PROJECT 2 REPORT

SAM LEVINE

1. TASK 2A

1.1. **Perception (30pt).**

1.1.1. *Overview.*

The `fit` method in the Perceptron class trains the model by initializing weights and bias to zero and updating them iteratively based on the training data. It adjusts the weights and bias whenever a prediction is incorrect, using the input features and the true label to move the decision boundary. The `predict` method then uses the trained weights and bias to classify new samples, applying the sign function to the linear combination of input features and weights to determine the class labels.

1.1.2. *Implementation & Results.*

```
class Perceptron(object):

    def __init__(self, T=1):
        self.T = T # number of iterations

    def fit(self, X, y):
        """
        Train perceptron model on data X with labels y and iteration T.
        """
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features, dtype=np.float64)
        self.b = 0.0
        #### YOUR CODE HERE ####
        for _ in range(self.T):
            for i in range(n_samples):
                y_pred = np.dot(X[i], self.w) + self.b
                if y[i] * y_pred <= 0:
                    self.w += y[i] * X[i]
                    self.b += y[i]


    def predict(self, X):
        """
        Predict class labels for samples in X.
        Output should be a 1D array with shape (n_samples,)
        """
        X = np.atleast_2d(X)

        #### YOUR CODE HERE ####
        return np.sign(np.dot(X, self.w) + self.b)
```

We achieved an accuracy of 100% on the test set. This is the plotted decision boundary.

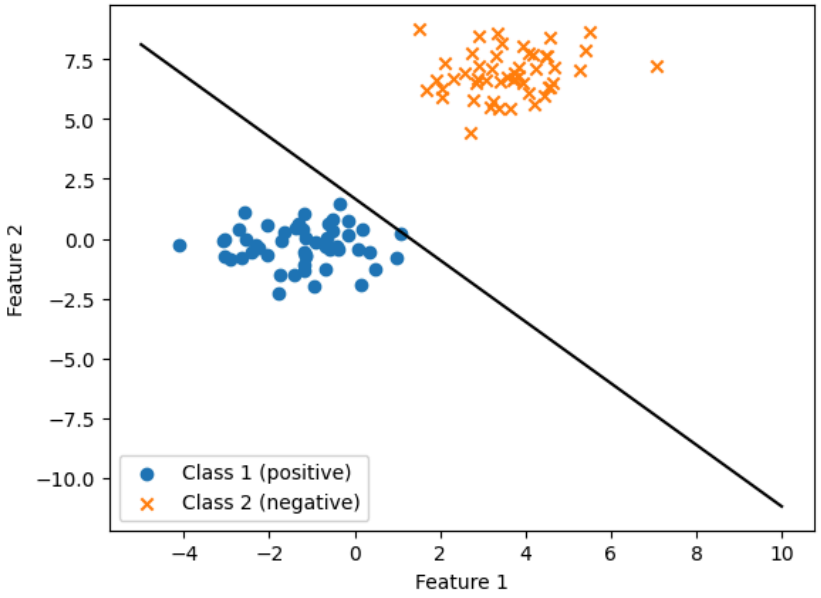

FIGURE 1. Accuracy = 100.00%

1.2. **Kernel Trick (40pt).**

1.2.1. *Overview.*

We implemented the three common kernels:

$$k_{\text{linear}}(x, x') = x^T x'$$

$$k_{\text{poly}}(x, x') = (1 + x^T x')^d$$

$$k_{\text{RBF}}(x, x') = \exp\left(-\frac{\|x - x'\|_2^2}{\sigma^2}\right)$$

The `KernelPerceptron` takes a Kernel function, and then has two key methods. The `fit` method calculates the Gram matrix using the chosen kernel and updates the alpha coefficients based on the training data. The `predict` method then uses these coefficients to classify new data points. This allows the perceptron to work with non-linearly separable data by mapping it into a higher-dimensional space.

