EX02

May 5, 2025

1 Machine Learning Essentials SS25 - Exercise Sheet 1

1.1 Instructions

- `TODO`'s indicate where you need to complete the implementations.
- You may use external resources, but write your own solutions.
- Provide concise, but comprehensible comments to explain what your code does.
- Code that's unnecessarily extensive and/or not well commented will not be scored.

1.2 Exercise 2: The Perceptron Algorithm

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.pyplot import legend
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from IPython.display import display
```

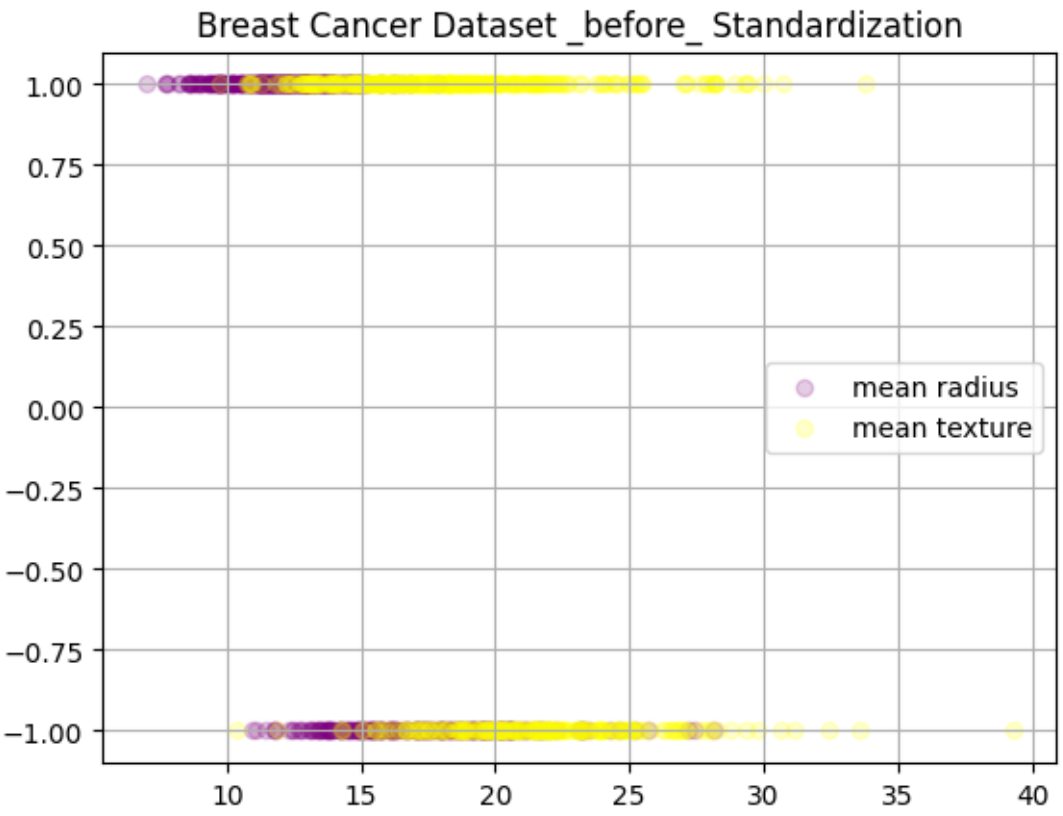
[6]:
```
def make_scatter_plot(X, x_values, y, title="Scatter Plot") -> None:
    """
    Creates a scatter plot for visualising data: y and the specified features␣
 ↪of X.
    Uses different colors for each feature.

    :param X: DataFrame containing the features
    :param x_values: List of feature names to plot
    :param y: Target variable
    :param title: Title of the plot
    :return: None
    """
    fig, ax = plt.subplots()

    colors = ['purple', 'yellow', 'green', 'blue', 'red', 'orange']
    for value, color in zip(x_values, colors):
        # alpha = opacity
        ax.scatter(X[value], y, c=color, label=value, alpha=0.2)

    ax.legend()
```

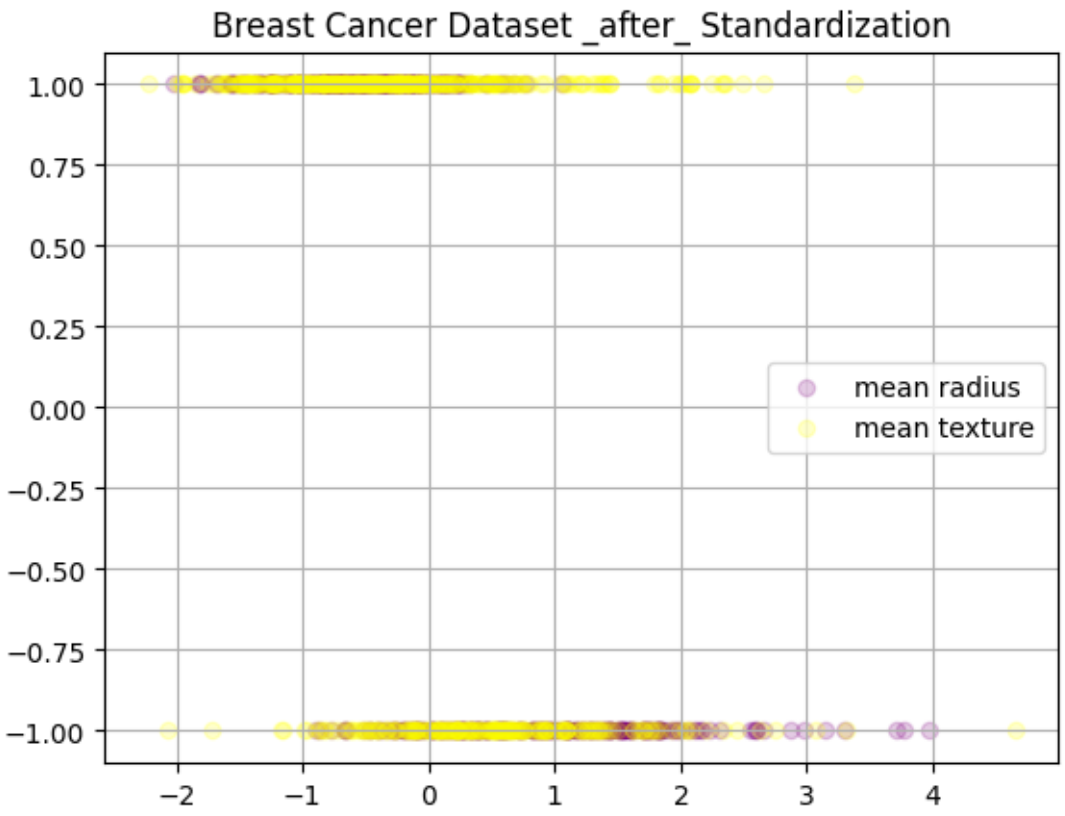
```
    ax.set_title(title)
    ax.grid(True)
    plt.show()
```

[7]:
```
# ================================
# 1. Load & Visualize the Dataset
# ================================

# + TODO: Load dataset, print feature names
dataset = load_breast_cancer(return_X_y=True, as_frame=True)
print("Feature Names:", *dataset[0].keys(), sep='\n\t- ', end='\n\n')

# + TODO: Select features & corresponding labels
features = ['mean radius', 'mean texture']
X = dataset[0][features]
# target labels
y = dataset[1]

# Convert labels from {0,1} to {-1,1} to match Perceptron convention from sheet
y = 2 * (y - 0.5)

# Before the standardization, the data instances look hardly separable, they␣
 ↪seem to be more uniform.
make_scatter_plot(X, features, y, "Breast Cancer Dataset _before_␣
 ↪Standardization")

# + TODO: Standardize the data to zero mean and unit variance, explain why it's␣
 ↪useful
X = (X - X.mean()) / X.std()
# Standardising data allows one to compare measurements from different people␣
 ↪excluding
# their differences (biases). Otherwise, the data might be not grouped␣
 ↪properly, leading
# to finding a wrong hyperplane or not finding it at all.

