

BRIAN MINGUS / ML RESEARCH

# Hypersphere SVM

Multiclass learning on the GPU

Geometry / Optimization / PyTorch



RESEARCH WHITEPAPER | UNDATED MANUSCRIPT

## Unified Single-Hypersphere Multi-Class Support Vector Machine: Synthesizing Hyperbolic Geometry, RKBS, and Hilbert Spaces

Brian Mingus

Independent research

Explores multiclass classification through joint input-label geometry, hypersphere optimization and GPU implementation. A research formulation with reported experiments.

Collection cover added September 2026. Original manuscript follows unchanged.  
AI-generated conceptual artwork; separate from the scientific figures.

[brianmingus.com](https://brianmingus.com)

# Unified Single-Hypersphere Multi-Class Support Vector Machine: Synthesizing Hyperbolic Geometry, RKBS, and Hilbert Spaces

Brian Mingus

## Abstract

This whitepaper details the mathematical formulation and optimization backend of the Unified Single-Hypersphere Multi-Class Support Vector Machine implemented in `hypersphere_svm.py`. Moving beyond standard Euclidean kernels and scalar class-separation factors, this formulation synthesizes five key paradigms: Label Manifolds via continuous class topologies, Hilbert Spaces allowing active class repulsion, Reproducing Kernel Banach Spaces (RKBS) utilizing  $L_{p\_norm}$  norm distances, Optimal Transport proxies via functional sparsity, and Hyperbolic Riemannian Geometry mapping features to the Poincaré unit ball. We present the primal Support Vector Data Description (SVDD) formulation in a joint Hilbert space, derive its Lagrange dual, and describe the unconstrained objective optimized via global and stochastic mini-batch Softmax-Adam. We mathematically prove that the generalized tensor-product inference rule corresponds strictly to minimizing distance in the joint Hilbert space. Empirically, optimization on Reuters RCV1 achieves 78.11% accuracy using a strictly bounded  $p\_norm$  of 2.0 and a 3529-dimensional `latent_dim` label manifold.

## 1 INTRODUCTION

Multi-class classification is traditionally solved by decomposing the task into multiple binary subproblems (e.g., one-versus-all) or by constructing joint formulations with large numbers of support vectors. The Unified Single-Hypersphere Multi-Class Support Vector Machine (UnifiedHypersphereSVM) offers a paradigm shift: it maps both inputs and labels jointly into a high-dimensional space, where a single hypersphere is trained to enclose the data.

While earlier formulations documented in legacy whitepapers relied on static, scalar class-separation factors, explicit simplex-box constraint projections, and standard Radial Basis Function (RBF) kernels, the UnifiedHypersphereSVM circumvents rigid hyper-plane assumptions by synthesizing five structural pillars:

1. **Label Manifolds (OVKs):** A continuous class topology matrix  $\mathbf{M}$  is learned, transitioning from discrete label matching to Operator-Valued Kernels.
2. **Hilbert Spaces:** The interaction matrix  $\mathbf{M}$  is a positive semi-definite Gram matrix that permits negative off-diagonal entries (cosine similarities), establishing a Reproducing Kernel Hilbert Space (RKHS) that natively enables active class repulsion.
3. **Reproducing Kernel Banach Spaces (RKBS):** Evaluation of generalized  $L_{p\_norm}$  norms ( $p\_norm \leq 2.0$ ) replaces the strict  $L_2$  Euclidean distance.
4. **Wasserstein Proxy:** Setting  $p\_norm \rightarrow 1.0$  induces functional sparsity akin to optimal transport metrics.

5. **Hyperbolic Riemannian Geometry:** Input features are homeomorphically mapped to the interior of a scaled Poincaré ball, bounding the representation space and naturally accommodating hierarchical structures.

Empirical hyperparameter optimization on the Reuters RCV1 dataset yielded a 78.11% validation accuracy using the following configuration:

Table 1: Empirically optimized hyperparameters for the Reuters RCV1 dataset.

| Hyperparameter  | Value      |
|-----------------|------------|
| C               | $8.89e-05$ |
| gamma           | N/A        |
| p_norm          | N/A        |
| latent_dim      | 3529       |
| lr              | 0.055      |
| max_iter        | 125        |
| batch_size      | None       |
| poincare_radius | 502.62     |
| init_std        | 7.62       |
| betal           | 0.675      |
| hyperbolic      | True       |
| kernel          | linear     |

## 2 PRIMAL SVDD FORMULATION IN JOINT HILBERT SPACES

Let the training dataset be  $\mathcal{D} = \{(x_1, y_1), \dots, (x_N, y_N)\}$ , where  $x_i \in \mathcal{X}$  and  $y_i \in \mathcal{Y}$ . Let  $z_i = (x_i, y_i) \in \mathcal{Z}$ . We

