

Visualizing KNN, SVM and XGBoost on the Iris Dataset

Use Python to visualize how KNN, SVM and XGBoost algorithms classify certain data points in the Iris dataset.

```
In [1]: # Import data and libraries
import pandas as pd
import numpy as np
from sklearn import datasets
%pylab inline
pylab.rcParams['figure.figsize'] = (10, 6)

iris = datasets.load_iris()

# We'll use the petal length and width only for this analysis
X = iris.data[:, [2, 3]]
y = iris.target

# Place the iris data into a pandas dataframe
iris_df = pd.DataFrame(iris.data[:, [2, 3]], columns=iris.feature_names[2:])

# View the first 5 rows of the data
print(iris_df.head())

# Print the unique labels of the dataset
print('\n' + 'The unique labels in this data are ' + str(np.unique(y)))
```

%pylab is deprecated, use %matplotlib inline and import the required libraries.
Populating the interactive namespace from numpy and matplotlib

	petal length (cm)	petal width (cm)
0	1.4	0.2
1	1.4	0.2
2	1.3	0.2
3	1.5	0.2
4	1.4	0.2

The unique labels in this data are [0 1 2]

Split the data into training and test datasets

```
In [2]: from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.3, random_state=0)

print('There are {} samples in the training set and {} samples in the test set'.format(
X_train.shape[0], X_test.shape[0]))
print()
```

There are 105 samples in the training set and 45 samples in the test set

Scale the data using sklearn.

```
In [3]: from sklearn.preprocessing import StandardScaler
```

```

sc = StandardScaler()

sc.fit(X_train)

X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)

print('After standardizing our features, the first 5 rows of our data now look like this:\n')
print(pd.DataFrame(X_train_std, columns=iris_df.columns).head())

```

After standardizing our features, the first 5 rows of our data now look like this:

	petal length (cm)	petal width (cm)
0	-0.182950	-0.293181
1	0.930661	0.737246
2	1.042022	1.638870
3	0.652258	0.350836
4	1.097702	0.737246

If we plot the original data, we can see that one of the classes is linearly separable, but the other two are not.

```

In [4]: from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

markers = ('s', 'x', 'o')
colors = ('red', 'blue', 'lightgreen')
cmap = ListedColormap(colors[:len(np.unique(y_test))])
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                c=cmap(idx), marker=markers[idx], label=cl)

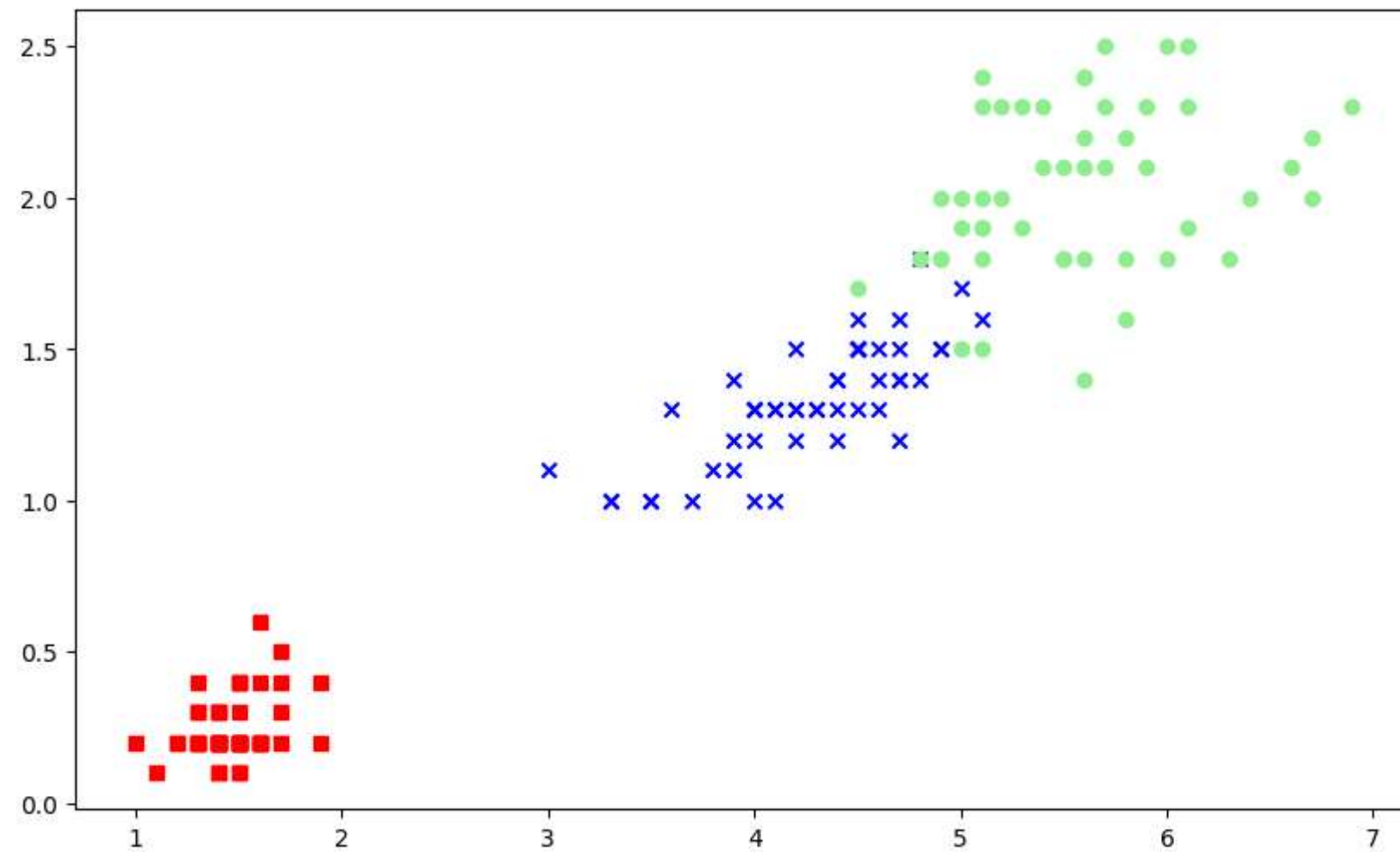
```