# + TODO: Visualize dataset using plt.scatter()
make_scatter_plot(X, features, y, "Breast Cancer Dataset _after_␣
 ↪Standardization")
# We observe that 'mean radius' values are set a bit further apart than 'mean␣
 ↪texture' values,
# potentially making them easier to separate.
```

```
Feature Names:
        - mean radius
        - mean texture
        - mean perimeter
        - mean area
```

- mean smoothness
- mean compactness
- mean concavity
- mean concave points
- mean symmetry
- mean fractal dimension
- radius error
- texture error
- perimeter error
- area error
- smoothness error
- compactness error
- concavity error
- concave points error
- symmetry error
- fractal dimension error
- worst radius
- worst texture
- worst perimeter
- worst area
- worst smoothness
- worst compactness
- worst concavity
- worst concave points
- worst symmetry
- worst fractal dimension


Breast Cancer Dataset _before_ Standardization
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
10
15
20
25
30
35
40
mean radius
mean texture



[56]:
```python
# ========================
# 2. Implement the Perceptron's training algorithm
# ========================

class Perceptron:
    def __init__(self, learning_rate=0.1, num_epochs=10):
        self.learning_rate = learning_rate
        self.num_epochs = num_epochs
        self.w = None  # Weights
        self.b = None # Bias
        self.history = [] # Store parameters for decision boundary @ each␣
 ↪update for visualization

    def train(self, X, y):
        """Train the perceptron using the online Perceptron algorithm."""
        # n_samples = N, n_features = D
        n_samples, n_features = X.shape
        # + TODO: Initialize weights and bias
        self.w = np.random.randn(n_features)
        self.b = np.random.randn()
```

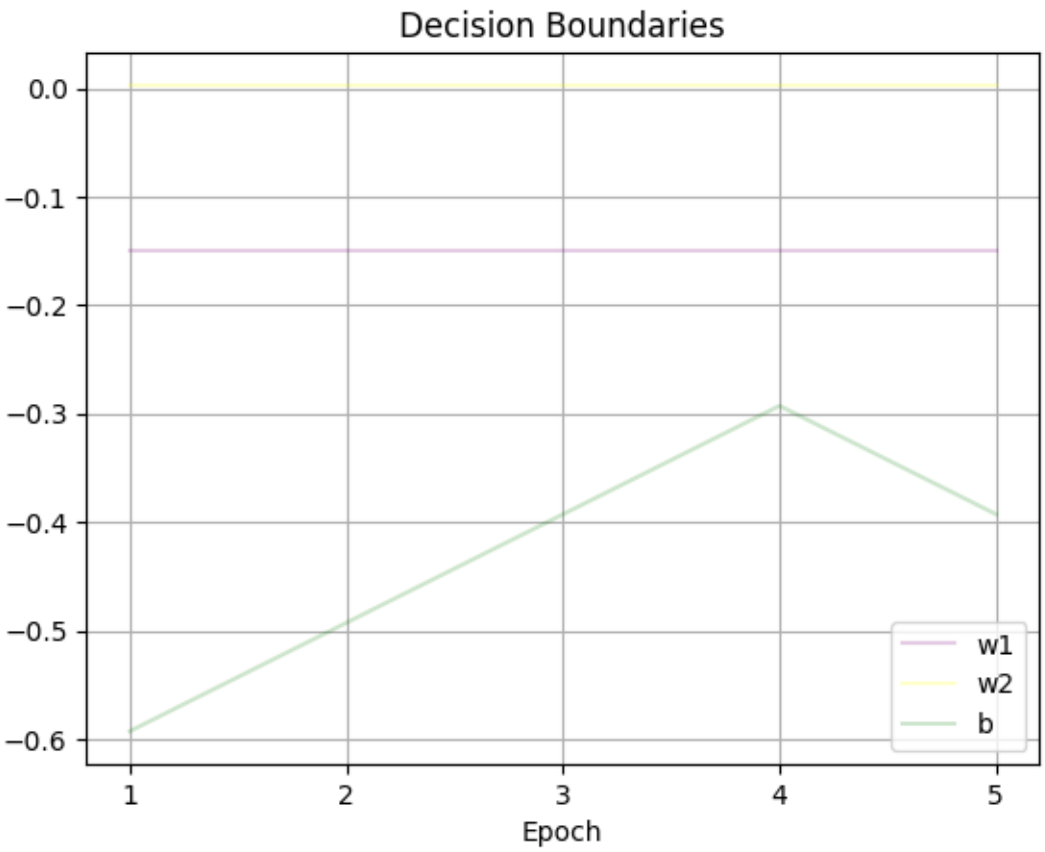
```
        # Train for num_epochs iterations
        for _ in range(self.num_epochs):
            for i in range(n_samples):
                # I used ChatGPT to figure out how to access the i-th row of a␣
 ↪DataFrame
                X_i = X.iloc[i].values
                y_i = y.iloc[i]

                # + TODO: Implement the update rule
                if self.predict(X_i) != y_i:
                    multiplication = self.learning_rate * y_i
                    self.w += multiplication * X_i
                    self.b += multiplication
                    self.history.append((self.w, self.b)) # Save state for␣
 ↪visualization

    def predict(self, X):
        """Predict the label of a sample."""
        # + TODO: Implement the prediction function
        # With weights transposed, the dimensions were not compatible
        return np.sign(self.w.dot(X.transpose()) + self.b)
```

[57]:
```
def get_accuracy(y_true, y_pred):
    """Compute the accuracy of the predictions."""
    return np.mean(y_true == y_pred)
```

[58]:
```
# ========================
# 3. Train the Perceptron & Evaluate Performance
# ========================

# + TODO: Split the data into training and test set
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8,␣
 ↪random_state=14)

# + TODO: Initialize the Perceptron and train it on the training set
perceptron = Perceptron()
perceptron.train(X_train, y_train)

# + TODO: Use the trained Perceptron to compute the accuracy on the training␣
 ↪set and on the test set
train_acc = get_accuracy(perceptron.predict(X_train), y_train)
test_acc = get_accuracy(perceptron.predict(X_test), y_test)

print(f"Training Accuracy: {train_acc:.3f}")
print(f"Test Accuracy: {test_acc:.3f}")
```

```
Training Accuracy: 0.824
Test Accuracy: 0.833
```

[75]:
```python
# ========================
# 4. Plot decision boundary evolution
# ========================

# Visualize the first 5 consecutive decision boundaries for data
decision_boundaries = perceptron.history[:5] # Get the parameters of the first␣
 ↪5 decision boundaries used during training

# + TODO: Plot decision boundaries for iterations 1-5
w1 = [w[0] for w, b in decision_boundaries]
w2 = [w[1] for w, b in decision_boundaries]
b = [b for w, b in decision_boundaries]

fig, ax = plt.subplots()
labels = ["w1", "w2", "b"]
colors = ['purple', 'yellow', 'green']
for value, color, label in zip((w1, w2, b), colors, labels):
    # alpha = opacity
    ax.plot(value, c=color, label=label, alpha=0.2)