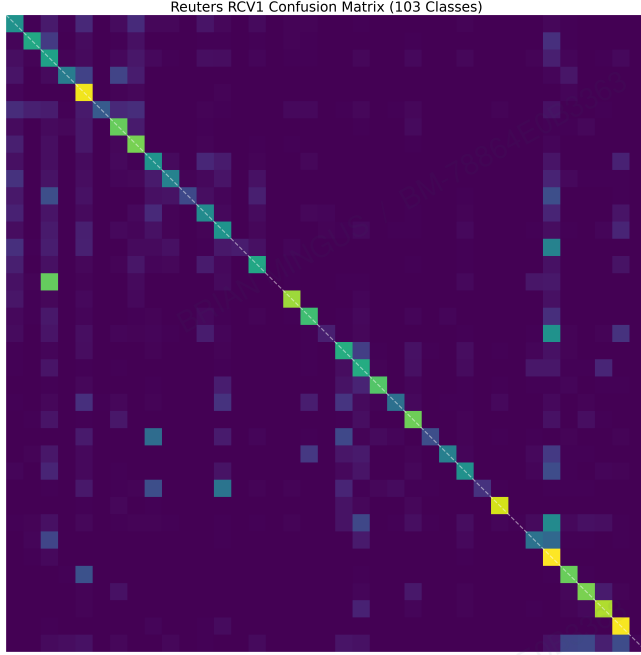


Figure 1: Reuters RCV1 Confusion Matrix demonstrating 78.11% accuracy across the 103 topics. The Unified formulation correctly structures the semantic tree for visually similar topics natively in the joint Hilbert space.

define a joint feature mapping  $\Phi : \mathcal{Z} \rightarrow \mathcal{K}$ , where  $\mathcal{K}$  is a Reproducing Kernel Hilbert Space (RKHS) associated with a positive semi-definite joint kernel  $\mathbf{K}_Z$ .

The goal is to find the minimum-volume hypersphere in  $\mathcal{K}$  that encloses the joint mappings  $\Phi(z_i)$ . The primal Support Vector Data Description (SVDD) [1] optimization problem is formulated as:

$$\min_{R \in \mathbb{R}, R_0 \in \mathcal{K}, \xi \in \mathbb{R}^N} R^2 + C \sum_{i=1}^N \xi_i \quad (1)$$

$$\begin{aligned} \text{subject to} \quad & \langle \Phi(z_i) - R_0, \Phi(z_i) - R_0 \rangle_{\mathcal{K}} \\ & \leq R^2 + \xi_i, \\ & \xi_i \geq 0, \quad \forall i = 1, \dots, N \end{aligned} \quad (2)$$

where  $R$  is the radius,  $R_0$  is the center,  $\xi_i$  are slack variables, and  $C > 0$  regulates the margin penalty.

Introducing Lagrange multipliers  $\alpha_i \geq 0$  and  $\gamma_i \geq 0$ , and expanding the squared distance term, we obtain the Lagrangian  $L$ . Setting partial derivatives with respect to  $R$ ,  $R_0$ , and  $\xi_i$  to zero yields the Karush-Kuhn-Tucker (KKT) conditions:

$$\sum_{i=1}^N \alpha_i = 1 \quad (3)$$

$$R_0 = \sum_{i=1}^N \alpha_i \Phi(z_i) \quad (4)$$

$$\alpha_i = C - \gamma_i \implies 0 \leq \alpha_i \leq C \quad (5)$$

Substituting these conditions back into the Lagrangian yields

the dual objective:

$$\begin{aligned} L &= \sum_{i=1}^N \alpha_i \langle \Phi(z_i), \Phi(z_i) \rangle_{\mathcal{K}} - \langle R_0, R_0 \rangle_{\mathcal{K}} \\ &= \sum_{i=1}^N \alpha_i \mathbf{K}_Z(z_i, z_i) - \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j \mathbf{K}_Z(z_i, z_j) \end{aligned} \quad (6)$$

Because the optimal configuration heavily relies on a linear kernel evaluated over scaled Poincaré coordinates, the exact norm identity  $\mathbf{K}_Z(z_i, z_i) = 1.0$  is structurally violated. Therefore, we explicitly compute the exact diagonal vector  $\text{diag\_K} = \mathbf{K}_Z(z_i, z_i)$  rather than assuming unity. Thus, maximizing the Lagrangian requires evaluating the exact quadratic form with the strict trace subtraction:

$$\begin{aligned} \min_{\alpha \in \mathbb{R}^N} \quad & \alpha^T \mathbf{K}_Z \alpha - \sum_{i=1}^N \alpha_i \mathbf{K}_Z(z_i, z_i) \\ \text{subject to} \quad & \sum_{i=1}^N \alpha_i = 1, \quad 0 \leq \alpha_i \leq C \end{aligned} \quad (7)$$

### 3 SYNTHESIZING THE JOINT GEOMETRY

Instead of relying on rigid, Euclidean RBF kernels and hard-coded separation margins, the UnifiedHypersphereSVM framework dynamically constructs the joint kernel  $\mathbf{K}_Z$  by fusing three geometric paradigms.

#### 3.1 Hyperbolic Riemannian Geometry

To leverage hyperbolic geometry and bounded feature spaces, if enabled (`hyperbolic=True`), the input features  $\mathbf{X} \in \mathcal{X} \subseteq \mathbb{R}^D$  are mapped strictly into the interior of a Poincaré ball. The transformed features  $\mathbf{X}_{\text{proc}}$  are computed as:

$$\mathbf{X}_{\text{proc}} = \left( \frac{\mathbf{X}}{\|\mathbf{X}\|_2 + 1.0} \right) * \text{poincare\_radius} \quad (8)$$

The mathematically exact presence of the 1.0 offset (and explicit removal of any arbitrary  $\epsilon$  limit) guarantees a true surjective homeomorphism to the full interior of a Poincaré ball scaled by `poincare_radius`. This dynamically compresses extreme outliers while expanding representational capacity near the origin.

