_boundary(xor, p=p)
```







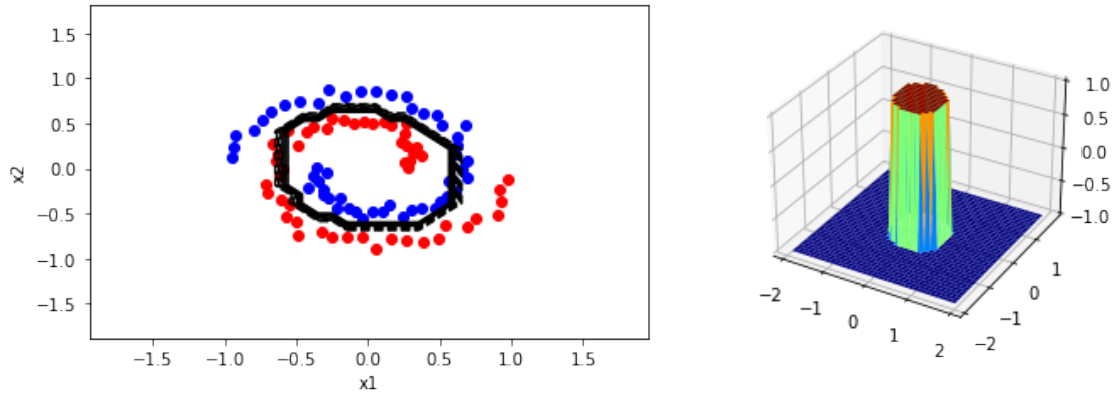


Para este modelo, nota-se que os melhores resultados foram obtidos em torno de 5 neurônios. O uso de 1 neurônio, se mostrou bastante inadequado, uma vez que os dados não estão distribuídos de forma circular. O uso de uma quantidade maior de neurônios foi capaz de aumentar a acurácia de treinamento, principalmente para uma classificação da região central dos dados, que é a mais difícil. Entretanto, é importante notar o aumento da complexidade da rede reduz a sua generalização consideravelmente.

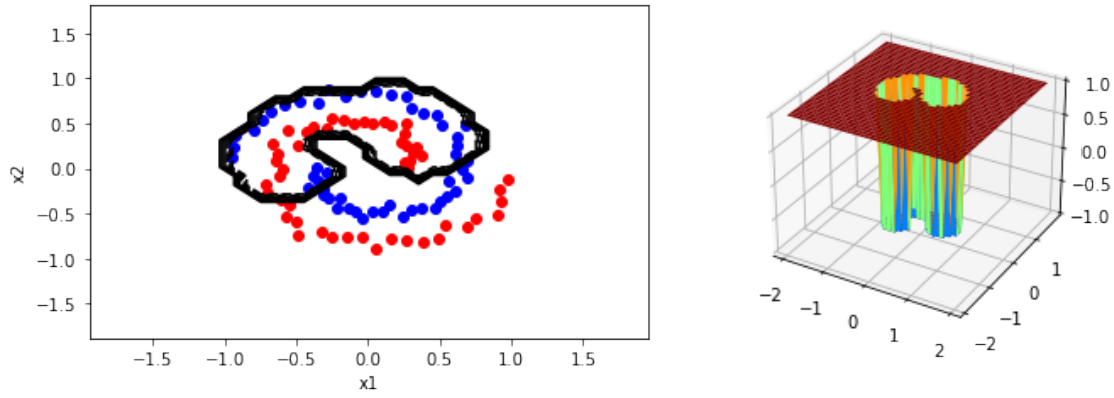
1.2.4 Base de dados `spirals`

[20]:
```
# spirals
for p in neurons:
    plot_decision_boundary(spirals, p=p)
```

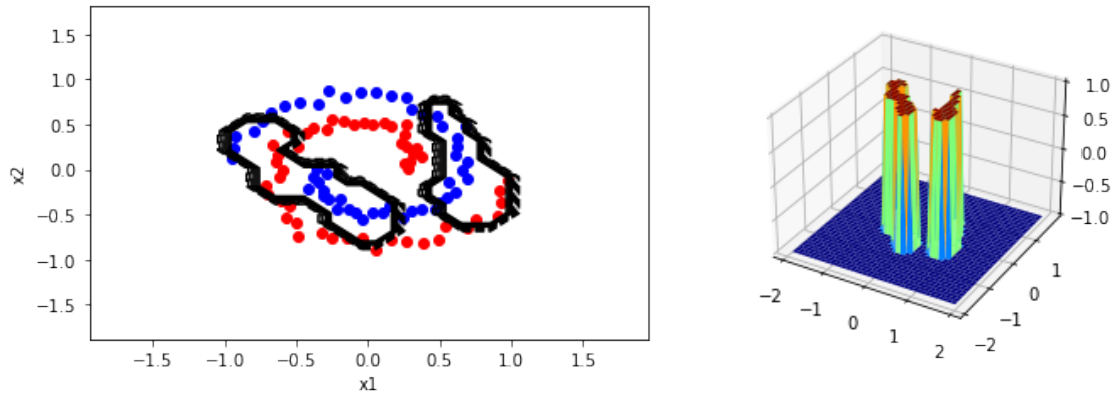