ax.legend()
ax.set_title("Decision Boundaries")
ax.grid(True)
ax.set_xlabel("Epoch")
ax.set_xticks(range(5))
ax.set_xticklabels(range(1, 6))
plt.show()
```



[78]:
```
len(perceptron.history)
```

[78]: 796

1.2.1 5.

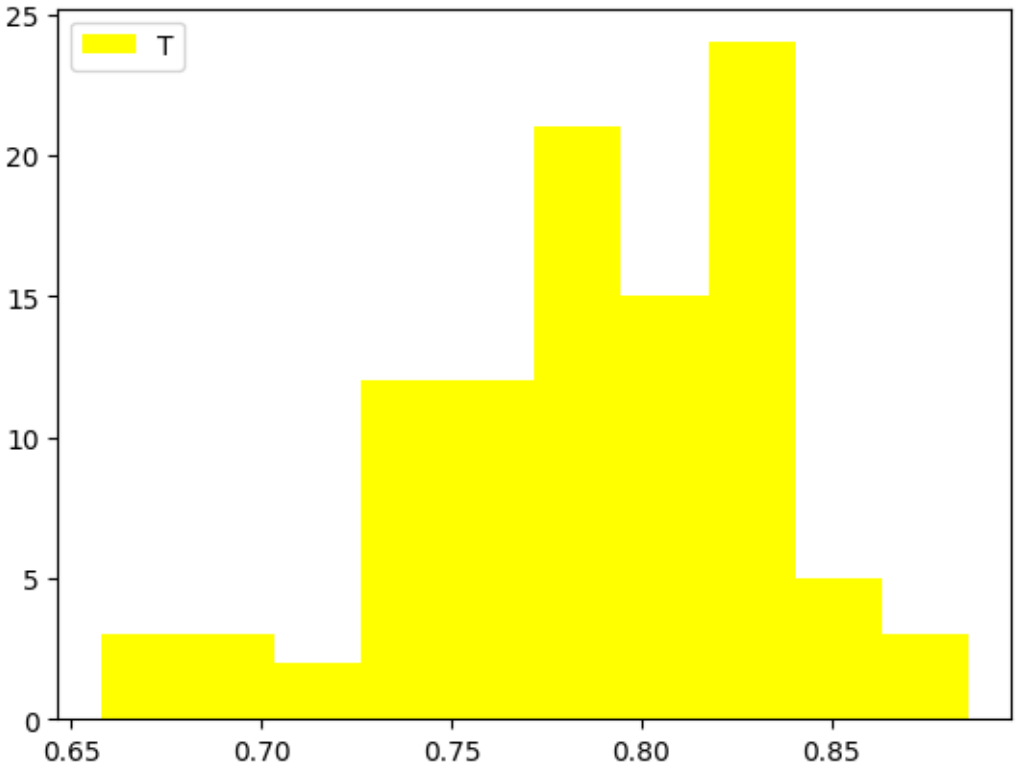
- TODO: How many updates do you need until convergence (i.e. until no more model updates occur)? Explain why.

Around 750-800 updates were necessary to reach convergence. The model arrived at a point where either it could predict any label correctly or the epochs went out. We see that the performance on the testing set is even higher than on the training set, so it generalized well.

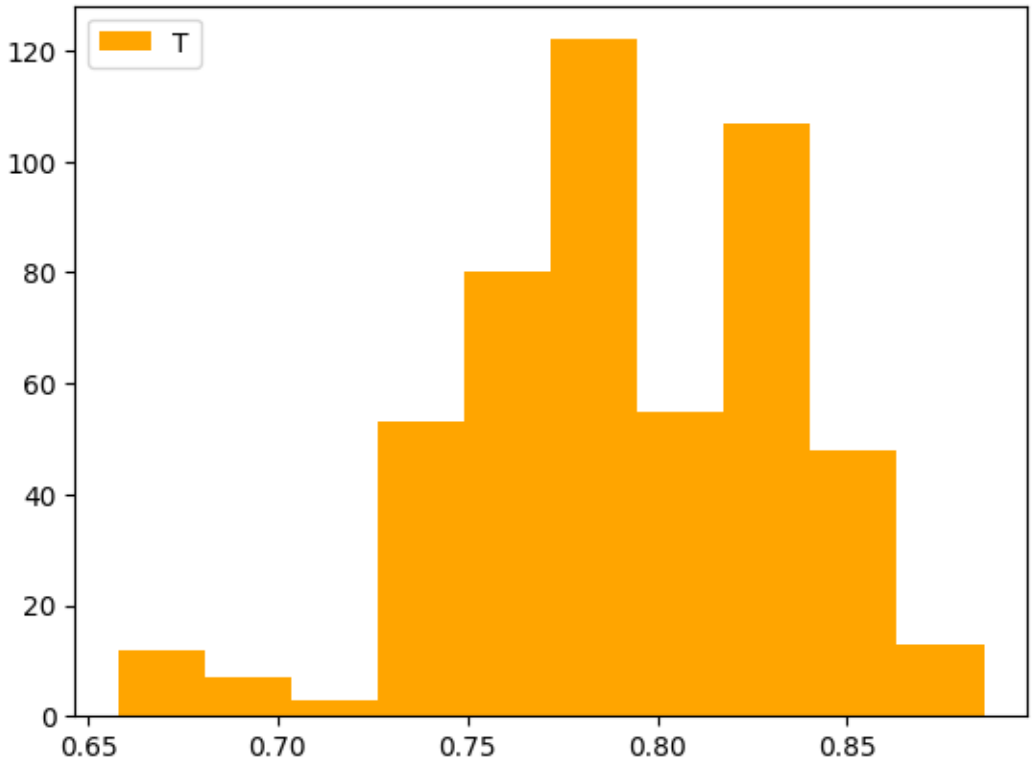
[80]:
```
# ========================
# 6. Evaluate Performance Over Multiple Runs
# ========================
test_accuracies = []
# + TODO: Evaluate performance over multiple runs. Compute and store test␣
 ↪accuracies
for _ in range(500):
```

```
    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8,␣
↪random_state=14)

    perceptron = Perceptron()
    perceptron.train(X_train, y_train)