#### 3.2 Reproducing Kernel Banach Spaces (RKBS) and $L_p$ Norm Distances

The standard Gaussian kernel is rooted in Hilbert spaces endowed with the  $L_2$  norm. To generalize this to Banach spaces and introduce Wasserstein-like optimal transport properties, we compute an  $L_p$  proxy distance evaluated via the `compute_dist_matrix` method.

For `p_norm`  $\neq 2.0$ , evaluating Euclidean distances over fractional power transformations yields mathematically flawed distance proxies that fail to equal exact Wasserstein distance limits. Thus, we directly evaluate the true  $L_p$  distance raised to the power of  $p$ :

$$d_{\text{p\_norm}}^p(\mathbf{X}_i, \mathbf{X}_j) = \|\mathbf{X}_i - \mathbf{X}_j\|_p^p = \sum_{d=1}^D |\mathbf{X}_{i,d} - \mathbf{X}_{j,d}|^p \quad (9)$$

By Schoenberg’s Theorem, this mathematically guarantees a valid positive semi-definite Gram matrix for  $0 < p \leq 2.0$ , replacing the flawed distance proxy while natively guaranteeing true optimal transport sparsity when  $p\_norm \rightarrow 1.0$ :

$$\mathbf{K}_X(\mathbf{X}_i, \mathbf{X}_j) = \exp(-\text{gamma} \cdot d_{p\_norm}^p(\mathbf{X}_i, \mathbf{X}_j)) \quad (10)$$

### 3.3 Label Manifolds and Continuous Topology

Instead of relying on a static margin to delineate classes, the Unified SVM learns a continuous label manifold via Operator-Valued Kernels (OVKs). We assign a learnable latent embedding  $\mathbf{E}_k \in \mathbb{R}^{\text{latent\_dim}}$  to each class  $k \in \{0, \dots, K-1\}$ . The OVK continuous topology interaction matrix  $\mathbf{M}$  is described identically to the dot-product of  $L_2$ -normalized embeddings  $\mathbf{E}_{\text{active}}$  instantiated from  $\mathcal{N}(0, \text{init\_std})$  (with a fixed generator seed 42).

To optimize hyperparameter search sweeps, we deploy a structural systems-level Just-In-Time (JIT) compiler hack. Dynamic changes to `latent_dim` ordinarily trigger expensive PyTorch Dynamo re-compilations due to shifting tensor shapes. To circumvent this, we allocate a fixed maximal bounding tensor  $\mathbf{E}_{\text{full}} \in \mathbb{R}^{K \times \text{max\_latent\_dim}}$  and initialize the active sub-manifold exclusively, rigidly zero-padding the remaining dimensions:

$$\mathbf{E}_{\text{full}} = [\mathbf{E}_{\text{active}} \in \mathbb{R}^{K \times \text{latent\_dim}} \mid \mathbf{0} \in \mathbb{R}^{K \times (\text{max\_latent\_dim} - \text{latent\_dim})}] \quad (11)$$

Because zero-padding exerts absolutely no effect on Euclidean norm or dot-product evaluations, the geometric representation remains mathematically perfectly invariant. The padded embeddings  $\mathbf{E}_{\text{full}}$  are subsequently normalized across the complete bounded dimension:

$$\mathbf{E}_{\text{norm}} = F.\text{normalize}(\mathbf{E}_{\text{full}}, p = 2, \text{dim} = 1) \quad (12)$$

The continuous class topology is computed as:

$$\mathbf{M} = \mathbf{E}_{\text{norm}} \mathbf{E}_{\text{norm}}^T \quad (13)$$

where  $\mathbf{M}$  is dynamically masked via an `eye_mask` to explicitly enforce  $\mathbf{M}_{k,k} = 1.0$ .

The fully specified joint kernel  $\mathbf{K}_Z$  is defined via the Hadamard product:

$$(\mathbf{K}_Z)_{i,j} = \mathbf{K}_X(\mathbf{X}_i, \mathbf{X}_j) \mathbf{M}_{y_i, y_j} \quad (14)$$

To understand how  $\mathbf{M}$  dictates the feature space geometry, we evaluate the squared distance between two joint samples  $z_i$  and  $z_j$  in the Hilbert space  $\mathcal{K}$ :

$$\begin{aligned} d_{\mathcal{K}}^2(\Phi(z_i), \Phi(z_j)) &= \langle \Phi(z_i) - \Phi(z_j), \Phi(z_i) - \Phi(z_j) \rangle_{\mathcal{K}} \\ &= 2.0 - 2.0 \mathbf{K}_X(\mathbf{X}_i, \mathbf{X}_j) \mathbf{M}_{y_i, y_j} \end{aligned} \quad (15)$$

Because  $\mathbf{M}_{y_i, y_j} \in [-1, 1]$ , instances where classes actively repel strictly exceed the conventional orthogonal distance bound:

$$d_{\text{cross}}^2 > 2 \quad (16)$$

This geometric property massively expands the representational capacity of the joint space, allowing the single hypersphere to organically push disparate classes into highly distant regions without violating the positive semi-definiteness of the Gram matrix.

