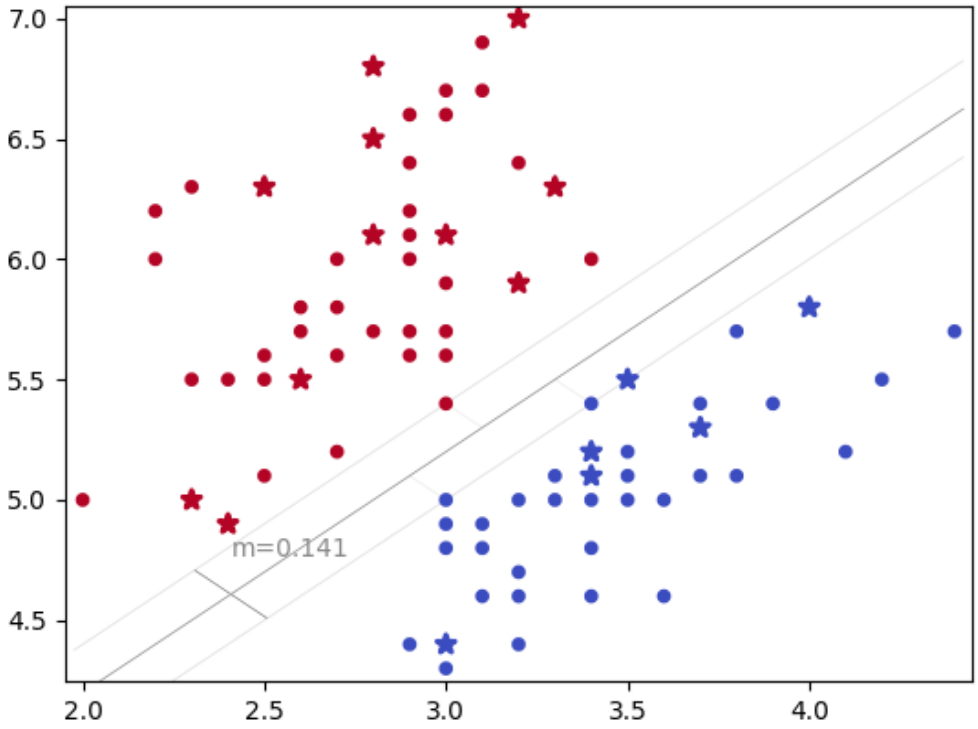
TASK 1

In this task I'm implementing the linear SVM algorithm with hard-margin and soft-margin variants. Then, I'm applying the SVM with hard-margin to the irisSV dataset and the SVM with soft-margin to the irisVV dataset.

SVM WITH HARD-MARGIN – IRISSV DATASET

After dividing the irisSV dataset into training $(4/5)$ and test $(1/5)$ subsets, I've trained the hard-margin SVM on the training subset and applied the resulting model to the test subset.

That's the plot of the hyperplane with the support vectors highlighted:



That's the confusion matrix:

$$\begin{bmatrix} 6 & 0 \\ 0 & 11 \end{bmatrix}$$

Here are the accuracy, the recall and the precision obtained:

$$accuracy = 1.0$$

$$recall = 1.0$$

$$precision = 1.0$$

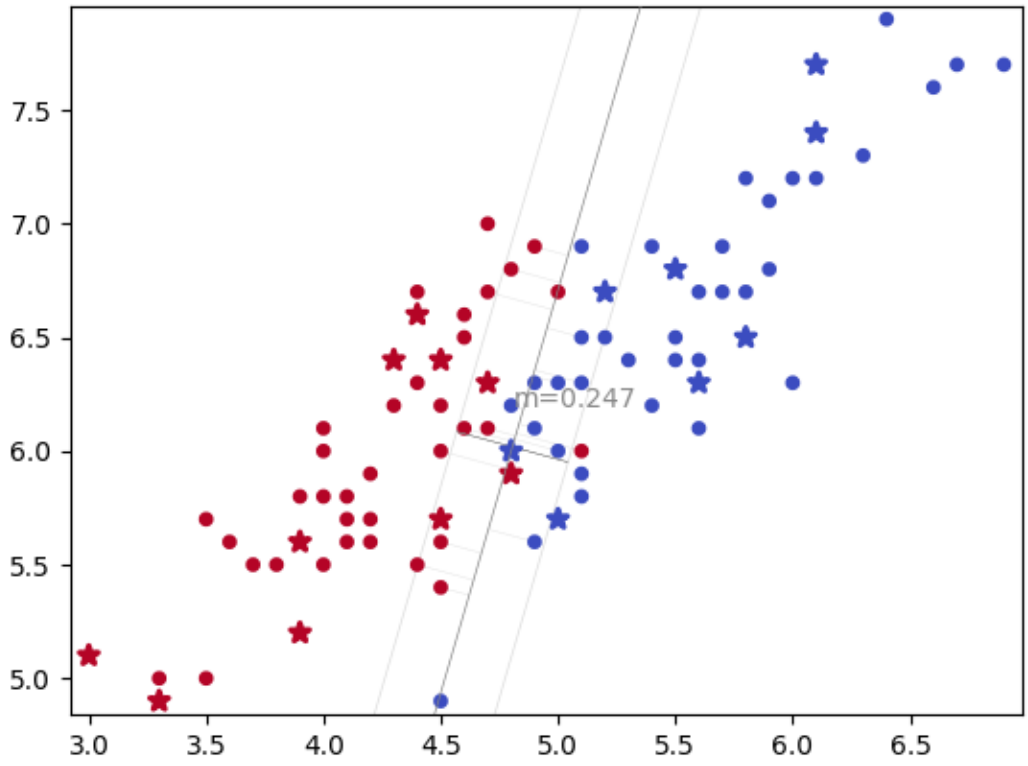
The equation of the separating hyperplane is

$$\begin{bmatrix} 5 \\ -5 \end{bmatrix} x + 10.99 = 0$$

SVM WITH SOFT-MARGIN – IRISVV DATASET

After dividing the irisVV dataset into training $(4/5)$ and test $(1/5)$ subsets, I've trained the soft-margin SVM on the training subset setting $c = 2$ and applied the resulting model to the test subset.

That's the plot of the hyperplane with the support vectors highlighted:



That's the confusion matrix:

$$\begin{bmatrix} 8 & 1 \\ 0 & 9 \end{bmatrix}$$

Here are the accuracy, the recall and the precision obtained:

$$accuracy = 0.944$$

$$recall = 0.9$$

$$precision = 1.0$$

The equation of the separating hyperplane is

$$\begin{bmatrix} 1.099 \\ -3.9 \end{bmatrix} x + 12.12 = 0$$

TASK 2

Run an evaluation of the soft-margin SVM on the irisVV dataset with cross-validation. That is, run 10 rounds of cross-validation with 5 folds on the irisVV dataset. Report the mean and variance of the classifier's accuracy across the successive rounds.

```
Round 1/10
Fold 1/5
Accuracy: 0.8823529411764706

Fold 2/5
Accuracy: 0.9411764705882353

Fold 3/5
Accuracy: 0.8823529411764706

Fold 4/5
Accuracy: 0.9411764705882353

Fold 5/5
Accuracy: 0.9411764705882353

Mean Accuracy for Round 1: 0.9176470588235295
Variance of Accuracy for Round 1: 0.0008304498269896196

Round 2/10
Fold 1/5
Accuracy: 0.9411764705882353

Fold 2/5
Accuracy: 0.9411764705882353

Fold 3/5
Accuracy: 0.9411764705882353

Fold 4/5
Accuracy: 1.0

Fold 5/5
Accuracy: 0.8823529411764706

Mean Accuracy for Round 2: 0.9411764705882352
Variance of Accuracy for Round 2: 0.0013840830449826996

Round 3/10
Fold 1/5
Accuracy: 1.0

Fold 2/5
Accuracy: 0.8823529411764706

Fold 3/5
Accuracy: 0.8823529411764706

Fold 4/5
Accuracy: 0.8823529411764706

Fold 5/5
Accuracy: 1.0

Mean Accuracy for Round 3: 0.9294117647058823
Variance of Accuracy for Round 3: 0.0033217993079584793

Round 4/10
Fold 1/5
Accuracy: 0.9411764705882353
```

```
Fold 2/5
Accuracy: 0.9411764705882353

Fold 3/5
Accuracy: 0.9411764705882353

Fold 4/5
Accuracy: 0.8235294117647058

Fold 5/5
Accuracy: 1.0

Mean Accuracy for Round 4: 0.9294117647058823
Variance of Accuracy for Round 4: 0.0033217993079584793

Round 5/10
Fold 1/5
Accuracy: 0.9411764705882353

Fold 2/5
Accuracy: 1.0

Fold 3/5
Accuracy: 0.9411764705882353

Fold 4/5
Accuracy: 0.9411764705882353

Fold 5/5
Accuracy: 0.8823529411764706

Mean Accuracy for Round 5: 0.9411764705882352
Variance of Accuracy for Round 5: 0.0013840830449826996

Round 6/10
Fold 1/5
Accuracy: 0.8823529411764706

Fold 2/5
Accuracy: 0.9411764705882353

Fold 3/5
Accuracy: 0.8823529411764706

Fold 4/5
Accuracy: 1.0

Fold 5/5
Accuracy: 0.9411764705882353

Mean Accuracy for Round 6: 0.9294117647058824
Variance of Accuracy for Round 6: 0.0019377162629757795

Round 7/10
Fold 1/5
Accuracy: 0.8823529411764706

Fold 2/5
Accuracy: 0.8823529411764706

Fold 3/5
Accuracy: 0.9411764705882353

Fold 4/5
Accuracy: 1.0
```

```
Fold 5/5
Accuracy: 1.0

Mean Accuracy for Round 7: 0.9411764705882353
Variance of Accuracy for Round 7: 0.002768166089965399

Round 8/10
Fold 1/5
Accuracy: 0.8823529411764706

Fold 2/5
Accuracy: 0.9411764705882353

Fold 3/5
Accuracy: 0.9411764705882353

Fold 4/5
Accuracy: 0.9411764705882353

Fold 5/5
Accuracy: 1.0

Mean Accuracy for Round 8: 0.9411764705882353
Variance of Accuracy for Round 8: 0.0013840830449826996

Round 9/10
Fold 1/5
Accuracy: 0.9411764705882353

Fold 2/5
Accuracy: 0.9411764705882353

Fold 3/5
Accuracy: 0.9411764705882353

Fold 4/5
Accuracy: 0.9411764705882353

Fold 5/5
Accuracy: 0.9411764705882353

Mean Accuracy for Round 9: 0.9411764705882353
Variance of Accuracy for Round 9: 0.0

Round 10/10
Fold 1/5
Accuracy: 1.0

Fold 2/5
Accuracy: 1.0

Fold 3/5
Accuracy: 0.9411764705882353

Fold 4/5
Accuracy: 0.8823529411764706

Fold 5/5
Accuracy: 1.0

Mean Accuracy for Round 10: 0.9647058823529413
Variance of Accuracy for Round 10: 0.00221453287197232

Overall Mean Accuracy: 0.9376470588235294
Overall Variance of Accuracy: 0.00199446366782007
```

TASK 3

Implement the AdaBoost algorithm with instance weighting done via sampling. Apply it to the credit dataset with linear SVM. Try using less aggressive weight updates, calculating the update factor as

$$\alpha_t \leftarrow \beta \cdot \frac{\ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)}{2}$$

where $0 < \beta < 1$. What happens if $\beta = 0$?