    test_accuracies.append(get_accuracy(perceptron.predict(X_test), y_test))
```

[79]:
```
# TODO: Plot histogram for the test accuracies
# WITH 100 RUNS
plt.hist(test_accuracies, color="yellow")
legend("Test Accuracies", loc="upper left")
plt.show()
```



[81]:
```
# TODO: Plot histogram for the test accuracies
# WITH 500 RUNS
plt.hist(test_accuracies, color="orange")
legend("Test Accuracies", loc="upper left")
plt.show()
```



1.2.2 (a)

- TODO: What does the shape of the histogram tell you?

At the first glance, the histogram looks as a bell-curve with gaps. I assumed that with more runs, the distribution would approximate a normal distribution closer but with 500 runs instead of 100, it didn't happen.

The range of accuracies is between 0.7 and 0.9, with the most frequent values around 0.73 and 0.83. None of them is prevalent.

On the good side, the mean and the median are almost equal, and the standard deviation is relatively low, which indicates that the distribution is not skewed.

[4]:
```
# (b)
# + TODO: Compute the sample mean and standard deviation of the test accuracy
```

[82]:
```
np.mean(test_accuracies)
```

[82]:
```
np.float64(0.790719298245614)
```

[83]:
```
np.median(test_accuracies)
```

[83]:
```
np.float64(0.7894736842105263)
```

[84]:
```
np.std(test_accuracies)
```

[84]:
```
np.float64(0.044419095187048466)
```

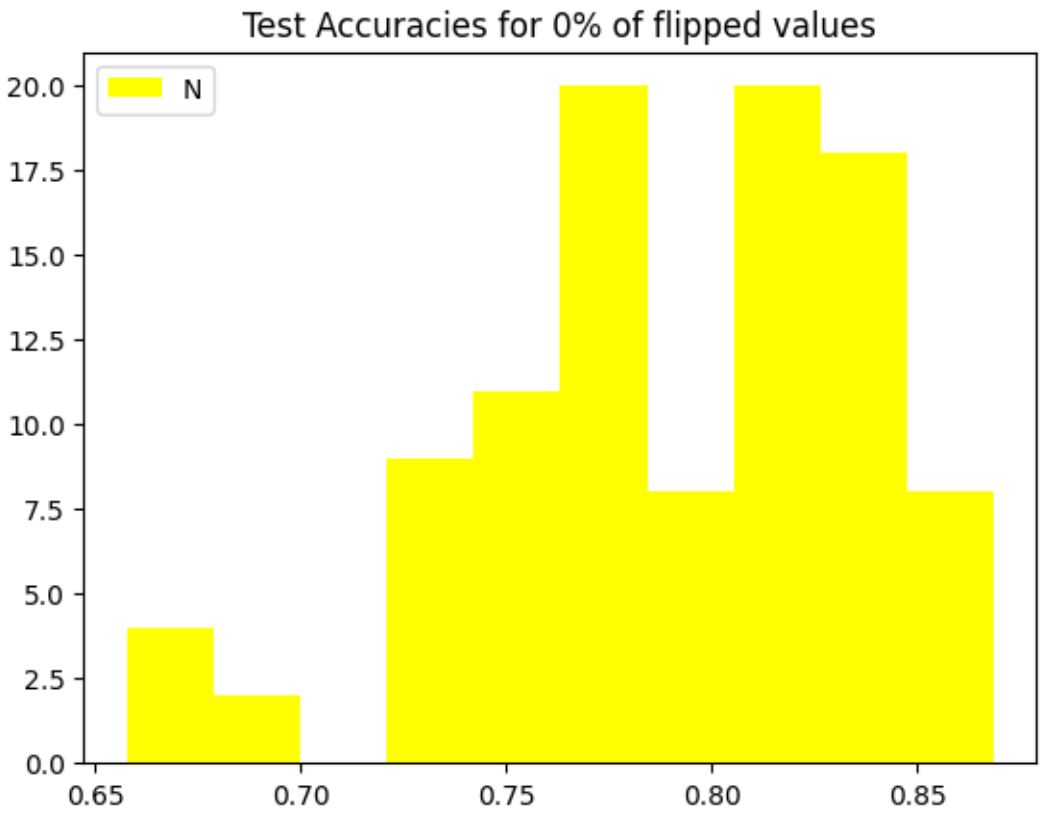
1.2.3 (c)

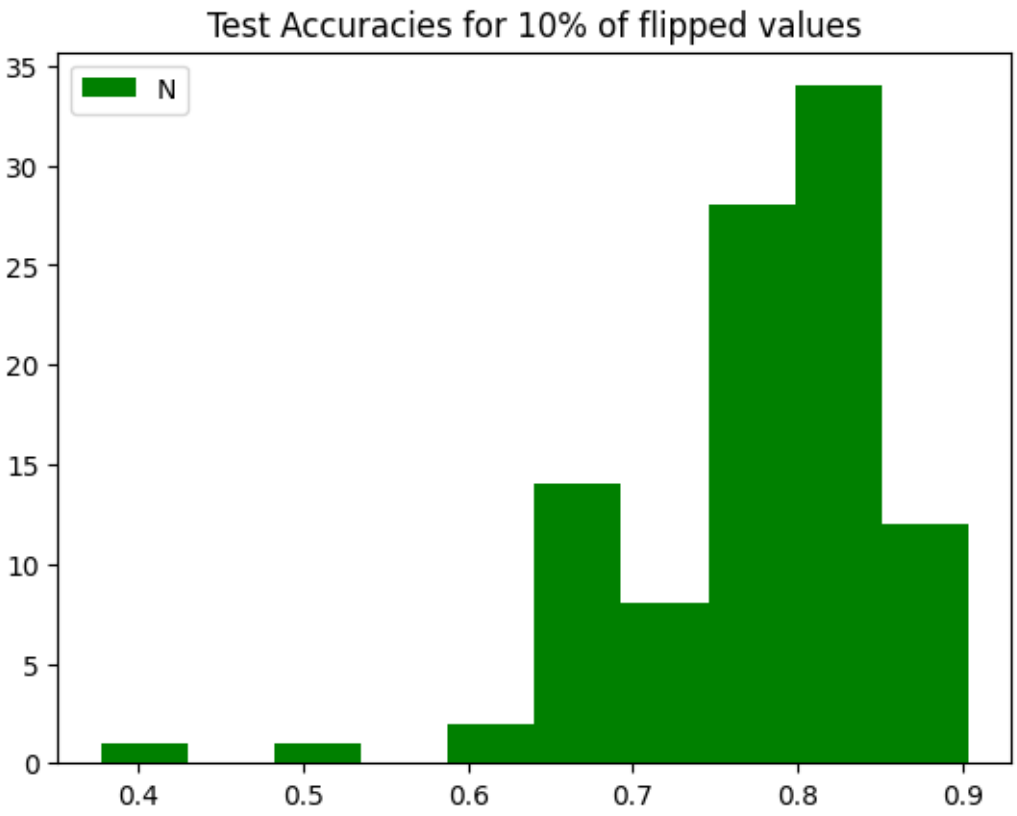
- TODO: Given enough data points and for many training runs, what type of probability distribution would the histogram approximate and what is the reason for that?

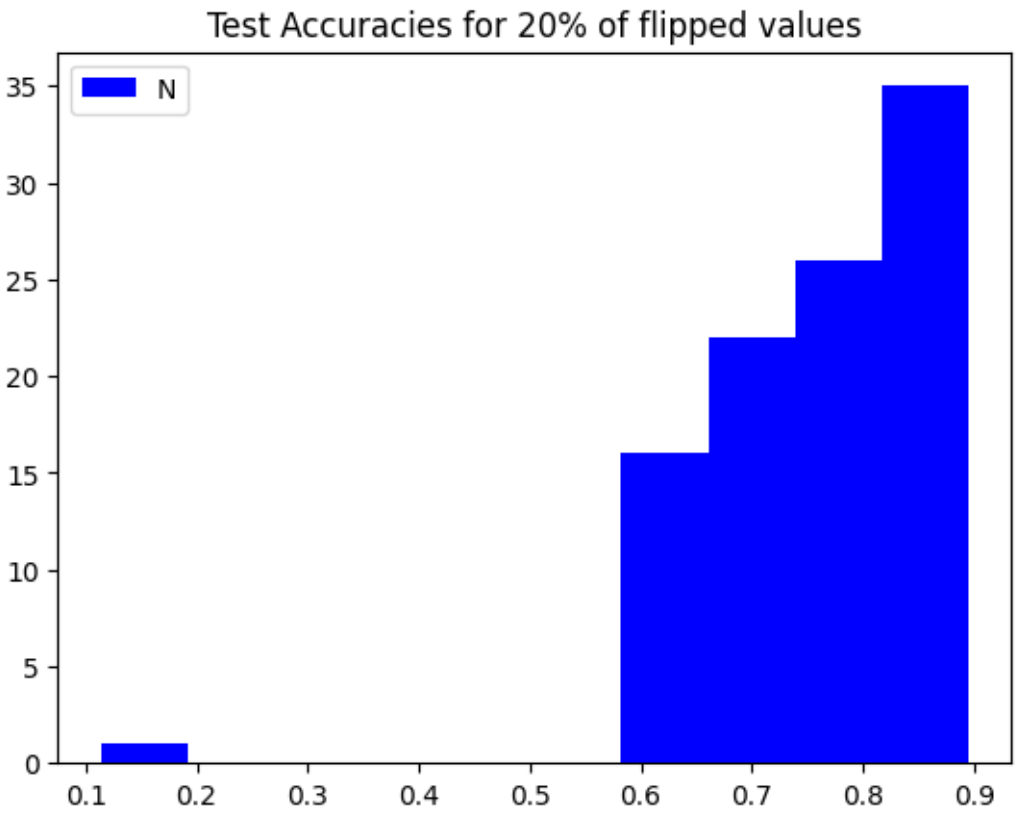
The histogram would approach the bimodal distribution. I suppose that the peaks might correspond to the two features we based the training on.

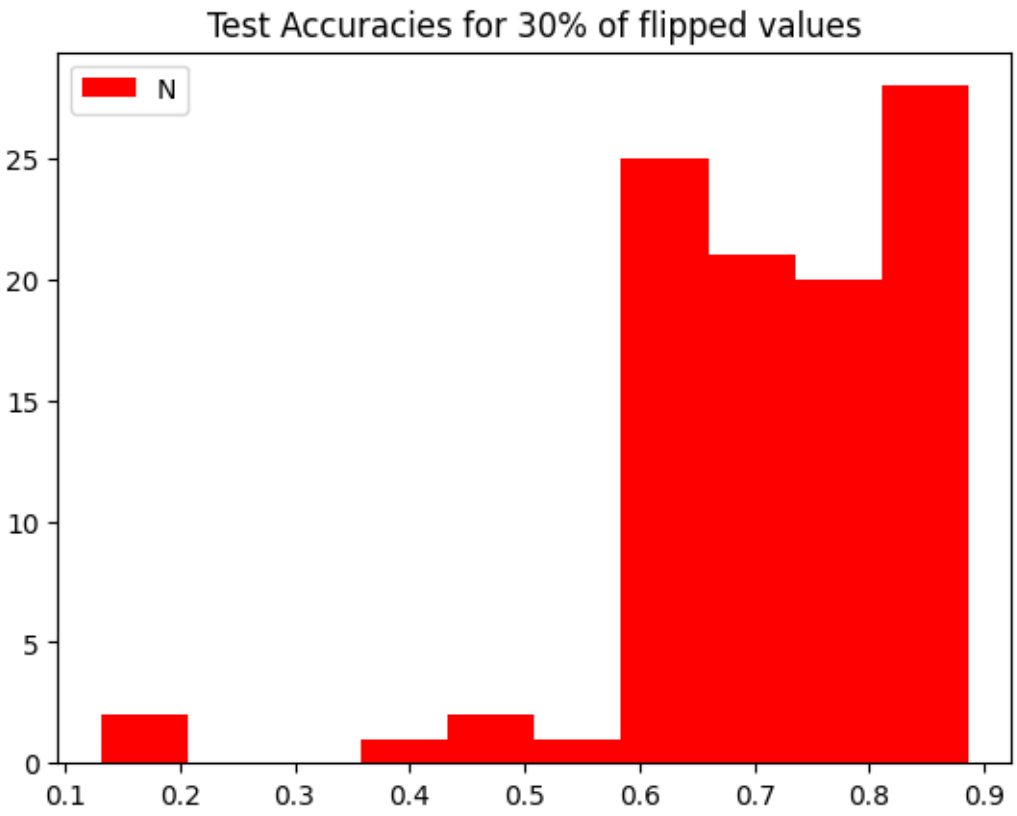
[85]:
```
# (d)
#TODO: Add noise by flipping p% of labels. Visualize the effect using␣
 ↪histograms for each p.