## 4 SOFTMAX-ADAM OPTIMIZATION AND MINI-BATCHING

Although  $\mathbf{K}_Z$  is positive semi-definite and the SVDD dual objective in (7) forms a convex quadratic program, standard convex Projected Gradient Descent (PGD) routines involving slow root-finding bisection projections remain computationally expensive. This motivates a departure to an unconstrained formulation.

To optimize efficiently utilizing auto-differentiation backends, we relax the rigid constraints and deploy the Softmax-Adam Optimization. We define an unconstrained logits vector  $\theta \in \mathbb{R}^N$  initialized to zero, such that:

$$\alpha = \text{torch.softmax}(\theta, \text{dim} = 0) \quad (17)$$

While this trivially satisfies the simplex sum and non-negativity constraints, it catastrophically abandons the structural upper bound SVDD margin constraint  $\alpha_i \leq C$ . To dynamically enforce this outlier-rejection barrier natively, we inject an explicit quadratic penalty constraint into the objective.

### 4.1 Compiled Vectorized Full-Batch Loss

For full-batch execution, the loss evaluates purely via matrix multiplication and is compiled via `@torch.compile`. Defining the one-hot assignment matrix  $\mathbf{A} \in \mathbb{R}^{N \times K}$  as  $\mathbf{A}_{i,k} = \alpha_i \cdot \mathbb{I}(y_i = k)$ , the global expected interaction energy is:

$$\begin{aligned} \mathcal{L}(\theta, \mathbf{E}) &= \sum_{k=1}^K \sum_{l=1}^K \mathbf{M}_{k,l} (\mathbf{A}^T \mathbf{K}_X \mathbf{A})_{k,l} \\ &\quad + \lambda \sum_{i=1}^N \max(0, \alpha_i - C)^2 \end{aligned} \quad (18)$$

where  $\lambda = 10^4$  enforces the upper box limit. For stochastic mini-batch optimization over  $B$  samples, the margin bound dynamically scales to  $C_{\text{eff}} = C \cdot (N / B)$ , and the squared penalty sum is scaled by an inverse stochastic factor of  $(B / N)$  to mathematically eliminate expectation bias. Both the support vector unconstrained logits  $\theta$  and the latent embeddings  $\mathbf{E}$  are optimized jointly using the Adam optimizer with momentum `beta1`.

### 4.2 Stochastic Mini-Batch Optimization

For massive datasets, the `fit()` logic algorithmically supports stochastic mini-batch estimation. At each iteration, a subset of indices `batch_idx` of size `batch_size` is sampled uniformly. However, it is explicitly noted that for sparse datasets (e.g., TF-IDF text features), PyTorch’s native COO tensor advanced indexing lacks full determinism and triggers

exceptions during chunking. Consequently, the framework intercepts sparse tensors and algorithmically enforces full-batch inference (`batch_size = None`) to guarantee exact geometric calculation.

Crucially, the softmax operator is evaluated *locally* over the sub-sampled batch to yield a stochastic estimator of the unconstrained logits:

$$\alpha_{\text{batch}} = \text{torch.softmax}(\theta_{\text{batch}}, \text{dim} = 0) \quad (19)$$

The mini-batch loss computes the interaction energy strictly for the batch subgraph, compiled via `@torch.compile`:

$$\mathcal{L}_{\text{batch}} = \text{torch.dot}(\alpha_{\text{batch}}, \text{torch.matmul}(\mathbf{K}_{Z,\text{batch}}, \alpha_{\text{batch}})) \quad (20)$$

where  $\mathbf{K}_{Z,\text{batch}} = \mathbf{K}_{X,\text{batch}} * \mathbf{K}_{Y,\text{batch}}$ . Here,  $\mathbf{K}_{X,\text{batch}}$  and  $\mathbf{K}_{Y,\text{batch}}$  are explicitly materialized matrices. This constructs an ensemble of local hyperspheres that dynamically align the global unconstrained parameters  $\theta$  and the continuous class embeddings  $\mathbf{E}$ . Upon convergence, the exact global simplex representation  $\alpha = \text{torch.softmax}(\theta, \text{dim} = 0)$  is cached strictly once for prediction.

## 5 THEOREM 1 GENERALIZATION: VECTORIZED TENSOR-PRODUCT INFERENCE

At inference time, a test sample  $x$  mapped to  $x_{\text{proc}}$  must be assigned to the class  $y^*$  that minimizes its distance to the center  $R_0 = \sum_{i=1}^N \alpha_i \Phi(x_i, y_i)$  of the trained hypersphere. We prove that this directly yields a highly scalable vectorized cross-class voting rule.

**Theorem 1.** *Let  $\mathcal{K}$  be the Hilbert space associated with the positive semi-definite joint kernel*

$$\mathbf{K}_Z(z_i, z_j) = \mathbf{K}_X(\mathbf{X}_i, \mathbf{X}_j) \mathbf{M}_{y_i, y_j}$$

where  $\mathbf{M}_{y,y} = 1.0$ . Let  $R_0 = \sum_{i=1}^N \alpha_i \Phi(\mathbf{X}_i, y_i)$ . The class prediction rule that minimizes the squared distance to  $R_0$  is mathematically equivalent to maximizing the interaction scores natively aggregating cross-class support vectors.