1.2.2. *Implementation & Results.*

```
class LinearKernel:
    def __init__(self):
        pass
    def __call__(self, x, y):
        #### YOUR CODE HERE ####
        return np.dot(x, y)

class PolynomialKernel:
    def __init__(self, p=1):
        self.p = p

    def __call__(self, x, y):
        #### YOUR CODE HERE ####
        return (1 + np.dot(x, y)) ** self.p

class GaussianKernel:
    def __init__(self, sigma=5):
        self.sigma = sigma

    def __call__(self, x, y):
        #### YOUR CODE HERE ####
        return np.exp(-np.linalg.norm(x - y) ** 2 / (2 * self.sigma ** 2))

class KernelPerceptron(object):

    def __init__(self, kernel=LinearKernel(), T=1):
        self.kernel = kernel
        self.T = T # number of iterations
        self.alpha = None
        self.Xtra = None
        self.ytra = None

    def fit(self, X, y):
        self.Xtra, self.ytra = X, y
        n_samples, n_features = X.shape
        self.alpha = np.zeros(n_samples, dtype=np.float64)

        # Gram matrix
        #### YOUR CODE HERE ####
        K = np.zeros((n_samples, n_samples))
        for i in range(n_samples):
            for j in range(n_samples):
                K[i, j] = self.kernel(X[i], X[j])

        # Training loop to update alpha
        #### YOUR CODE HERE ####
        for _ in range(self.T):
            for i in range(n_samples):
                if np.sign(np.sum(K[i, :] * self.alpha * y)) != y[i]:
                    self.alpha[i] += 1


    def predict(self, X):
        X = np.atleast_2d(X)
```

```
        n_samples, n_features = X.shape

        #### YOUR CODE HERE ####
        predictions = np.zeros(n_samples)
        for i in range(n_samples):
            predictions[i] = np.sign(np.sum([self.alpha[j] * self.ytra[j] *
self.kernel(self.Xtra[j], X[i]) for j in range(len(self.Xtra))]))
        return predictions
```

1.2.3. *Results.*

We found that the linear kernel struggled. It achieved an accuracy of `25.00%` percent. Out of curiousity, we also tried a couple different values for `T`, and found that `T=3` actually increased our accuracy to `62.50%`. But we can definitely do better with more powerful kernels.

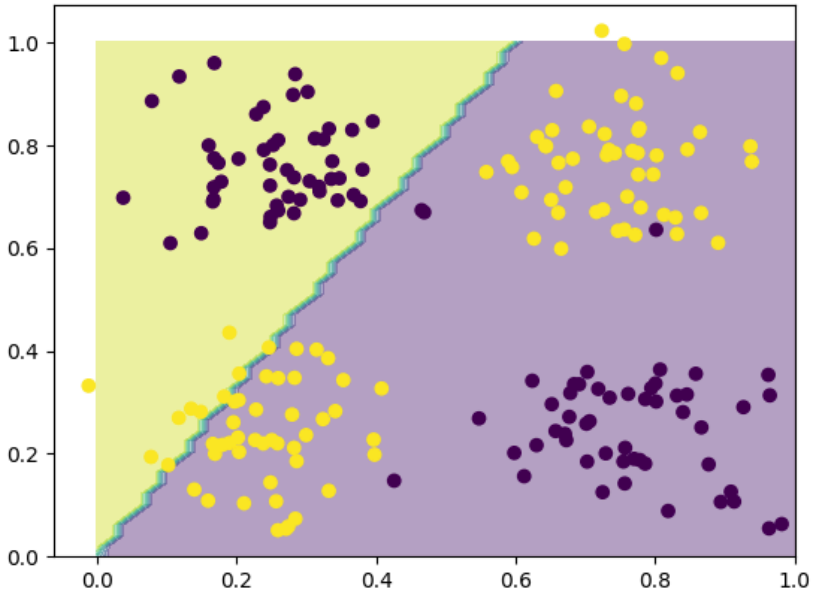

FIGURE 2. Accuracy = 25.00%

With a little bit of tuning, we found that our polynomial kernel can achieve an accuracy of `97.50%` for `p=3, T=2`. (We iterated through a handful of values for `p`, and manually tested a couple different `T`).

```
best_p = 1
best_acc = 0

for p in range(5):
    model = KernelPerceptron(kernel=PolynomialKernel(p=p), T=2)
    model.fit(Xtr, Ytr)
    y_pred = model.predict(Xtest)
    print(f'Accuracy for p={p}: {accuracy_score(Ytest, y_pred) * 100:.2f}
%')
    if accuracy_score(Ytest, y_pred) > best_acc:
```

```
        best_acc = accuracy_score(Ytest, y_pred)
        best_p = p
print('Best p:', best_p)

p = best_p
model = KernelPerceptron(kernel=PolynomialKernel(p=best_p), T=2)
model.fit(Xtr, Ytr)
y_pred = model.predict(Xtest)
print(f"Accuracy for best p={p}: {accuracy_score(Ytest, y_pred) * 100:.2f}
%")
plot_decision_bounday_kernel_perceptron(model)
```

```
Accuracy for p=0: 47.50%
Accuracy for p=1: 47.50%
Accuracy for p=2: 47.50%
Accuracy for p=3: 97.50%
Accuracy for p=4: 95.00%
Best p: 3
```