test_accuracies = {}
p_values = [0, 10, 20, 30, 40, 50] # % of flipped training labels
for p_value in p_values:
    test_accuracies[p_value] = []
    for _ in range(100):
        X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.
 ↪8, random_state=14)

        flipped_values = 0
        for random_i in np.random.permutation(range(len(y_train))):
            if flipped_values >= p_value:
                break
            # Flip the labels
            y_train.iloc[random_i] = - y_train.iloc[random_i]
            flipped_values += 1

        perceptron = Perceptron()
        perceptron.train(X_train, y_train)

        test_accuracies[p_value].append(get_accuracy(perceptron.
 ↪predict(X_test), y_test))
```

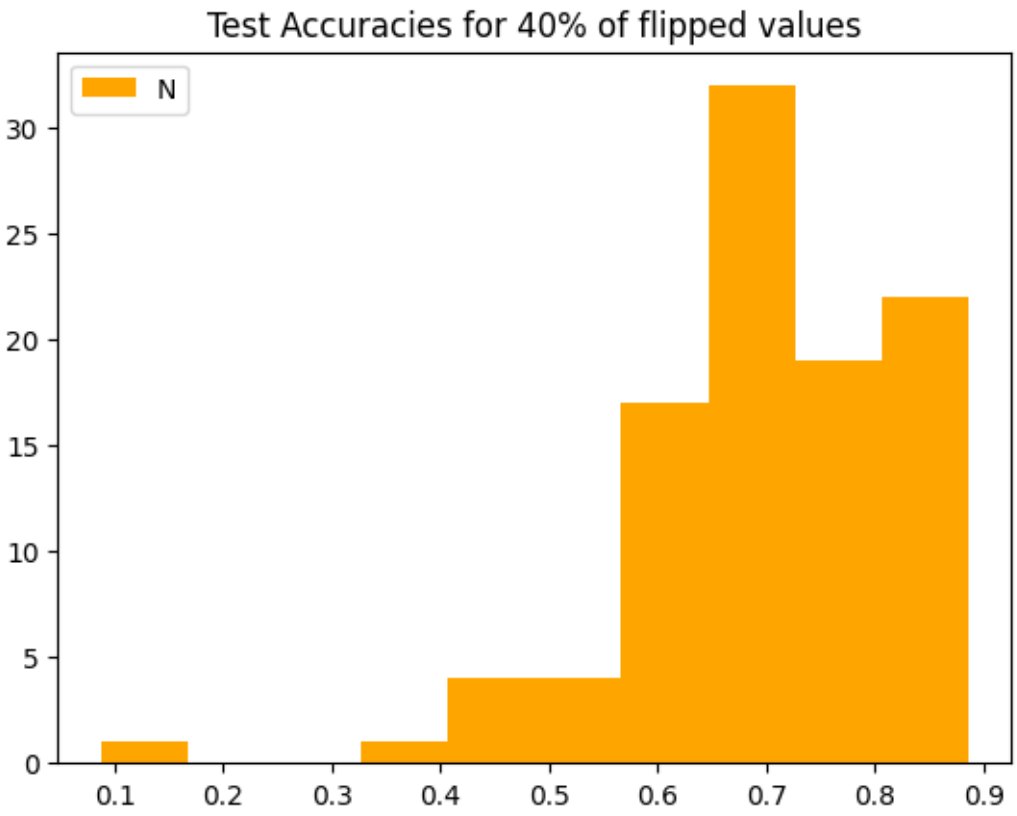
[87]:
```
colors = ['yellow', 'green', 'blue', 'red', 'orange', 'purple']
for (p_value, accuracies), color in zip(test_accuracies.items(), colors):
    plt.hist(accuracies, color=color)
    legend(f"Number of Accuracies", loc="upper left")
    plt.title(f"Test Accuracies for {p_value}% of flipped values")
    plt.show()
```

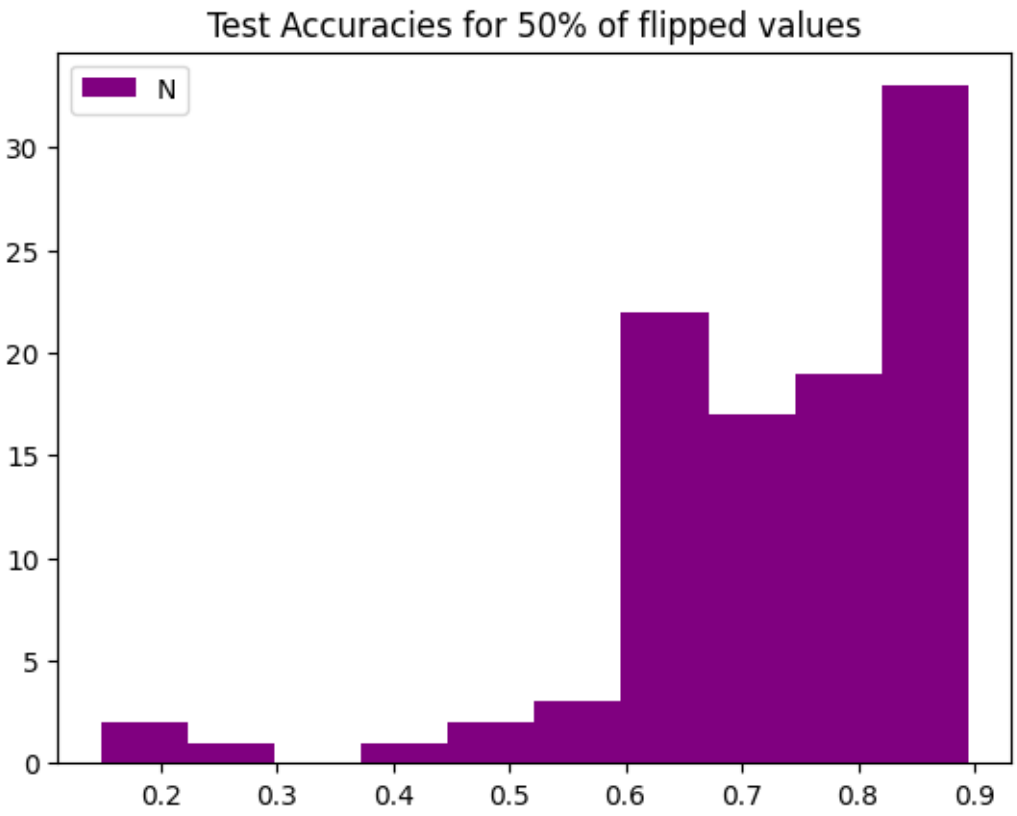

Test Accuracies for 0% of flipped values
N
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.65
0.70
0.75
0.80
0.85


Test Accuracies for 10% of flipped values
N
35
30
25
20
15
10
5
0
0.4
0.5
0.6
0.7
0.8
0.9


Test Accuracies for 20% of flipped values
N
35
30
25
20
15
10
5
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


Test Accuracies for 30% of flipped values
N
25
20
15
10
5
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9


Test Accuracies for 40% of flipped values
N
30
25
20
15
10
5
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9



[89]:
```
display(np.mean(test_accuracies[20]))
display(np.median(test_accuracies[20]))
display(np.std(test_accuracies[20]))
```