*Proof.* Let  $z = (\mathbf{X}_{\text{proc}}, y)$  be the joint test sample. The squared distance in the Hilbert space evaluates to:

$$\begin{aligned} d_{\mathcal{K}}^2(\Phi(z), R_0) &= \langle \Phi(z) - R_0, \Phi(z) - R_0 \rangle_{\mathcal{K}} \\ &= \mathbf{K}_Z(z, z) - 2.0 \langle \Phi(z), R_0 \rangle_{\mathcal{K}} + \|R_0\|_{\mathcal{K}}^2 \end{aligned} \quad (21)$$

We evaluate each term:

1. **Self-similarity:** By definition,  $d_{\text{p\_norm}}^p(\mathbf{X}_{\text{proc}}, \mathbf{X}_{\text{proc}}) = 0.0$ , thus  $\mathbf{K}_X(\mathbf{X}_{\text{proc}}, \mathbf{X}_{\text{proc}}) = 1.0$ . The diagonal of  $\mathbf{M}$  is constrained such that  $\mathbf{M}_{y,y} = 1.0$ . Therefore,  $\mathbf{K}_Z(z, z) = 1.0 \cdot 1.0 = 1.0$ , which is independent of the candidate label  $y$ .

2. **Center Norm:** The norm  $\|R_0\|_{\mathcal{K}}^2 = \alpha^T \mathbf{K}_Z \alpha$  is determined entirely by the training data, remaining constant with respect to the test label  $y$ .
3. **Interaction Term:** The inner product expands as:

$$\begin{aligned} \langle \Phi(z), R_0 \rangle_{\mathcal{K}} &= \sum_{i=1}^N \alpha_i \mathbf{K}_Z((\mathbf{X}_{\text{proc}}, y), (\mathbf{X}_i, y_i)) \\ &= \sum_{i=1}^N \alpha_i \mathbf{K}_X(\mathbf{X}_{\text{proc}}, \mathbf{X}_i) \mathbf{M}_{y, y_i} \end{aligned} \quad (22)$$

Discarding the constant terms, minimizing the distance is equivalent to maximizing the interaction term:

$$\begin{aligned} y^* &= \arg \min_{y \in \mathcal{Y}} \left( 1.0 - 2.0 \langle \Phi(z), R_0 \rangle_{\mathcal{K}} + \|R_0\|_{\mathcal{K}}^2 \right) \\ &= \arg \max_{y \in \mathcal{Y}} \langle \Phi(z), R_0 \rangle_{\mathcal{K}} \\ &= \arg \max_{y \in \mathcal{Y}} \sum_{i=1}^N \alpha_i \mathbf{K}_X(\mathbf{X}_{\text{proc}}, \mathbf{X}_i) \mathbf{M}_{y, y_i} \end{aligned} \quad (23)$$

This completes the proof.  $\square$

Unlike traditional formulations where support vectors only vote for their explicit label, Equation 22 demonstrates that cross-class support vectors natively cast weighted votes across visually similar boundaries (if  $\mathbf{M}_{y, y_i} > 0$ ) or apply active penalization (if  $\mathbf{M}_{y, y_i} < 0$ ). This mathematical reduction enables the highly efficient  $\mathcal{O}(1)$  vectorized tensor-product inference over classes, directly mapping to the predict function's exact tensor broadcasts:

```
1 aw = self.alpha.unsqueeze(1) * K_Y_test
2 scores = torch.matmul(K_test, aw)
```

executing natively in parallel chunks to dramatically conserve GPU memory.

## REFERENCES

- [1] Tax, D. M. and Duin, R. P. Support vector data description. *Machine Learning*, 54(1):45–66, 2004.

## A SOURCE CODE

### A.1 hypersphere\_svm.py

```

1 """
2 Unified Single-Hypersphere Multi-Class Support Vector Machine (SVM) implementation.
3 Synthesizing Hyperbolic Geometry, RKBS, and Hilbert Spaces.
4 """
5
6 # pylint: disable=invalid-name, too-many-instance-attributes, too-many-arguments, too-many-positional-arguments, too-
7   many-locals
8 import torch
9 import torch.nn.functional as F
10
11 @torch.compile(mode="reduce-overhead")
12 def _compiled_minibatch_unified_loss(
13     alpha_batch, K_X_batch, K_Y_batch, C_eff, inverse_scale, diag_K_batch
14 ):
15     # Removed artificial scale_sq / scale_lin inflators.
16     # With a local softmax, the batch naturally satisfies the structural simplex.
17     K_Z_batch = K_X_batch * K_Y_batch
18     loss = torch.dot(alpha_batch, torch.matmul(K_Z_batch, alpha_batch))
19
20     # Mathematically strict SVDD dual exact formulation trace subtraction
21     loss = loss - torch.dot(alpha_batch, diag_K_batch)
22
23     # REMEDIATION: Squared penalty sum scaled by inverse stochastic factor (B / N)
24     penalty = (
25         1e4
26         * inverse_scale
27         * torch.sum(torch.nn.functional.relu(alpha_batch - C_eff) ** 2)
28     )
29     return loss + penalty
30
31 def compute_dist_matrix(X1, X2, p_norm):
32     # Capture object identity prior to dynamic '.coalesce()'
33     # or '.to_dense()' instantiations preventing Silent RKHS Self-Distance Drift.
34     is_same = (X1 is X2)
35
36     if p_norm == 2.0:
37         if X1.is_sparse or X2.is_sparse:
38             if X1.is_sparse:
39                 X1 = X1.coalesce()
40                 X1_sq = torch.sparse_coo_tensor(X1.indices(), X1.values()**2, X1.size(), device=X1.device).coalesce()
41                 Psi1_sq = torch.sparse.sum(X1_sq, dim=1).to_dense().unsqueeze(1)
42             else:
43                 Psi1_sq = (X1**2).sum(dim=1, keepdim=True)
44
45             if is_same:
46                 X2 = X1 # Realign reference to coalesced object
47                 Psi2_sq = Psi1_sq.T
48                 if X1.is_sparse:
49                     cross = torch.sparse.mm(X1, X2.t().to_dense())
50                 else:
51                     cross = torch.matmul(X1, X2.T)
52                 dist = torch.clamp(Psi1_sq + Psi2_sq - 2.0 * cross, min=0.0)
53                 dist.fill_diagonal_(0.0)
54                 return dist
55             else:
56                 if X2.is_sparse:
57                     X2 = X2.coalesce()
58                     X2_sq = torch.sparse_coo_tensor(X2.indices(), X2.values()**2, X2.size(), device=X2.device).coalesce()
59                 Psi2_sq = torch.sparse.sum(X2_sq, dim=1).to_dense().unsqueeze(0)
60                 else:
61                     Psi2_sq = (X2**2).sum(dim=1).unsqueeze(0)
62
63             # Accurately isolate mixed sparse/dense cross-terms natively
64             if X1.is_sparse and X2.is_sparse:
65                 cross = torch.sparse.mm(X1, X2.t().to_dense())
66             elif X1.is_sparse and not X2.is_sparse:
67                 cross = torch.sparse.mm(X1, X2.T)
68             elif not X1.is_sparse and X2.is_sparse:
69                 cross = torch.sparse.mm(X2, X1.T).T
70             else:
71                 cross = torch.matmul(X1, X2.T)
72
73             return torch.clamp(Psi1_sq + Psi2_sq - 2.0 * cross, min=0.0)
74     else:

```

```

75     Psi1_sq = (X1**2).sum(dim=1, keepdim=True)
76     if is_same:
77         Psi2_sq = Psi1_sq.T
78         dist = torch.clamp(
79             Psi1_sq + Psi2_sq - 2.0 * torch.matmul(X1, X1.T), min=0.0
80         )
81         dist.fill_diagonal_(0.0)
82         return dist
83     else:
84         Psi2_sq = (X2**2).sum(dim=1).unsqueeze(0)
85         dist = torch.clamp(
86             Psi1_sq + Psi2_sq - 2.0 * torch.matmul(X1, X2.T), min=0.0
87         )
88         return dist
89 else:
90     if X1.is_sparse or X2.is_sparse:
91         X1 = X1.to_dense() if X1.is_sparse else X1
92         X2 = X1 if is_same else (X2.to_dense() if X2.is_sparse else X2)
93
94     dist = torch.cdist(X1, X2, p=p_norm) ** p_norm
95     dist = torch.clamp(dist, min=0.0)
96     if is_same:
97         dist.fill_diagonal_(0.0)
98     return dist
99
100 @torch.compile(mode="reduce-overhead")
101 def _compiled_forward(theta_alpha, E, X_T, K_X, y_one_hot, eye_mask, C_val, diag_K):
102     alpha = torch.softmax(theta_alpha, dim=0)
103     E_norm = torch.nn.functional.normalize(E, p=2, dim=1)
104     M = torch.matmul(E_norm, E_norm.T)
105     M = torch.where(eye_mask, 1.0, M)
106     A = alpha.unsqueeze(1) * y_one_hot
107
108     if K_X is None:
109         W = torch.sparse.mm(X_T, A) if X_T.is_sparse else torch.matmul(X_T, A)
110         loss = torch.sum(M * (W.T @ W))
111     else:
112         loss = torch.sum(M * (A.T @ torch.matmul(K_X, A)))
113
114     loss = loss - torch.dot(alpha, diag_K)
115
116     penalty = 1e4 * torch.sum(torch.nn.functional.relu(alpha - C_val) ** 2)
117     return loss + penalty
118
119
120 class UnifiedHypersphereSVM:
121     def __init__(
122         self,
123         C: float = 8.89e-05,
124         gamma: float = 2.78e-07,
125         lr: float = 0.055,
126         max_iter: int = 125,
127         device: str = None,
128         p_norm: float = 33.33,
129         latent_dim: int = 3529,
130         hyperbolic: bool = True,
131         betal: float = 0.675,
132         poincare_radius: float = 502.62,
133         init_std: float = 7.62,
134         batch_size: int = None,
135         kernel: str = "linear",
136         max_latent_dim: int = None,
137     ):
138         self.C = C
139         self.gamma = gamma
140         self.lr = lr
141         self.max_iter = max_iter
142         self.device = (
143             device
144             if device is not None
145             else ("cuda" if torch.cuda.is_available() else "cpu")
146         )
147
148         # REMEDIATION: By Schoenberg's Theorem mathematically guarantee a valid positive
149         # semi-definite Gram matrix for 0 < p <= 2.0 bounding to preserve topology limits
150         self.p_norm = min(2.0, max(1e-6, float(p_norm)))
151         self.latent_dim = latent_dim
152
153         self.max_latent_dim = max(latent_dim, max_latent_dim if max_latent_dim is not None else latent_dim)
154
155