/tmp/ipykernel_32/3783131042.py:8: UserWarning: *c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*. Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.

```

plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],

```



Let's try to use a Linear SVC to predict the labels of our test data.

```
In [5]: from sklearn.svm import SVC

svm = SVC(kernel='rbf', random_state=0, gamma=.10, C=1.0)
svm.fit(X_train_std, y_train)

print('The accuracy of the svm classifier on training data is {:.2f} out of 1'.format(svm.score(X_train_std, y_train)))

print('The accuracy of the svm classifier on test data is {:.2f} out of 1'.format(svm.score(X_test_std, y_test)))
```

The accuracy of the svm classifier on training data is 0.95 out of 1
The accuracy of the svm classifier on test data is 0.98 out of 1

It looks like our classifier performs pretty well. Let's visualize how the model classified the samples in our test data.

```
In [6]: import warnings

def versiontuple(v):
    return tuple(map(int, (v.split("."))))

def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
```

```

colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
cmap = ListedColormap(colors[:len(np.unique(y))])

# plot the decision surface
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                        np.arange(x2_min, x2_max, resolution))
Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())

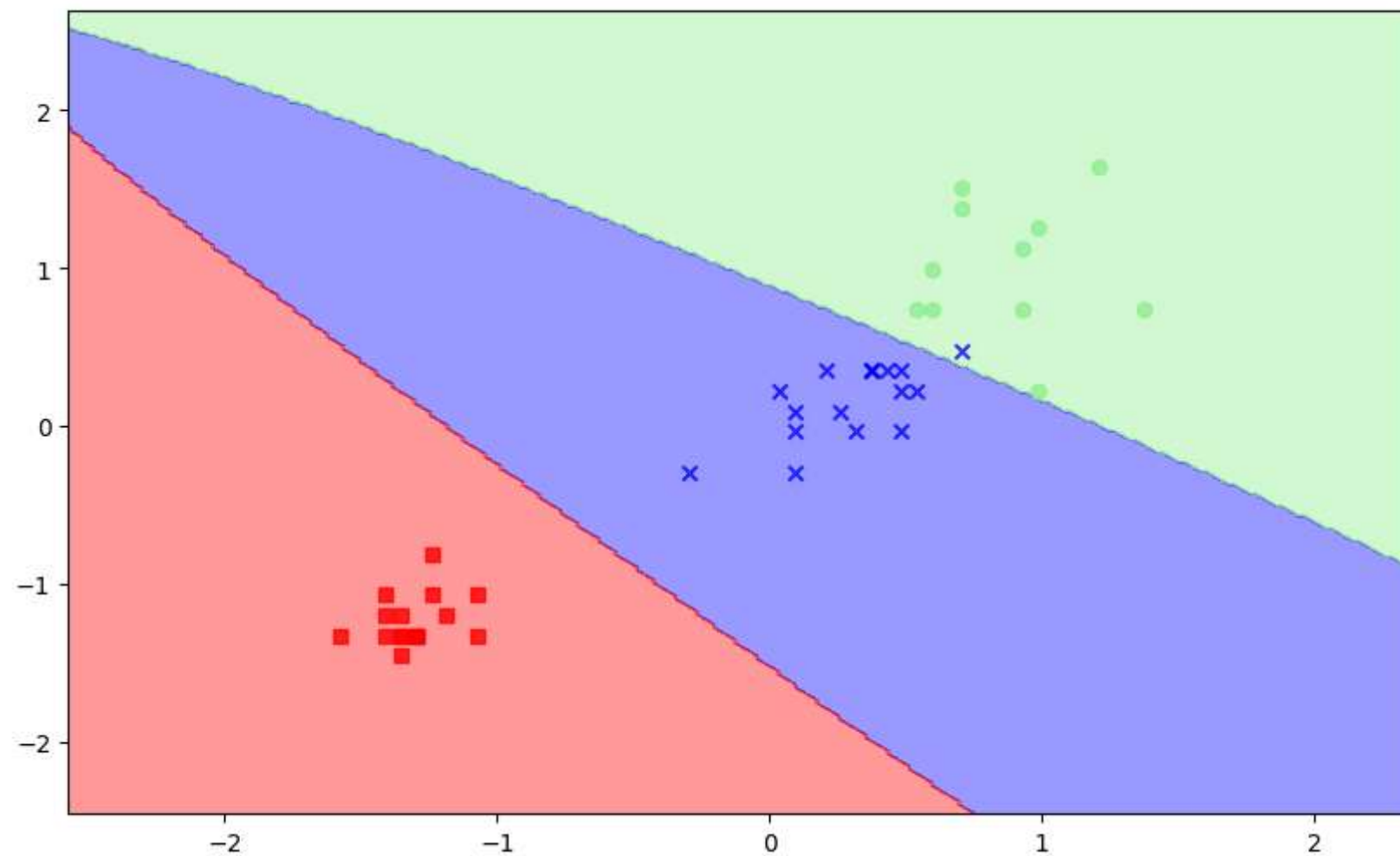
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                alpha=0.8, c=cmap(idx),
                marker=markers[idx], label=cl)

```

In [7]: plot_decision_regions(X_test_std, y_test, svm)

/tmp/ipykernel_32/76825547.py:27: UserWarning: *c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*. Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.

```
plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
```



Now, let's test out a KNN classifier.

```
In [8]: from sklearn.neighbors import KNeighborsClassifier
```