```
np.float64(0.7599122807017545)
```

```
np.float64(0.7719298245614035)
```

```
np.float64(0.10351364097974108)
```

[90]:
```
display(np.mean(test_accuracies[50]))
display(np.median(test_accuracies[50]))
display(np.std(test_accuracies[50]))
```

```
np.float64(0.7275438596491226)
```

```
np.float64(0.7456140350877193)
```

```
np.float64(0.14046497555164827)
```

- TODO: Interpret the results

We can clearly observe that the noise introduced randomness into the models, leading to both way worse (10% of accuracy) and slightly better (90% of accuracy) models. The distributions generally stayed the same, except for the one with 20% noise, which doesn't look bimodal anymore but rather

unsymmetrical, skewed to the right. Furthermore, the mean and median got a bit higher, while still staying close to each other. The standard deviation increased a bit, too, which seems a logical consequence of randomness.

On the other hand, for the case of 50% noise, the mean and median didn't degrade much, but the standard deviation increased significantly. Therefore, the deviation seems to be a more reliable metric than the mean or median.

I think that if there were drastic improvements happening under this setting, it would mean that the dataset quality has to be questioned.

Task_03

May 4, 2025

0.1 Exercise 3: SVM

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles, make_blobs
from cvxopt import matrix, solvers # Install cvxopt via "pip install cvxopt"
```

[2]:
```
# ========================
# 1. Complete SVM implementation
# ========================

#Note for correctors: I comment more than Van Rossum would prefer, comments␣
 ↪include links to websites.
#This first version is a bit incosistent. I underestimated the workload a bit,␣
 ↪so yeah...
#There are several things wrong with the code and I will be really pleased to␣
 ↪see the solution...

class DualSVM:
    def __init__(self, C=1.0, kernel="linear", gamma=1.0):
        self.C = C # Regularization constant
        self.kernel = kernel # Kernel type: "linear" or "rbf"
        self.gamma = gamma # Kernel parameter ("bandwith")
        self.alpha = None # Lagrange multipliers
        self.sv_X = None # Support vectors
        self.sv_y = None # Support vector labels
        self.w = None # Weights
        self.b = None # Bias

    def linear_kernel(self, X1, X2) -> np.array:
        #interestingly enough, I found all kinds of versions for the linear␣
 ↪kernel: transpoe x1. no, x2... and others didn't even bother at all
        #this version was the one that worked first, after the 1 to 1␣
 ↪implementation of the instructions seemed to fail.
        return np.dot(X1,X2.T)

    def rbf_kernel(self, X1, X2):
```

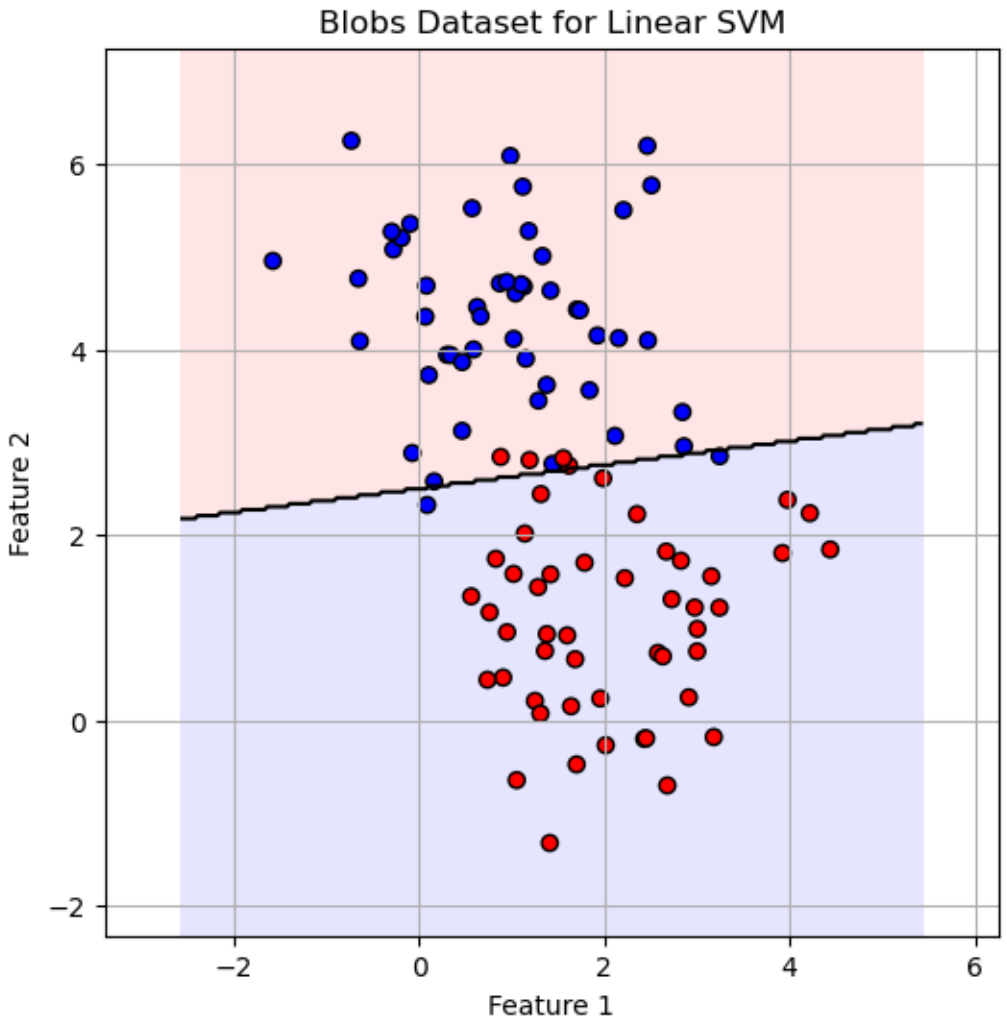
```
        # there are a few ways to compute the rbf kernel, notably different are␣
↪the one for vectors and for matrixes :


        """was the version I found first, but it failed me in task 5
        return np.exp(-self.gamma * np.linalg.norm(X1 - X2)**2)"""

        # https://github.com/xbeat/Machine-Learning/blob/main/
↪The%20Mathematics%20of%20RBF%20Kernel%20in%20Python.md:
        X1_sq = np.sum(X1**2, axis=1).reshape(-1, 1)
        X2_sq = np.sum(X2**2, axis=1).reshape(1, -1)
        dist_sq = X1_sq + X2_sq - 2 * np.dot(X1, X2.T)
        return np.exp(-self.gamma * dist_sq)


    def compute_kernel(self, X1, X2):
        if self.kernel == "linear":
            return self.linear_kernel(X1, X2)
        elif self.kernel == "rbf":
            return self.rbf_kernel(X1, X2)
        else:
            raise ValueError("Unknown kernel type.")

    def fit(self, X, y):
        #in fit, we are refering back to the dual formulation we were given at␣
↪the beginning (hence optimization objective & constraints)
        n_samples, n_features = X.shape

        # Compute kernel matrix K: K[i,j] = K(x_i, x_j)
        K = self.compute_kernel(X, X)

        """
        The dual objective is:
            max  sum_i alpha_i - 1/2 sum_i sum_j alpha_i alpha_j y_i y_j K(x_i,␣
↪x_j)
        subject to:
            sum_i alpha_i y_i = 0  and  0 <= alpha_i <= C for all i.
        In QP formulation:
            P = (y_i y_j K(x_i,x_j))_{i,j},   q = -1 (vector),
            A = y^T, b = 0, and G, h implement 0 <= alpha_i <= C.
        """

        #It seems to be recommemded to reshape y for cvxopt into a 2D array␣
↪column vector
        Y = y.reshape(-1, 1)
        #To create the y_i * y_j matrix part of the formula, we are recommended␣
↪to do
```

```
        yiyj = np.outer(y,y)
        P = matrix(yiyj * K )
        q = matrix(-np.ones(n_samples))
        A = matrix(y.reshape(-1,1).astype("double"), (1, n_samples),"d") # Use "d" flag to make sure that the matrix is in double precision format (labels are integers)
        b = matrix(0.0)


        #after the inequality constraints in the enoncé, we require some prerequisits to do G and h
        I_minus = - np.eye(n_samples)
        I = np.eye(n_samples)

        G = matrix(np.vstack((I_minus, I)))

        zeros = np.zeros(n_samples)
        c_matrix = np.ones(n_samples) * self.C
        h = matrix(np.hstack((zeros, c_matrix)), tc="d")

        # Solve the QP problem using cvxopt
        solvers.options["show_progress"] = False
        solution = solvers.qp(P, q, G, h, A, b)
        alphas = np.ravel(solution["x"]) # Get optimal alphas

        # Get support vectors (i.e. data points with non-zero lagrange multipliers, that are on the margin)
        sv = alphas > 1e-5 # alpha > 1e-5 to account for numerical errors
        self.alpha = alphas[sv]
        self.sv_X = X[sv]
        self.sv_y = y[sv]

        # The bias corresponds to the average error over all support vectors:
        # Why does the bias corresponds to the average error over all support vectors?
        # The answer is that the bias is the average of the differences between the true labels and the predicted labels
        # for the support vectors. The predicted labels are computed by the decision function f(x) = sum(alpha_i y_i K(x,x_i)) + b.
        # The difference between the true labels and the predicted labels is the error for each support vector.
        # The bias is the average of these errors.
        self.b = np.mean(self.sv_y - np.sum(self.alpha * self.sv_y * K[sv][:, sv], axis=1))
```

```
    def predict(self, X):
        y_pred = []
        for x in X:
            s = 0
            for alpha_i, y_i, x_i in zip(self.alpha, self.sv_y, self.sv_X):
                s += alpha_i * y_i * self.compute_kernel(x, x_i)
            y_pred.append(s + self.b)
        return np.sign(y_pred)
```