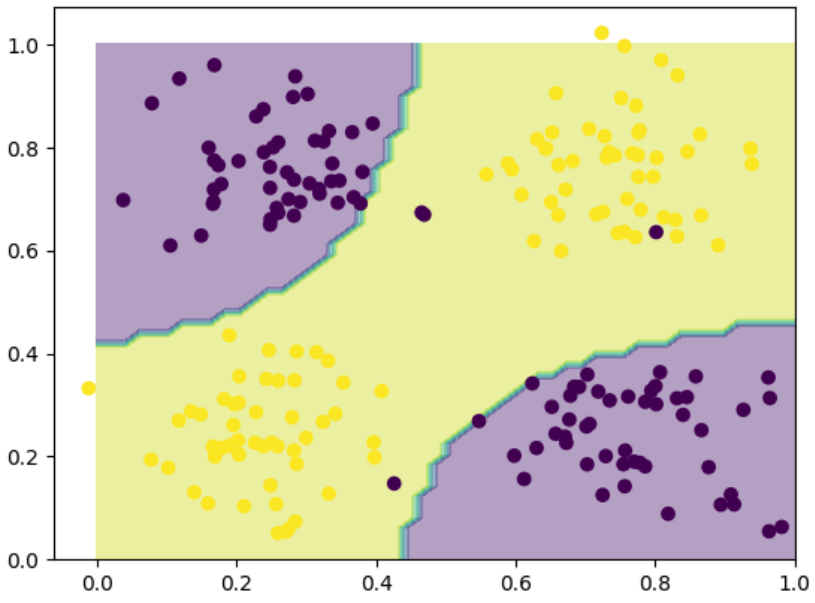

FIGURE 3. Accuracy for best p=3: 97.50%

The gaussian kernel achieved an accuracy of `97.50%` with σ=`0.1, 0.5, 1` and `T=1`. (We iterated through a handful of values for σ, and manually tested a couple different `T`).

```
best_sigma = 1
best_acc = 0

for sigma in [0.1, 0.5, 1, 2, 5, 10]:
    model = KernelPerceptron(kernel=GaussianKernel(sigma=sigma), T=1)
    model.fit(Xtr, Ytr)
    y_pred = model.predict(Xtest)
```

```
    print(f"Accuracy for sigma={sigma}: {accuracy_score(Ytest, y_pred) *
100:.2f}%")
    if accuracy_score(Ytest, y_pred) > best_acc:
        best_acc = accuracy_score(Ytest, y_pred)
        best_sigma = sigma
print('Best sigma:', best_sigma)

sigma = best_sigma
model = KernelPerceptron(kernel=GaussianKernel(sigma=sigma), T=1)
model.fit(Xtr, Ytr)
y_pred = model.predict(Xtest)
print(f"Accuracy for best sigma={sigma}: {accuracy_score(Ytest, y_pred) *
100:.2f}%")
plot_decision_bounday_kernel_perceptron(model)
```

```
Accuracy for sigma=0.1: 97.50%
Accuracy for sigma=0.5: 97.50%
Accuracy for sigma=1: 97.50%
Accuracy for sigma=2: 47.50%
Accuracy for sigma=5: 47.50%
Accuracy for sigma=10: 47.50%
Best sigma: 0.1
```