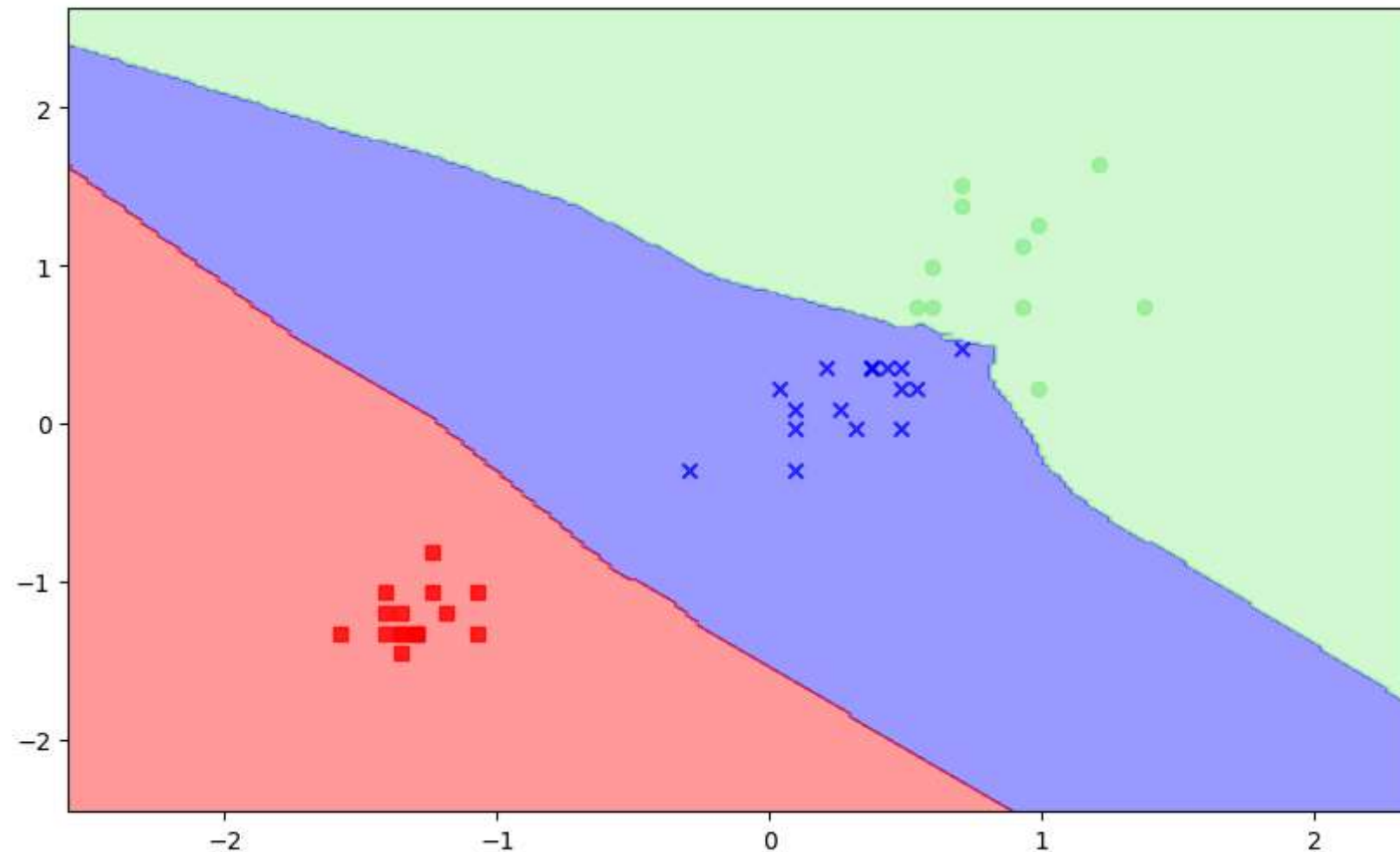
```
knn = KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski')  
knn.fit(X_train_std, y_train)
```

```
print('The accuracy of the knn classifier is {:.2f} out of 1 on training data'.format(knn.score(X_train_std, y_train)))  
print('The accuracy of the knn classifier is {:.2f} out of 1 on test data'.format(knn.score(X_test_std, y_test)))
```

The accuracy of the knn classifier is 0.95 out of 1 on training data
The accuracy of the knn classifier is 1.00 out of 1 on test data

```
In [9]: plot_decision_regions(X_test_std, y_test, knn)
```

```
/tmp/ipykernel_32/76825547.py:27: UserWarning: *c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches with *x* & *y*. Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.  
plt.scatter(x=X[y == c1, 0], y=X[y == c1, 1],
```



