Exercise 1.1

```
#1.
print("Dimensions of X:", X.shape)

#2.
print("Dimension of y:", y.ndim)

#3.
#The vector y contains the classifiers,
#divided into two groups of zeros and ones.
```

```
Dimensions of X: (500, 2)
Dimension of y: 1
```

Exercise 1.2

Values of m:
[1, 0.5, -0.2]

Values of b:
[0.65, 1.6, 2.9]

The value m is the slope to the line,
while the value b is a constant and tells us where the line crosses y- axis

The line is also called for 'hyperplane' in this case.

Exercise 1.3

The value *d* represents the constraint of a hyperplane.
It is used in yfit to set the upper- and lower bound of a hyperplane (the distance from the hyperplane).

Value *d* is used to compute the margin, the distance between parallel hyperplanes.

Exercise 1.4

```
#1.
print(model.support_vectors_)

print("There are ", len(model.support_vectors_), " support vectors")

#2.
print("Vectors divided in", model.n_support_[0], "red and",
      model.n_support_[1], "yellow")
```

```
[[0.44359863 3.11530945]
 [2.33812285 3.43116792]
 [2.06156753 1.96918596]]
There are  3  support vectors
Vectors divided in 2 red and 1 yellow
```

Exercise 1.5

Where we calculated radial basis r:

```
r = np.exp(-(X ** 2).sum(1))
```

We use SVC(kernel='linear') to center our data (having one classifier centered and the rest around the centered data).
Since the data is in two-dimensional, we cannot use Linear Regression to create a hyperplane to seperate our data. Thats why we need to make our data into three-dimensional space. We use radial basis r to transform our data into three-dimensional space.
Data near the origo will be the center and the top of the three-dimensional space. Data that is further away will come further below in the the three-dimensional space.

Exercise 1.6

```
point = [0.5 , 0]

#1.
print("The point", point, "will be classified as", clf.predict([[0.5, 0]]))
print()

#2.
prob0 = clf.predict_proba([point])[0][0]*100
prob1 = clf.predict_proba([point])[0][1]*100
print("Probability of %.2f" % prob0,
      "% that it will be belonging to the red class")
print("Probability of %.2f" % prob1,
      "% that it will be belonging to the yellow class")
```

```
The point [0.5, 0] will be classified as [1]

Probability of 12.05 % that it will be belonging to the red class
Probability of 87.95 % that it will be belonging to the yellow class
```

Exercise 1.7

The best value of C depends on the training set.

For a large value of C, the SVM will rarely missclassify any single points because the tolerance of the SVM is high for higher values of C.

For a smaller value of C, the SVM is allowed some degree of freedom to be able to find the best hyperplane, but will missclassify more points than a larger value of C would.

One can perform a grid serach cross-validation before the final training to estimate the best value of C.

Exercise 1.8

```
#1.
import time
start_timeA = time.time()
a = calculate_dot_product_in_higher_dimensional_space(x, z)
print("Time a: %0.10f seconds" % (time.time() - start_timeA))

start_timeB = time.time()
b = calculate_dot_product_in_lower_dimensional_space_with_kernel_trick(x, z)

print("Time b: %0.10f seconds" % (time.time() - start_timeB))
```

```
Time a: 0.7550382614 seconds
Time b: 0.0019989014 seconds
```

2. It is much faster for the function that implements the kernel trick
Note that the run time will differ for each time, but the conclusion of the kernel trick being faster still holds.