Neurônios:1
Accurácia: 49.0 %

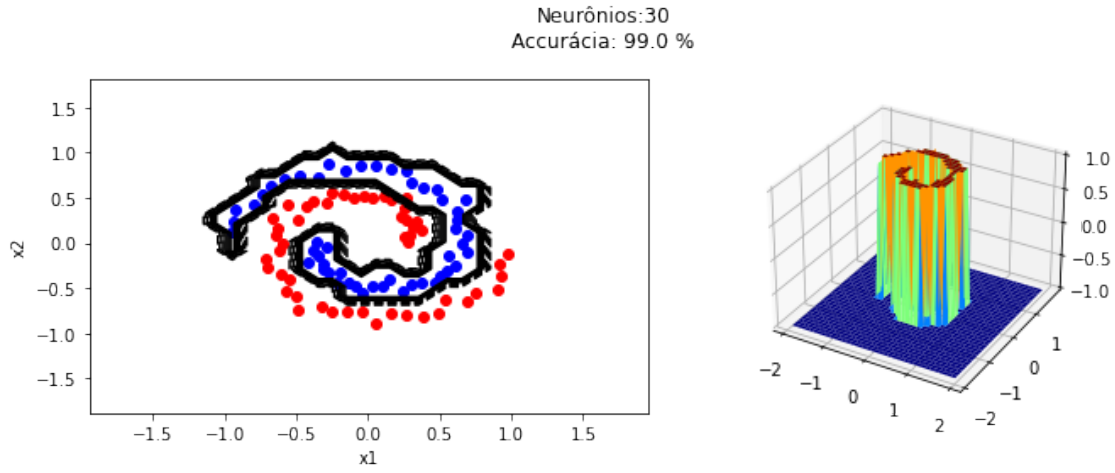


Neurônios:5
Accurácia: 50.0 %



Neurônios:10
Accurácia: 69.0 %





Conforme pode ser visto no gráfico, é necessária uma quantidade muito grande de neurônios para obter uma boa aproximação da superfície de separação destes dados. O uso de 1, 5 ou 10 neurônios não foi suficiente para obter uma boa aproximação. O uso de 30 neurônios obeteve os melhores resultados para esta base de dados, representando de forma bem generalizada a superfície de decisão.

1.3 Problema de Regressão

Inicialmente, geramos a base de dados conforme descrito no enunciado.

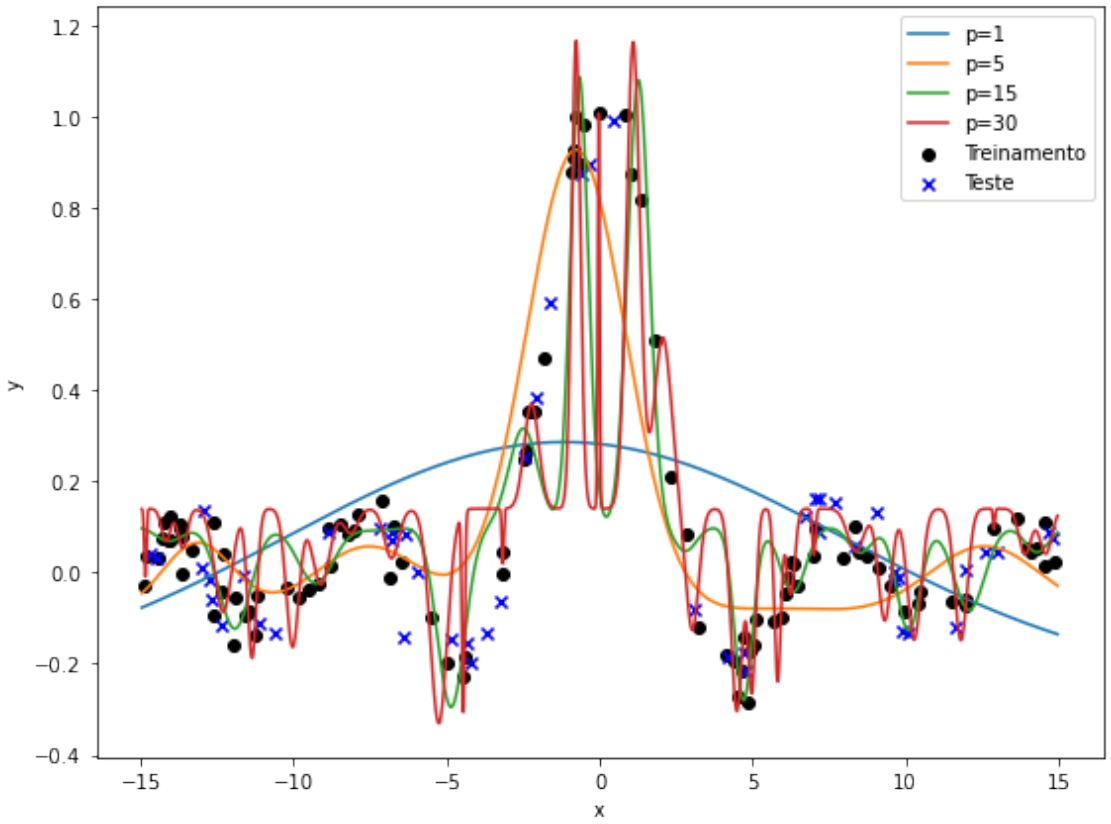
[21]:
```
# define the sinc function generator
n_train = 100
n_test = 50