Next we plot an XGBoost classifier.

```
In [10]: import xgboost as xgb
```

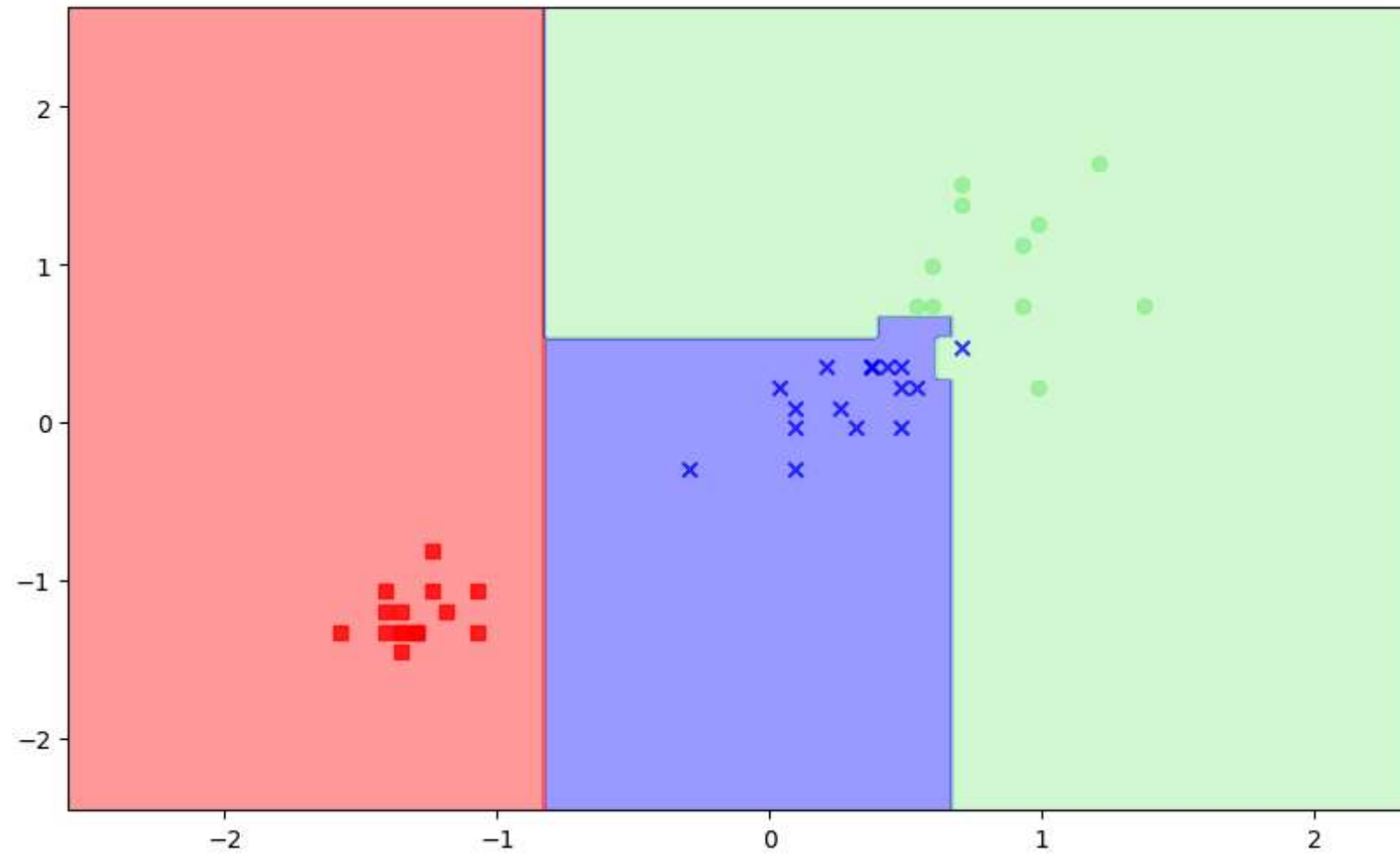
```
xgb_clf = xgb.XGBClassifier()  
xgb_clf = xgb_clf.fit(X_train_std, y_train)
```

```
print('The accuracy of the xgb classifier is {:.2f} out of 1 on training data'.format(xgb_clf.score(X_train_std, y_train)))  
print('The accuracy of the xgb classifier is {:.2f} out of 1 on test data'.format(xgb_clf.score(X_test_std, y_test)))
```

The accuracy of the xgb classifier is 0.99 out of 1 on training data
The accuracy of the xgb classifier is 0.98 out of 1 on test data

```
In [11]: plot_decision_regions(X_test_std, y_test, xgb_clf)
```

```
/tmp/ipykernel_32/76825547.py:27: UserWarning: *c* argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have precedence in case its length matches w  
ith *x* & *y*. Please use the *color* keyword-argument or provide a 2D array with a single row if you intend to specify the same RGB or RGBA value for all points.  
plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
```



In all classifiers, the performance on the test data was better than the training data. At least with the parameters specified in this very simple approach, the KNN algorithm seems to have performed the best. However, this may not be the case depending on the dataset and more careful parameter tuning.