Exercise 1.9

```
def calculate_dot_product_in_higher_dimensional_space_6d(x, z):
  # transform x into higher dimensional space
  transformed_x = np.ndarray(shape=(x.shape[0], x.shape[1]+4))
  for i in range(x.shape[0]):
    transformed_x[i] = np.array([1,
                                 sqrt(2)*x[i][0],
                                 sqrt(2)*x[i][1],
                                 x[i][0]*x[i][0],
                                 x[i][1]*x[i][1],
                                 sqrt(2)*x[i][0]*x[i][1]])
  # transform z into higher dimensional space
  transformed_z = np.array([[1],
                            [sqrt(2)*z[0]],
                            [sqrt(2)*z[1]],
                            [z[0]*z[0]],
                            [z[1]*z[1]],
                            [sqrt(2)*z[0]*z[1]]])

  dot_product = np.ndarray(shape=(transformed_x.shape[0], 1))
  for i in range(transformed_x.shape[0]):
    dot_product[i] = transformed_x[i].T.dot(transformed_z)

  return dot_product


def calculate_dot_product_in_lower_dimensional_space_with_kernel_trick_6d(x, z):
  dot_product = 1 + x.dot(z)

  return dot_product**2
```

Using the functions:

```
a = calculate_dot_product_in_higher_dimensional_space_6d(x, z)
b = calculate_dot_product_in_lower_dimensional_space_with_kernel_trick_6d(x, z)

print(a)
print(b)
```

Will give the results:

```
[[2.70459484]
 [1.64899488]
 [4.59286593]
 ...
 [4.93709036]
 [2.04406641]
 [2.11175258]]
[2.70459484 1.64899488 4.59286593 ... 4.93709036 2.04406641 2.11175258]
```

Both function will give the same result, but the function with kernel trick is much faster.

Exercise 1.10

```
def calculate_precision(yfit, ytest):
    TP = 0
    FP = 0

    for y in range(len(yfit)):
        if ytest[y] == yfit[y]:
            TP = TP + 1
        else:
            FP = FP + 1

    precision = TP/(TP + FP) * 100
    print("Precision: ", precision)
    return precision
```

Exercise 2.1

Voting parameter: 'hard'
VotingClassifier 0.904

Voting parameter: 'soft'
VotingClassifier 0.912

Parameter 'hard' takes the predicted class labels that rules the voting.

Parameter 'soft' takes the average of the probabilities of the predicted class labels.

Exercise 2.2

```
import pandas as pd
import io
import requests
import numpy
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.d
s = requests.get(url).content

# save the data in a dataframe
df = pd.read_csv(io.StringIO(s.decode('utf-8')), header=None)

y_df = numpy.array(df)[:,0]
X_df = numpy.array(df)[:, 1:]
```

```
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import AdaBoostClassifier
import operator

le = LabelEncoder()

y_fit = le.fit_transform(y_df)

X_df_train, X_df_test, y_df_train, y_df_test = train_test_split(
                        X_df, y_fit, test_size=0.33, random_state=42)

dt_clf = DecisionTreeClassifier(random_state=42)

rnd_clf = RandomForestClassifier(random_state=42,
                                 n_estimators=500,
                                 max_features=1)

bag_clf = BaggingClassifier(n_estimators=500,
                            max_samples=100,
                            bootstrap=True,
                            n_jobs=-1,
                            random_state=42)

ada_clf = AdaBoostClassifier(n_estimators=500, random_state=42)

dt_clf.fit(X_df_train, y_df_train)
rnd_clf.fit(X_df_train, y_df_train)
bag_clf.fit(X_df_train, y_df_train)
ada_clf.fit(X_df_train, y_df_train)

dt_score = dt_clf.score(X_df_test, y_df_test)
rnd_score = rnd_clf.score(X_df_test, y_df_test)
bag_score = bag_clf.score(X_df_test, y_df_test)
ada_score = ada_clf.score(X_df_test, y_df_test)
```

```
dict = {
        "DecisionTreeClassifier": dt_score,
        "RandomTreeClassifier": rnd_score,
        "BaggingClassifier": bag_score,
        "AdaBoostClassifier": ada_score,
        }

for keys, values in dict.items():
    print(keys,":  ", values)

print("Best classifier:", max(dict.items(), key=operator.itemgetter(1))[0])
```

```
DecisionTreeClassifier :   0.5068890500362582
RandomTreeClassifier :   0.5395213923132705
BaggingClassifier :   0.5467730239303843
AdaBoostClassifier :   0.5315445975344453
Best classifier: BaggingClassifier
```

The best classifier depends on multiple factors like test_size, n_estimator, random_state, etc.