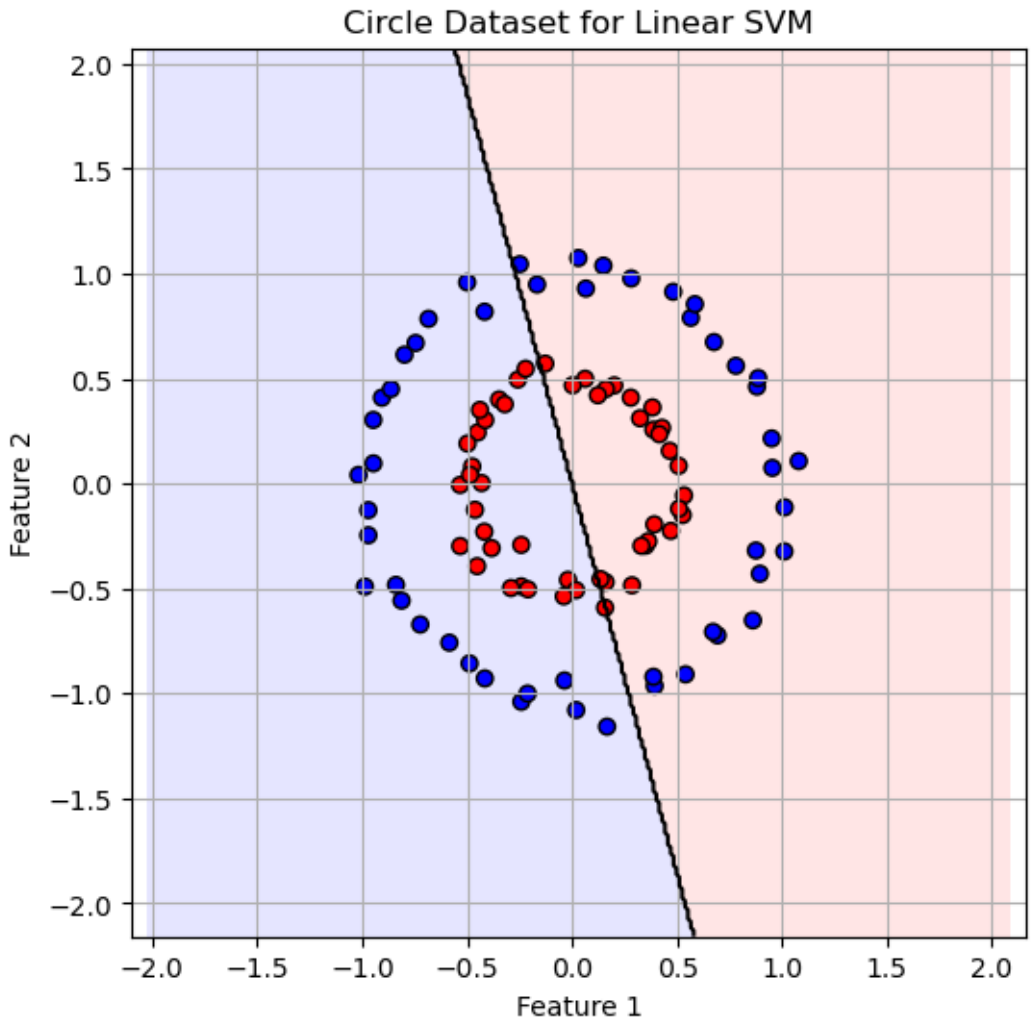
[3]:
```
# ========================
# 2. Apply linear SVM on blobs
# ========================

# TODO: Generate blobs dataset
X_linear, y_linear = make_blobs(n_samples=100, centers=2, n_features=2,
                  random_state=0)

# Convert labels from {0,1} to {-1,1}
y_linear = 2 * (y_linear - 0.5)

#TODO: Train SVM with linear kernel
SVM_Linear = DualSVM()
SVM_Linear.fit(X_linear,y_linear)

#TODO: Plot decision boundary

# most of the following instructions are purely to make it prettier.
plt.figure(figsize=(6, 6))
plt.title("Blobs Dataset for Linear SVM")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.grid(True)
plt.axis("equal")

# Draw decision boundary and margin
# There are a few extra steps to making the decision boundary onto the graph:␣
 ↪create a meshgrid, contour it (and color it in case you want it a bit more␣
 ↪chique)
xx, yy = np.meshgrid(
    np.linspace(X_linear[:, 0].min() - 1, X_linear[:, 0].max() + 1, 300),
    np.linspace(X_linear[:, 1].min() - 1, X_linear[:, 1].max() + 1, 300)
)
grid = np.c_[xx.ravel(), yy.ravel()]  # shape (300*300, 2)

# Compute predictions over the grid
Z = SVM_Linear.predict(grid).reshape(xx.shape)
```

```
plt.contourf(xx, yy, Z, levels=[-1, 0, 1], colors=["#FFAAAA", "#AAAAFF"],␣
  ↪alpha=0.3)
plt.contour(xx, yy, Z, levels=[0], colors='k', linewidths=1.5)


plt.scatter(X_linear[:, 0], X_linear[:, 1], c=y_linear, cmap="bwr",␣
  ↪edgecolors="k")
plt.show()
```



[4]:
```
# ============================
# 3. Apply linear SVM on circles
# ============================

# TODO: Generate blobs dataset
```

```
X_circles, y_circles = make_circles(n_samples=100, noise=0.05, factor=0.5,␣
 ↪random_state=0)
y_circles = 2 * (y_circles - 0.5)  # Convert labels from {0,1} to {-1,1}

# TODO: Train SVM with linear kernel
SVM_Linear_Circle = DualSVM()
SVM_Linear_Circle.fit(X_circles, y_circles)

#TODO: Plot decision boundary

xx_2, yy_2 = np.meshgrid(
    np.linspace(X_circles[:, 0].min() - 1, X_circles[:, 0].max() + 1, 300),
    np.linspace(X_circles[:, 1].min() - 1, X_circles[:, 1].max() + 1, 300)
)
new_grid = np.c_[xx_2.ravel(), yy_2.ravel()]
Z_2 = SVM_Linear_Circle.predict(new_grid).reshape(xx_2.shape)

plt.figure(figsize=(6, 6))
plt.title("Circle Dataset for Linear SVM")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.grid(True)
plt.axis("equal")
plt.contourf(xx_2, yy_2, Z_2, levels=[-1, 0, 1], colors=["#FFAAAA", "#AAAAFF"],␣
 ↪alpha=0.3)
plt.contour(xx_2, yy_2, Z_2, levels=[0], colors='k', linewidths=1.5)

plt.scatter(X_circles[:, 0], X_circles[:, 1], c=y_circles, cmap="bwr",␣
 ↪edgecolors="k")
plt.show()
```