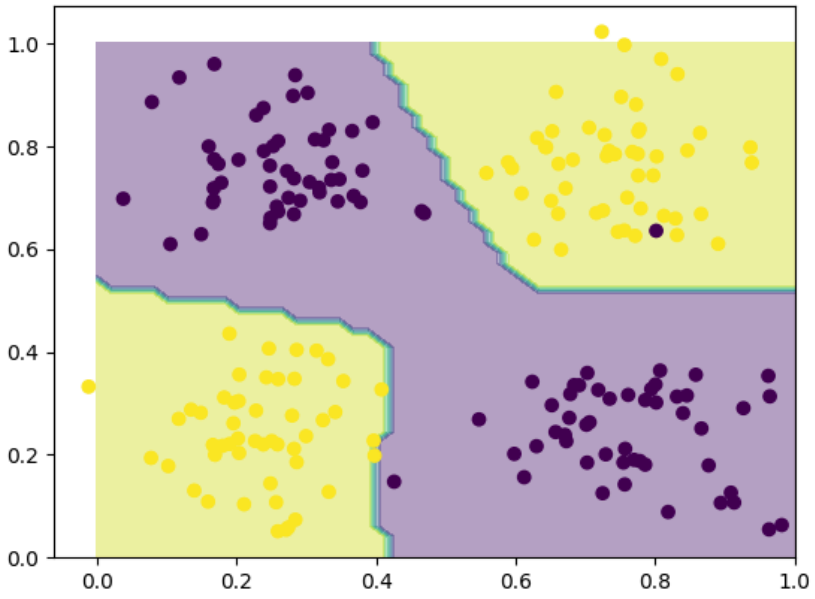

FIGURE 4. Accuracy for best sigma=0.1: 97.50%

2. TASK 2B

2.1. **Real World Data: Seoul Bike Rental Data (30pt).**

2.1.1. *Dev Split.*

We created an additional `dev` split to select the optimal hyperparameters as per the instructions.

```
Xtrain, Xdev, Ytrain, Ydev = train_test_split(Xtr, Ytr, test_size=0.2,
random_state=42)
```

2.1.2. *Polynomial Kernel.*

2.1.2.1. *Overview.*

For the polynomial kernel, we decided to look at $p \in [1, 2, 3, 4, 5]$. The choice of the degree p in the range of 1 to 5 for the polynomial kernel in our kernel perceptron model is justified by the need to balance model complexity and generalization. Lower degrees (1-2) offer simpler, more linear decision boundaries, suitable for less complex datasets, while slightly higher degrees (3-5) allow for capturing non-linear relationships without significant risk of overfitting, a common issue with very high degrees. This range is also aligned with typical default settings in machine learning libraries and empirical observations in various applications, suggesting its general effectiveness across a wide range of problems.

2.1.2.2. *Implementation.*

```
# find best p for polynomial kernel
p_values = [1, 2, 3, 4, 5]
best_p = None
best_accuracy = 0
for p in p_values:
    model = KernelPerceptron(kernel=PolynomialKernel(p=p), T=10)
    model.fit(Xtrain, Ytrain)  # Train on the training set
    y_pred = model.predict(Xdev)  # Predict on the dev set
    accuracy = accuracy_score(Ydev, y_pred)
    print(f"Accuracy for p={p}: {accuracy * 100:.2f}%")
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_p = p

print(f"Best p: {best_p} with accuracy: {best_accuracy * 100:.2f}%")

# testing on the test set
model = KernelPerceptron(kernel=PolynomialKernel(p=best_p), T=10)
model.fit(Xtr, Ytr)
y_pred = model.predict(Xtest)
print('Accuracy on test set: %.2f%%' % (accuracy_score(Ytest, y_pred) * 100))
```

2.1.2.3. *Results.*

```
Accuracy for p=1: 75.00%
Accuracy for p=2: 46.88%
Accuracy for p=3: 76.56%
Accuracy for p=4: 46.88%
Accuracy for p=5: 71.88%
Best p: 3 with accuracy: 76.56%
Accuracy on test set: 71.50%
```

2.1.3. *Gaussian Kernel.*

2.1.3.1. *Overview.*

For the gaussian kernel, smaller values like 0.05 and 0.2 correspond to narrower kernels that focus on capturing local patterns in the data, which is effective for

datasets with distinct clusters but risks overfitting on noise or irregularities. Conversely, larger values like 1.5 and 3 lead to broader, more general kernels that capture wider patterns and are less prone to overfitting, but might overlook important local differences. This range provides a comprehensive assessment from very local to more global influences, allowing for an optimal balance between capturing the complexity of the data and avoiding overfitting or underfitting.

2.1.3.2. *Implementation.*

```
# find best signma for gaussian kernel
sigma_values = [0.05, 0.2, 0.7, 1.5, 3]
best_sigma = None
best_accuracy = 0
for sigma in sigma_values:
    model = KernelPerceptron(kernel=GaussianKernel(sigma=sigma), T=10)
    model.fit(Xtrain, Ytrain)  # Train on the training set
    y_pred = model.predict(Xdev)  # Predict on the dev set
    accuracy = accuracy_score(Ydev, y_pred)
    print(f"Accuracy for sigma={sigma}: {accuracy * 100:.2f}%")
    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_sigma = sigma

print(f"Best sigma: {best_sigma} with accuracy: {best_accuracy * 100:.2f}
%")

# testing on the test set
model = KernelPerceptron(kernel=GaussianKernel(sigma=best_sigma), T=10)
model.fit(Xtr, Ytr)
y_pred = model.predict(Xtest)
print('Accuracy on test set: %.2f%%' % (accuracy_score(Ytest, y_pred) * 100))
```

2.1.3.3. *Results.*

```
Accuracy for sigma=0.05: 67.97%
Accuracy for sigma=0.2: 50.00%
Accuracy for sigma=0.7: 46.88%
Accuracy for sigma=1.5: 46.88%
Accuracy for sigma=3: 46.88%
Best sigma: 0.05 with accuracy: 67.97%
Accuracy on test set: 62.00%
```

Interestingly, the accuracy dropped off significantly.