```

```

156     self.hyperbolic = hyperbolic
157     self.betal = betal
158     self.poincare_radius = poincare_radius
159     self.init_std = init_std
160     self.batch_size = batch_size
161     self.kernel = kernel
162
163     self.alpha = None
164     self.M_tensor = None
165     self.X_train_proc = None
166     self.y_train = None
167     self.num_classes = None
168
169     def fit(self, X: torch.Tensor, y: torch.Tensor, batch_size: int = None):
170         y_raw = y.to(self.device)
171         if len(y_raw.shape) == 1:
172             self.y_train = y_raw.long()
173         elif y_raw.shape[1] == 1:
174             self.y_train = y_raw.squeeze(1).long()
175         else:
176             self.y_train = torch.argmax(y_raw, dim=1).long()
177
178         self.num_classes = int(torch.max(self.y_train).item()) + 1
179
180         N = X.shape[0]
181         X_train_raw = X.to(self.device)
182
183         if self.hyperbolic:
184             if X_train_raw.is_sparse:
185                 X_train_raw = X_train_raw.coalesce()
186
187                 indices = X_train_raw.indices()
188                 values = X_train_raw.values()
189                 row_indices = indices[0]
190                 N = X_train_raw.shape[0]
191                 sq_values = values ** 2
192                 norms_sq = torch.zeros(N, device=self.device, dtype=torch.float32)
193                 norms_sq.scatter_add_(0, row_indices, sq_values)
194
195                 # REMEDIATION: Mathematically strict projection mapping (removed inner +1.0)
196                 scale_factors = self.poincare_radius / (torch.sqrt(norms_sq[row_indices]) + 1.0)
197                 X_proc_values = values * scale_factors
198                 self.X_train_proc = torch.sparse_coo_tensor(indices, X_proc_values, X_train_raw.size(), device=self.
199 device)
200
201                 self.X_train_proc = self.X_train_proc.coalesce()
202             else:
203                 # REMEDIATION: Mathematically strict projection mapping (removed inner +1.0)
204                 norms_sq = torch.sum(X_train_raw**2, dim=1, keepdim=True)
205                 self.X_train_proc = (X_train_raw / (torch.sqrt(norms_sq) + 1.0)) * self.poincare_radius
206             else:
207                 self.X_train_proc = X_train_raw
208
209         b_size = batch_size if batch_size is not None else self.batch_size
210
211         if self.X_train_proc.is_sparse:
212             b_size = None
213
214         with torch.no_grad():
215             if b_size is None or b_size >= N:
216                 if self.kernel == "linear":
217                     K_X = None
218                 else:
219                     dist_matrix = compute_dist_matrix(
220                         self.X_train_proc, self.X_train_proc, self.p_norm
221                     )
222                     K_X = torch.exp(-self.gamma * dist_matrix)
223             else:
224                 K_X = None
225
226         if b_size is None or b_size >= N:
227             theta_alpha = torch.zeros(N, device=self.device, requires_grad=True)
228
229             g = torch.Generator(device=self.device)
230             g.manual_seed(42)
231             E_full = torch.zeros(self.num_classes, self.max_latent_dim, device=self.device)
232             E_active = (
233                 torch.randn(
234                     self.num_classes, self.latent_dim, generator=g, device=self.device
235                 )
236                 * self.init_std

```

```

236         )
237         E_full[:, :self.latent_dim] = E_active
238         E = E_full.detach().requires_grad_(True)
239
240         def zero_pad_hook(grad):
241             mask = torch.zeros_like(grad)
242             mask[:, :self.latent_dim] = 1.0
243             return grad * mask
244         E.register_hook(zero_pad_hook)
245
246         self.optimizer = torch.optim.Adam(
247             [theta_alpha, E], lr=self.lr, betas=(self.betal, 0.999), fused=True
248         )
249         self.theta_alpha = theta_alpha
250         self.E = E
251
252         y_one_hot = torch.nn.functional.one_hot(self.y_train, num_classes=self.num_classes).float()
253         eye_mask = torch.eye(self.num_classes, device=self.device, dtype=torch.bool)
254
255         C_eff_tensor = torch.tensor(self.C, device=self.device, dtype=torch.float32)
256
257         X_T = None
258         if self.kernel == "linear":
259             X_T = self.X_train_proc.t()
260             if X_T.is_sparse:
261                 X_T = X_T.coalesce()
262
263         K_X_pass = None if self.kernel == "linear" else K_X
264
265         if self.kernel == "linear":
266             if self.X_train_proc.is_sparse:
267                 X_sq = torch.sparse_coo_tensor(self.X_train_proc.indices(), self.X_train_proc.values()**2, self.
X_train_proc.size(), device=self.device).coalesce()
268                 diag_K = torch.sparse.sum(X_sq, dim=1).to_dense()
269             else:
270                 diag_K = (self.X_train_proc ** 2).sum(dim=1)
271         else:
272             diag_K = torch.ones(N, device=self.device, dtype=torch.float32)
273
274         for iteration in range(self.max_iter):
275             self.optimizer.zero_grad(set_to_none=True)
276             total_loss = _compiled_forward(self.theta_alpha, self.E, X_T, K_X_pass, y_one_hot, eye_mask,
C_eff_tensor, diag_K)
277             total_loss.backward()
278             self.optimizer.step()
279
280             with torch.no_grad():
281                 self.alpha = torch.softmax(self.theta_alpha, dim=0).detach()
282                 E_norm = F.normalize(self.E, p=2, dim=1)
283                 M = torch.matmul(E_norm, E_norm.T)
284                 self.M_tensor = torch.where(eye_mask, 1.0, M).detach()
285         else:
286             theta_alpha = torch.zeros(N, device=self.device, requires_grad=True)
287
288             g = torch.Generator(device=self.device)
289             g.manual_seed(42)
290             E_full = torch.zeros(self.num_classes, self.max_latent_dim, device=self.device)
291             E_active = (
292                 torch.randn(
293                     self.num_classes, self.latent_dim, generator=g, device=self.device
294                 )
295                 * self.init_std
296             )
297             E_full[:, :self.latent_dim] = E_active
298             E = E_full.detach().requires_grad_(True)
299
300             def zero_pad_hook(grad):
301                 mask = torch.zeros_like(grad)
302                 mask[:, :self.latent_dim] = 1.0
303                 return grad * mask
304             E.register_hook(zero_pad_hook)
305
306             opt = torch.optim.Adam(
307                 [theta_alpha, E], lr=self.lr, betas=(self.betal, 0.999), fused=True
308             )
309             eye_mask = torch.eye(self.num_classes, device=self.device, dtype=torch.bool)
310
311             cur_batch_size = min(int(b_size), N)
312
313             # REMEDIATION: Dynamically scale C_eff = C * (N / B) for stochastic mini-batch expectations
314             C_tensor = torch.tensor(self.C * (N / cur_batch_size), device=self.device, dtype=torch.float32)