X_train = np.random.uniform(-15,15,size=(n_train,1))
y_train = np.sin(X_train) / X_train + np.random.normal(loc=0,scale=0.05,␣
 ↪size=(n_train,1))

X_test = np.random.uniform(-15,15, size=(n_test,1))
y_test = np.sin(X_test) / X_test + np.random.normal(loc=0,scale=0.05,␣
 ↪size=(n_test,1))
```

A seguir, é criada uma rotina que gera gráficos comparando as aproximações

[22]:
```
from matplotlib.pyplot import cm
cmap = cm.get_cmap('tab10', 10)    # PiYG

def plot_regression_results(X_train, y_train, X_test, y_test,␣
 ↪neurons=[1,5,15,30]):
    fig, ax = plt.subplots(figsize=(8,6))
```

```
    # horizontal stack vectors to create x1,x2 input for the model
    grid = np.arange(np.min(X_train)-0.1, np.max(X_train)+0.1, 0.01).
↪reshape(-1, 1)
    mse_test = {}
    mse_train = {}

    ax.scatter(X_train, y_train, color='black', label='Treinamento')
    ax.scatter(X_test, y_test, marker='x', color='blue', label='Teste')

    for i, p in enumerate(neurons):
        # train the model
        model = RBF(p=p).fit(X_train, y_train)