[5]:
```
# ============================
# 4. Apply feature transformation
# ============================
def transform_features(X):
    # TODO: compute feature transformation: f(x) = [x1, x2, x1^2 + x2^2]
    x1, x2 = X_circles[:,0], X_circles[:,1]
    last = x1**2 + x2**2
    return np.column_stack([x1, x2, last])


#TODO: Train SVM with linear kernel on transformed features
SVM_Linear_Circle_tf = DualSVM()

SVM_Linear_Circle_tf.fit(transform_features(X_circles), y_circles)
```

```
def plot_decision_boundary_transformed(X, y, model, title="SVM Decision␣
 ↪Boundary (Transformed)"):
    # TODO: Implement plotting function for decision boundary in the␣
 ↪transformed feature space
    # Hint: You could do this by creating a 2D meshgrid which you transform␣
 ↪using the feature mapping.
    # Then, after evaluating the model on it, you can plot the result as a␣
 ↪contour plot (plt.contourf).

    xx, yy = np.meshgrid(
    np.linspace(X[:, 0].min() - 1, X[:, 0].max() + 1, 300),
    np.linspace(X[:, 1].min() - 1, X[:, 1].max() + 1, 300)
    )
    grid = np.c_[xx.ravel(), yy.ravel()]
    grid_transformed = transform_features(grid)
    Z = model.predict(grid_transformed).reshape(xx.shape) # it always fails␣
 ↪here due to reshaping, can't seem to fix it.
    #I know the error has to do with the transformation, but I can't seem to␣
 ↪find the right way to do it.

    plt.figure(figsize=(6, 6))
    plt.title("Circle Dataset for Linear SVM")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.grid(True)
    plt.axis("equal")
    plt.contourf(xx, yy, Z, levels=[-1, 0, 1], colors=["#FFAAAA", "#AAAAFF"],␣
 ↪alpha=0.3)
    plt.contour(xx, yy, Z, levels=[0], colors='k', linewidths=1.5)
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap="bwr", edgecolors="k")
    plt.show()

#TODO: Plot decision boundary in the transformed feature space
plot_decision_boundary_transformed(X_circles, y_circles, SVM_Linear_Circle_tf,␣
 ↪title="SVM Decision Boundary with Feature Transformation")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[5], line 42
     39     plt.show()
     41 #TODO: Plot decision boundary in the transformed feature space
---> 42␣
 ↪plot_decision_boundary_transformed(X_circles, y_circles, SVM_Linear_Circle_tf  title=

Cell In[5], line 27, in plot_decision_boundary_transformed(X, y, model, title)
     25 grid = np.c_[xx.ravel(), yy.ravel()]
```

```
    26 grid_transformed = transform_features(grid)
---> 27 Z = model.predict(grid_transformed).reshape(xx.shape) # it always fails here due to reshaping, can't seem to fix it.
    28 #I know the error has to do with the transformation, but I can't seem to find the right way to do it.
    30 plt.figure(figsize=(6, 6))

ValueError: cannot reshape array of size 100 into shape (300,300)
```

[6]:
```
# ============================
# 5. SVM with RBF Kernel on Circular Data
# ============================

#TODO: Train SVM with RBF kernel on circular data
SVM_RBF = DualSVM(1., "rbf")
SVM_RBF.fit(X_circles, y_circles)




#TODO: Plot decision boundary

xx_3, yy_3 = np.meshgrid(
    np.linspace(X_circles[:, 0].min() - 1, X_circles[:, 0].max() + 1, 300),
    np.linspace(X_circles[:, 1].min() - 1, X_circles[:, 1].max() + 1, 300)
)
new_grid_2 = np.c_[xx_3.ravel(), yy_3.ravel()]
Z_3 = SVM_RBF.predict(new_grid_2).reshape(xx_3.shape) #Isn't this nice? Failing at the same place tiwce, but for different reasons.
#In this case the error is a dimension error.
plt.figure(figsize=(6, 6))
plt.title("Circle Dataset for RBF SVM")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.grid(True)
plt.axis("equal")
plt.contourf(xx_3, yy_3, Z_3, levels=[-1, 0, 1], colors=["#FFAAAA", "#AAAAFF"], alpha=0.3)
plt.contour(xx_3, yy_3, Z_3, levels=[0], colors='k', linewidths=1.5)
plt.scatter(X_circles[:, 0], X_circles[:, 1], c=y_circles, cmap="bwr", edgecolors="k")
plt.show()
```

```
---------------------------------------------------------------------------
AxisError                                 Traceback (most recent call last)
Cell In[6], line 19
```

```
    14 xx_3, yy_3 = np.meshgrid(
    15     np.linspace(X_circles[:, 0].min() - 1, X_circles[:, 0].max() + 1,␣
 ↪300),
    16     np.linspace(X_circles[:, 1].min() - 1, X_circles[:, 1].max() + 1,␣
 ↪300)
    17 )
    18 new_grid_2 = np.c_[xx_3.ravel(), yy_3.ravel()]
---> 19 Z_3 = SVM_RBF.predict(new_grid_2).reshape(xx_3.shape) #Isn't this nice?
 ↪Failing at the same place tiwce, but for different reasons.
    20 #In this case the error is a dimension error.
    21 plt.figure(figsize=(6, 6))

Cell In[2], line 109, in DualSVM.predict(self, X)
   107     s = 0
   108     for alpha_i, y_i, x_i in zip(self.alpha, self.sv_y, self.sv_X):
--> 109         s += alpha_i * y_i * self.compute_kernel(x, x_i)
   110     y_pred.append(s + self.b)
   111 return np.sign(y_pred)

Cell In[2], line 43, in DualSVM.compute_kernel(self, X1, X2)
    41     return self.linear_kernel(X1, X2)
    42 elif self.kernel == "rbf":
---> 43     return self.rbf_kernel(X1, X2)
    44 else:
    45     raise ValueError("Unknown kernel type.")

Cell In[2], line 33, in DualSVM.rbf_kernel(self, X1, X2)
    29 """was the version I found first, but it failed me in task 5
    30 return np.exp(-self.gamma * np.linalg.norm(X1 - X2)**2)"""
    32 # https://github.com/xbeat/Machine-Learning/blob/main/
 ↪The%20Mathematics%20of%20RBF%20Kernel%20in%20Python.md:
---> 33 X1_sq = np.sum(X1**2, axis=1).reshape(-1, 1)
    34 X2_sq = np.sum(X2**2, axis=1).reshape(1, -1)
    35 dist_sq = X1_sq + X2_sq - 2 * np.dot(X1, X2.T)

File /opt/homebrew/Caskroom/miniconda/base/envs/ml_homework/lib/python3.13/
 ↪site-packages/numpy/_core/fromnumeric.py:2466, in sum(a, axis, dtype, out,␣
 ↪keepdims, initial, where)
  2463         return out
  2464     return res
-> 2466 return _wrapreduction(
  2467     a, np.add,      , axis, dtype, out,
  2468     keepdims=keepdims, initial=initial, where=where
  2469 )

File /opt/homebrew/Caskroom/miniconda/base/envs/ml_homework/lib/python3.13/
 ↪site-packages/numpy/_core/fromnumeric.py:86, in _wrapreduction(obj, ufunc,␣
 ↪method, axis, dtype, out, **kwargs)
```

```
     83         else:
     84             return reduction(axis=axis, out=out, **passkwargs)
---> 86 return ufunc.reduce(obj, axis, dtype, out, **passkwargs)

AxisError: axis 1 is out of bounds for array of dimension 1
```

0.1.1 6.

TODO: Compare the decision boundaries from Tasks 3, 4, and 5. How does feature transformation differ from using an RBF kernel? When would one approach be preferable to the other?

//

0.1.2 7.

TODO: Besides the dual formulation, SVMs also have an equivalent primal formulation. The key factor in choosing which one to use as the optimization criterion is the dimensionality of the features. Explain why.