I've tried this kind of code but it doesn't run. The issue is that it generates nan when trying to compute the error of the model in predicting on the training set (highlighted line):

```
### TASK 3

SEED = 2304423
RandomNumGen.set_seed(SEED)

series = "creditDE"
data_params = {"filename": "creditDE.csv", "last_column_str": False}
algo_params_series = [{"c": 1, "ktype": "linear", "kparams": {}},
                      {"c": 1, "ktype": "polynomial", "kparams": {"degrees": [2]}},
                      {"c": 1, "ktype": "RBF", "kparams": {"sigma": 1.}}]
beta = 0

def ada_boost(data_params, algo_params_series, ratio_train = .8):

    dataset, _, _ = load_csv(**data_params)

    ids = RandomNumGen.get_gen().permutation(dataset.shape[0])
    split_pos = int(len(ids)*ratio_train)
    train_ids, test_ids = ids[:split_pos], ids[split_pos:]
    train_set = dataset[train_ids]
    test_set = dataset[test_ids]

    for algo_params in algo_params_series:

        classifiers = []

        # training ada_boost
        weights = numpy.ones(train_set.shape[0])
        for _ in range(num_classifiers):
            base_classifier, support_vector_indices = prepare_svm_model(
                train_set[:, :-1], train_set[:, -1], **algo_params
            )
            predictions = svm_predict_vs(train_set[:, :-1], base_classifier)
            errors = weights * (predictions != train_set[:, -1])
            error_rate = numpy.sum(errors) / numpy.sum(weights)

            if error_rate == 0:
                classifiers.append((base_classifier, support_vector_indices, 1.0, 0))
                break

            alpha = beta * numpy.log((1 - error_rate) / error_rate) / 2.0
            weights *= numpy.exp(alpha * errors)

            classifiers.append((base_classifier, support_vector_indices, alpha,
base_classifier["bias"]))

    # prediction
    final_predictions = numpy.zeros(test_set.shape[0])
    for classifier, _, alpha, bias in classifiers:
        predictions = svm_predict_vs(test_set[:, :-1], classifier)
        final_predictions += alpha * predictions
```

```
    return numpy.sign(final_predictions + bias)

cvxopt.solvers.options['show_progress'] = False

for algo_params in algo_params_series:
    num_classifiers = 10
    adaboost = ada_boost(data_params, algo_params_series, ratio_train = .8)
    print("Final prediction ", adaboost)
```

TASK 4

Empirically compare the impact of boosting and bagging when combined to a linear SVM vs. to a SVM with a RBF kernel on the credit dataset.

For this task, I've implemented the bagging method but I cannot do the comparison yet because I need to fix the boosting code.

```
### TASK 4

SEED = 2304423
RandomNumGen.set_seed(SEED)

# bagging
def bagging(train_set, base_classifier_params):
    classifiers = []
    n_samples = train_set.shape[0]
    n_features = train_set.shape[1] - 1

    bootstrap_indices = numpy.random.choice(n_samples, n_samples, replace=True)
    bootstrap_sample = train_set[bootstrap_indices]
    base_classifier, _ = prepare_svm_model(
        bootstrap_sample[:, :-1], bootstrap_sample[:, -1], **base_classifier_params
    )
    classifiers.append(base_classifier)
    return classifiers

def bagging_predict(data, classifiers):
    predictions = numpy.zeros(data.shape[0])
    for classifier in classifiers:
        predictions += svm_predict_vs(data[:, :-1], classifier)
    return numpy.sign(predictions / len(classifiers))

series = "creditDE"
data_params = {"filename": "creditDE.csv", "last_column_str": False}
algo_params_series = [{"c": 1, "ktype": "linear", "kparams": {}},
                      {"c": 1, "ktype": "polynomial", "kparams": {"degrees": [2]}},
                      {"c": 1, "ktype": "RBF", "kparams": {"sigma": 1.}}]

train_set, _, _ = load_csv(**data_params)
for algo_params in algo_params_series:
    classifiers = bagging(train_set, algo_params)
    test_set, _, _ = load_csv(**data_params)
    predictions = bagging_predict(test_set, classifiers)
```