        # make predictions for the grid
        yhat_grid = model.predict(grid)
        # make predictions for the datasets
        yhat_test = model.predict(X_test)
        yhat_train = model.predict(X_train)

        mse_test[p] = mean_squared_error(y_test, yhat_test.ravel())
        mse_train[p] = mean_squared_error(y_train, yhat_train.ravel())

        ax.plot(grid, yhat_grid, color=cmap(i), linestyle='-',label=f'p={p}')

    ax.set_xlabel('x')
    ax.set_ylabel('y')

    ax.legend()
    fig.tight_layout()
    fig.show()

    return mse_test, mse_train
```

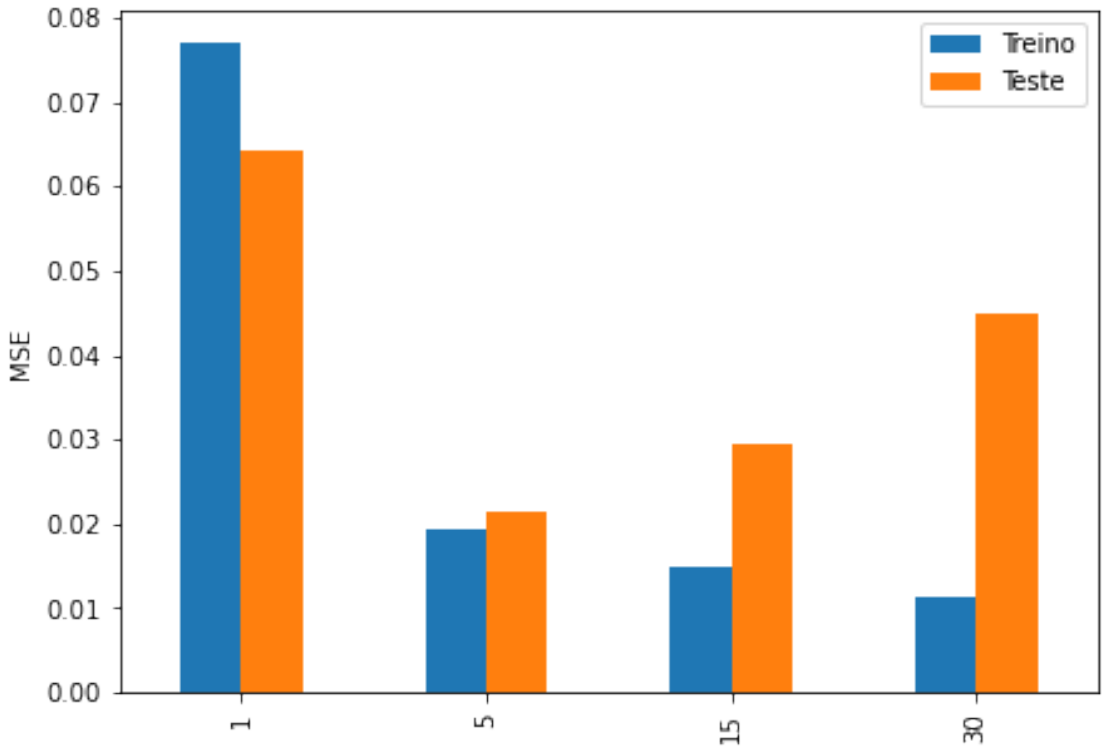
[34]:
```
mse_test, mse_train = plot_regression_results(X_train, y_train, X_test, y_test,␣
↪[1, 5, 15, 30])
```



[61]:
```
res_dict = {'Treino': mse_train, 'Teste': mse_test}
res_df = pd.DataFrame().from_dict(res_dict)#.reset_index()
print('MSE')
display(res_df)
res_df.plot(kind='bar', stacked=False, xlabel='', ylabel='MSE', table=False,␣
 ↪figsize=(7,5))
plt.show()
```

```
MSE
```

	Treino	Teste
1	0.076990	0.064249
5	0.019396	0.021376
15	0.014813	0.029336
30	0.011274	0.045041



A partir dos gráficos e tabela anteriores, podemos concluir para este problema que:

- O uso de somente 1 neurônio na camada escondida resultou em um under-fitting aos dados.
- Um número de cerca de 5 neurônios para a camada escondida resultou em um boa generalização, conforme visto no gráfico e também por aprensentar o menor erro para o conjunto de teste.
- Com somente 15 neurônios já é possível observar o fênomeno de over-fitting, o qual se torna muito mais pronunciado à medida em que essa quantidade aumenta. Conforme pode ser visto acima, embora o erro de treinamento diminua há o aumento do erro de teste, indicando a baixa generalização do modelo.