

Experiment 8 : Support Vector Machines (SVMs) and the Kernel Trick

Total Marks: 100

1. Learning Objectives

Upon successful completion of this assignment, students will be able to:

- Articulate the core concepts of Support Vector Machines, including the **maximal margin hyperplane**, **support vectors**, and the **soft margin** (`C`) parameter.
- Understand and implement the **kernel trick** to solve non-linear classification problems.
- Implement and compare the performance of different SVM kernels (**Linear**, **Polynomial**, and **RBF**).
- Understand the role of key hyperparameters like `C`, `gamma`, and `degree`.
- Use `GridSearchCV` to systematically tune hyperparameters for an SVM.
- **Visualize 2D decision boundaries** to intuitively understand how different kernels and hyperparameters work.
- Rigorously evaluate and interpret the performance of a highly-tuned SVM on a hold-out test set.

2. Introduction

Support Vector Machines (SVMs) are a powerful class of supervised machine learning models. The core idea is to find an optimal "hyperplane" that best separates the classes in your data.

However, many real-world datasets are **not linearly separable**. You can't draw a single straight line to separate the classes. This is where the **kernel trick** comes in. A kernel function can project your data into a much higher-dimensional space where a linear separator *can* be found, without the massive computational cost.

In this assignment, you will work with a classic non-linear "moons" dataset. You will see firsthand why a linear SVM fails and how kernelized SVMs (RBF, Polynomial) can easily

solve the problem. You will also tune the key hyperparameters (`C`, `gamma`) that govern the model's behavior.

3. Prerequisites

Ensure your Python environment has the following libraries installed:

```
pip install numpy pandas scikit-learn matplotlib seaborn
```

4. Experiment Tasks

You will use a synthetically generated "moons" dataset, which is a classic example of a non-linearly separable problem.

Task 1: Data Loading and Preprocessing (10 Marks)

1. **Load Data:** Generate the `make_moons` dataset from scikit-learn.

```
from sklearn.datasets import make_moons
# Generate 500 samples with noise to make it challenging
X, y = make_moons(n_samples=500, noise=0.25, random_state=42)
```

2. **Create Hold-Out Set:** Perform a single **70/30 split** on the data.

- `X_train`, `y_train` (70%)
- `X_val`, `y_val` (30%)
- Use `train_test_split` with `random_state=42`.

3. **Standardize Features:** This step is **critical** for SVMs, as they are sensitive to the scale of input features.

- Fit a `StandardScaler` from `sklearn.preprocessing` on `X_train` **only**.
- Transform both `X_train` and `X_val` using the *fitted* scaler.
- You will use `X_train_scaled` and `X_val_scaled` for all model training and evaluation.

Task 2: Model 1 - The (Failing) Linear SVM (15 Marks)

1. **Train Model:**

- Import `from sklearn.svm import SVC` (Support Vector Classifier).
- Instantiate a linear SVM: `linear_model = SVC(kernel='linear', C=1.0, random_state=42)`.

- `fit` the model on `X_train_scaled` and `y_train`.

2. Evaluate:

- Make predictions on `X_val_scaled`.
- Print the `classification_report` and `accuracy_score`.

3. Analyze:

- Briefly explain why the accuracy is not perfect. (Hint: Look at the data you generated).
- Explain what the `C` parameter represents. What would happen if you set `C` to a very small value (e.g., 0.01)?

Task 3: Model 2 & 3 - The Kernel Trick (25 Marks)

You will now explore non-linear kernels to solve the problem.

1. RBF Kernel Model:

- Instantiate a new `SVC` using the **RBF (Radial Basis Function)** kernel: `rbf_model = SVC(kernel='rbf', random_state=42)`. (Use default `C=1.0` and `gamma='scale'`).
- `fit` the model on `X_train_scaled` and `y_train`.
- Evaluate on `X_val_scaled` and print the `classification_report`.

2. Polynomial Kernel Model:

- Instantiate a new `SVC` using the **Polynomial** kernel: `poly_model = SVC(kernel='poly', degree=3, random_state=42)`. (Use the default `degree=3`).
- `fit` the model on `X_train_scaled` and `y_train`.
- Evaluate on `X_val_scaled` and print the `classification_report`.

3. Analyze:

- Create a simple table comparing the **Validation Accuracy** of the `linear`, `rbf`, and `poly` models.
- Which kernel performed best with default settings? Why does this make sense for the 'moons' dataset?

Task 4: Hyperparameter Tuning with GridSearchCV (30 Marks)

The RBF model was likely the best, but its performance depends heavily on the `C` (regularization) and `gamma` (kernel influence) parameters. You will now find the *optimal*

combination.

1. Define Search Space:

- Create a parameter grid to search. A good starting point is:

```
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.1, 1, 10, 100],
    'kernel': ['rbf']
}
```

2. Setup Grid Search:

- Import `from sklearn.model_selection import GridSearchCV`.
- Instantiate GridSearchCV:

```
grid = GridSearchCV(SVC(random_state=42), param_grid, refit=True, verbose=2,
cv=5, scoring='accuracy')
```
- `refit=True` ensures the best model is retrained on all `X_train_scaled` data.
- `cv=5` specifies 5-fold cross-validation.

3. Run Grid Search:

- `fit` the `grid` object on your *entire* training set (`X_train_scaled`, `y_train`).

4. Analyze Results:

- Print the `grid.best_params_` to see the best `C` and `gamma`.
- Print the `grid.best_score_` to see the best cross-validated accuracy.

Task 5: Final Evaluation and Visualization (20 Marks)

1. Evaluate Final Model:

- The `grid` object now holds the best model. Use it to make predictions on your **hold-out set** (`X_val_scaled`).

```
final_predictions = grid.predict(X_val_scaled)
```
- Print the `classification_report(y_val, final_predictions)`.
- Generate and plot a `confusion_matrix` for these final predictions.

2. Visualize Decision Boundaries:

- This is the most important part for understanding *how* the kernels work.
 - You need to plot the decision boundary for the `linear_model`, the default `rbf_model`, and your final `grid` (best) model.
 - *Hint:* Create a 2D mesh grid (`np.meshgrid`), make predictions for *every point* on the grid, and use `plt.contourf` to plot the resulting decision regions. Plot the `X_train_scaled` data points on top.
 - Create a 1x3 subplot to show the three decision boundaries side-by-side.
-

5. Submission Guidelines

Submit a single `.zip` archive containing:

1. **Source Code:** A single Jupyter Notebook (`.ipynb`) containing all your code, outputs, and plots.
2. **PDF Report:** A formal report (`StudentID_Report.pdf`) that includes:
 - **Model Comparison:** A table showing the **Validation Accuracy** for the four models:
 1. `SVC (kernel='linear', C=1)`
 2. `SVC (kernel='rbf', default params)`
 3. `SVC (kernel='poly', degree=3)`
 4. `GridSearchCV Best Model`
 - **Decision Boundary Plots:** The 1x3 subplot from Task 5 showing the decision boundaries for the `linear`, default `rbf`, and final `tuned rbf` models.
 - **Final Performance:** Include the **Classification Report** and **Confusion Matrix** for the final tuned model on the 30% hold-out validation set.
 - **Conclusion:** Briefly answer the following:
 - Why did the linear SVM fail, and why did the RBF kernel succeed? (Refer to your plots).
 - What did the `GridSearchCV` tell you? What were the best `C` and `gamma`?
 - What happens if `gamma` is set *too high* (e.g., 1000)? What happens if `C` is set *too low* (e.g., 0.01)? How would this change the decision boundary?