INF283 – Exercise 4

Sindre Eik de Lange

1. Support Vector Machines (SVMs)

1.1

1. Code to find the dimensions of a vector **X** is already defined as *X_shape*, which, according to *make_blobs()* DocStrings is equal to (**n_samples**, **n_features**).

2. Code to find the dimensions of a vector y is already defined as *y_shape*, which, according to *make_blobs()* DocStrings is equal to (**n_samples**).

3. Again, according to *make_blobs()* DocString, vector **y** contains: 'The integer labels for cluster membership of each sample', aka. each tuples' (in **X**) label.

1.2

m and **b** are the values in each touple, contained in the list (of touples) iterated through in the for-loop. More specifically, the for-loop iterates through a list, containing touples, consisting of values (**m_i**, **b_i**), where **i** = iteration number. They represent the values **a** and **b** for linear functions, given as *y=ax+b*, where **xfit** then represents the **x**.

1.3

d represents the distance between each vectors' line and the nearest data point on each side.

1.4

1. Code to print out the value of support vectors: *model.support_vectors_* (after assigning **model** as the SVM classifier).

2. There are 3 support vectors.

1.5

They centered **r** in origo.

1.6

According to https://matplotlib.org/examples/color/colormaps_reference.html, when specifying *cmap='autumn'*, then the colors will range from **red** to **yellow**, so I assume that it starts on **red** (left side), e.g. 0 = **red**, 1 = **yellow**.

```
new_data_p = [[0.5, 0]]
prediction = clf.predict(new_data_p)
print(prediction)
print(clf.predict_proba(new_data_p)[0][1])
```

```
[1]
0.853731808290216
```

So, as one can see here, the new data point will be classified as 1 = **yellow**, and the probability of this happening is approximately 0.854.

1.7

This is probably data dependent like everything else in machine learning, but I would assume one would utilize a validation set, combined with either GridSearch or RandomSearch, with or without cross validation (depending on available resources and the task domain), in order to get the optimal hyperparameter. Alternatively, one could utilize Bayesian Optimization with 'skopt'.

1.8

```
import time
start_1 = time.time()
a = calculate_dot_product_in_higher_dimensional_space(x, z)
end_1 = time.time()
tot_time_1 = (end_1 - start_1)

start_2 = time.time()
b = calculate_dot_product_in_lower_dimensional_space_with_kernel_trick(x, z)
end_2 = time.time()
tot_time_2 = (end_2 - start_2)
print("Total time for the first function: ", tot_time_1)
print("Total time for the second function: ", tot_time_2)
print(tot_time_1/tot_time_2)
```

```
Total time for the first function:  0.4200263023376465
Total time for the second function:  0.0009660720825195312
434.77739387956564
```

The **kernel trick** function is faster than the other one.

1.9

```
def calculate_dot_product_in_higher_dimensional_space_6D(x, z):
  #-------------------------------------------------------------------------#
  # Step 1: Transform the data explicitly in higher dimensional space
  #-------------------------------------------------------------------------#

  # 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] ** 2,
                                     x[i][1] ** 2,
                                     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]**2],
                               [z[0]**2],
                               [sqrt(2)*z[0]*z[1]]])


  #-------------------------------------------------------------------------#
  # Step 2: Take dot product between each row of transformed_x & transformed_z
  #-------------------------------------------------------------------------#

    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):

  #-------------------------------------------------------------------------#
  # Step 1: Take dot product of x.T and z in lower dimensional space
  #-------------------------------------------------------------------------#
  dot_product = x.dot(z)

  #-------------------------------------------------------------------------#
  # Step 2: Square the dot product and return
  #-------------------------------------------------------------------------#
  return dot_product**6
```

```
import time
start_1 = time.time()
a = calculate_dot_product_in_higher_dimensional_space_6D(x, z)
end_1 = time.time()
tot_time_1 = (end_1 - start_1)

start_2 = time.time()
b = calculate_dot_product_in_lower_dimensional_space_with_kernel_trick_6D(x, z)
end_2 = time.time()
tot_time_2 = (end_2 - start_2)
print("Total time for the first function: ", tot_time_1)
print("Total time for the second function: ", tot_time_2)
print(tot_time_1/tot_time_2)
```

```
Total time for the first function:  0.5058426856994629
Total time for the second function:  0.003958940505981445
127.77223727792834
```

1.10

```
from sklearn.metrics import confusion_matrix

def calc_precision_metric(y_true, y_pred):
    """
    y_true --> Correct target values
    y_pred --> Estimated target values returned by classifier
    """
    mat = confusion_matrix(y_true, y_pred)
    FP = mat.sum(axis=0) - np.diag(mat)
    TP = np.diag(mat)
    precision = (TP / (FP + TP))
    return precision
```

```
print(calc_precision_metric(ytest, yfit))
```

```
[0.61904762 0.82191781 0.71428571 0.98058252 0.73076923 0.875
 0.8        0.83333333]
```

2. Ensemble Learning

2.1

Changing the 'voting' parameter from 'hard' to 'soft' slightly improves the **VotingClassifier**. This is because the 'hard' classifier only considers the binary input from each voter, while the 'soft' classifier takes into account the average of probabilities. This means that in our case, where each of the "weak" learners are fairly good (> .85), it helps to consider how certain each classifier is.

2.2

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

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

```
df.describe()
```

```
df.head()
```

```
target_column = 0
X = df.drop(target_column, axis=1)
y = df[target_column]
```

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
le.fit(y)
y_le_trans = le.transform(y)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y_le_trans, random_state=random_state, stratify=y_le_trans)
```

```
X_train.shape
```

```
y_train.shape
```

Adaboost

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
```

```
adaB_clf = AdaBoostClassifier(base_estimator=DecisionTreeClassifier(random_state=42),
                              n_estimators=500,
                              random_state = random_state)
adaB_clf.fit(X_train, y_train)
y_pred = adaB_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.4842105263157895
```

BaggingClassifier

```
from sklearn.ensemble import BaggingClassifier
```

```
bagging_clf = BaggingClassifier(base_estimator=DecisionTreeClassifier(random_state=random_state),
                             n_estimators = 500,
                              warm_start = True,
                              n_jobs = -1,
                              random_state = 42)
bagging_clf.fit(X_train, y_train)
y_pred = bagging_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.5435406698564593
```

Random Forest Classifier

```
from sklearn.ensemble import RandomForestClassifier
```

```
random_f_clf = RandomForestClassifier(n_estimators=500,
                                      n_jobs = 2,
                                      random_state=random_state)
random_f_clf.fit(X_train, y_train)
y_pred = random_f_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.5416267942583732
```

Decision Tree Classifier

```
from sklearn.tree import DecisionTreeClassifier
```

```
tree_clf = DecisionTreeClassifier()
tree_clf.fit(X_train, y_train)
y_pred = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.48229665071770333
```

So, one can see that the **Bagging Classifier** is the best classifier. Important to note that this is based on running the models one time, and not utilizing any hyperparameter optimization tools like GridSearch(CV) or RandomSearch(CV).