2.1.4. *New Kernel (Sigmoid).*

Lastly, for our `NewKernel`, we implemented a Sigmoid Kernel.

$$k_{\text{sigmoid}}(x, x') = \tanh(\gamma x^T x' + r)$$

```
class NewKernel(object):
    def __init__(self, gamma=1, r=0):
        self.gamma = gamma
        self.r = r
    def __call__(self, x, y):
        return np.tanh(self.gamma * np.dot(x, y) + self.r)
```

We iterated through $\gamma \in [0.1, 0.5, 1, 2, 5]$ and $r \in [-2, -1, 0, 1, 2]$. The gamma parameter, which scales the dot product of the input vectors, is varied from 0.1 to 5 to assess how different degrees of steepness in the kernel function influence the model's ability to capture complex patterns in the data. Lower gamma values lead to smoother decision boundaries, suitable for more general patterns, while higher gamma values allow for sharper boundaries, potentially capturing finer details but with an increased risk of overfitting. The r parameter, which shifts the kernel function, is explored with both negative and positive values to determine the impact of this shift on the decision boundary. Negative values of r can make the kernel more prone to classify inputs as the negative class, and positive values do the opposite.

2.1.4.1. *Implementation.*

```
# find best parameters for the new (sigmoid) kernel
best_gamma = None
best_r = None
best_accuracy = 0
gamma_values = [0.1, 0.5, 1, 2, 5]
r_values = [-2, -1, 0, 1, 2]
for gamma in gamma_values:
    for r in r_values:
        model = KernelPerceptron(kernel=NewKernel(gamma=gamma, r=r), T=10)
        model.fit(Xtrain, Ytrain)  # Train on the training set
        y_pred = model.predict(Xdev)  # Predict on the dev set
        accuracy = accuracy_score(Ydev, y_pred)
        print(f"Accuracy for gamma={gamma} and r={r}: {accuracy * 100:.2f}
%")
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_gamma = gamma
            best_r = r

print(f"Best gamma: {best_gamma}, best r: {best_r} with accuracy:
{best_accuracy * 100:.2f}%")

# testing on the test set
model = KernelPerceptron(kernel=NewKernel(gamma=best_gamma, r=best_r),
T=10)
model.fit(Xtr, Ytr)
y_pred = model.predict(Xtest)
print('Accuracy on test set: %.2f%%' % (accuracy_score(Ytest, y_pred) * 100))
```

2.1.4.2. *Results.*

```
Accuracy for gamma=0.1 and r=-2: 46.88%
Accuracy for gamma=0.1 and r=-1: 46.88%
Accuracy for gamma=0.1 and r=0: 71.88%
Accuracy for gamma=0.1 and r=1: 46.88%
Accuracy for gamma=0.1 and r=2: 46.88%
Accuracy for gamma=0.5 and r=-2: 46.88%
Accuracy for gamma=0.5 and r=-1: 49.22%
Accuracy for gamma=0.5 and r=0: 58.59%
Accuracy for gamma=0.5 and r=1: 67.97%
Accuracy for gamma=0.5 and r=2: 46.88%
```

```
Accuracy for gamma=1 and r=-2: 48.44%
Accuracy for gamma=1 and r=-1: 60.94%
Accuracy for gamma=1 and r=0: 60.16%
Accuracy for gamma=1 and r=1: 75.00%
Accuracy for gamma=1 and r=2: 68.75%
Accuracy for gamma=2 and r=-2: 61.72%
Accuracy for gamma=2 and r=-1: 65.62%
Accuracy for gamma=2 and r=0: 67.19%
Accuracy for gamma=2 and r=1: 76.56%
Accuracy for gamma=2 and r=2: 75.00%
Accuracy for gamma=5 and r=-2: 65.62%
Accuracy for gamma=5 and r=-1: 67.19%
Accuracy for gamma=5 and r=0: 69.53%
Accuracy for gamma=5 and r=1: 66.41%
Accuracy for gamma=5 and r=2: 75.00%
Best gamma: 2, best r: 1 with accuracy: 76.56%
Accuracy on test set: 82.00%
```