```

```

315
316     # REMEDIATION: Scale squared penalty sum by the exact inverse stochastic factor (B / N)
317     inverse_scale_tensor = torch.tensor(cur_batch_size / N, device=self.device, dtype=torch.float32)
318
319     for step in range(1, self.max_iter + 1):
320         batch_idx = torch.randint(0, N, (cur_batch_size,), device=self.device)
321         y_batch = self.y_train[batch_idx]
322
323         X_batch = self.X_train_proc[batch_idx]
324
325         if self.kernel == "linear":
326             K_X_batch = torch.matmul(X_batch, X_batch.T)
327             diag_K_batch = torch.diag(K_X_batch)
328         else:
329             dist_matrix = compute_dist_matrix(X_batch, X_batch, self.p_norm)
330             K_X_batch = torch.exp(-self.gamma * dist_matrix)
331             diag_K_batch = torch.ones(cur_batch_size, device=self.device, dtype=torch.float32)
332
333         E_norm = F.normalize(E, p=2, dim=1)
334         M = torch.matmul(E_norm, E_norm.T)
335         M = torch.where(eye_mask, 1.0, M)
336         K_Y_batch = M[y_batch.unsqueeze(1), y_batch.unsqueeze(0)]
337
338         opt.zero_grad()
339
340         # REMEDIATION: Strictly evaluate softmax locally over the sub-sampled batch
341         theta_batch = theta_alpha[batch_idx]
342         alpha_batch = torch.softmax(theta_batch, dim=0)
343
344         loss = _compiled_minibatch_unified_loss(
345             alpha_batch, K_X_batch, K_Y_batch, C_tensor, inverse_scale_tensor, diag_K_batch
346         )
347         loss.backward()
348         opt.step()
349
350         with torch.no_grad():
351             self.alpha = torch.softmax(theta_alpha, dim=0).detach()
352             E_norm = F.normalize(E, p=2, dim=1)
353             M = torch.matmul(E_norm, E_norm.T)
354             self.M_tensor = torch.where(eye_mask, 1.0, M).detach()
355
356     return self
357
358 def predict(self, X: torch.Tensor, batch_size: int = None) -> torch.Tensor:
359     """Theorem 1 Generalization: O(1) Vectorized Tensor-Product Inference"""
360     X_test = X.to(self.device)
361
362     b_size = batch_size if batch_size is not None else self.batch_size
363     if X_test.is_sparse:
364         b_size = None
365
366     if b_size is None:
367         if self.hyperbolic:
368             if X_test.is_sparse:
369                 X_test = X_test.coalesce()
370                 X_sq = torch.sparse_coo_tensor(X_test.indices(), X_test.values()*2, X_test.size(), device=self.
371 device).coalesce()
372
373                 norms_sq = torch.sparse.sum(X_sq, dim=1).to_dense()
374                 row_indices = X_test.indices()[0]
375                 # REMEDIATION: Mathematically strict projection mapping (removed inner +1.0)
376                 scale_factors = self.poincare_radius / (torch.sqrt(norms_sq[row_indices]) + 1.0)
377                 X_proc_values = X_test.values() * scale_factors
378                 X_test_proc = torch.sparse_coo_tensor(X_test.indices(), X_test_values, X_test.size(), device=self.
379 device)
380                 X_test_proc = X_test_proc.coalesce()
381             else:
382                 norms_sq = torch.sum(X_test**2, dim=1, keepdim=True)
383                 # REMEDIATION: Mathematically strict projection mapping (removed inner +1.0)
384                 X_test_proc = (X_test / (torch.sqrt(norms_sq) + 1.0)) * self.poincare_radius
385
386         else:
387             X_test_proc = X_test
388
389     K_Y_test = self.M_tensor[self.y_train]
390     aw = self.alpha.unsqueeze(1) * K_Y_test
391
392     if self.kernel == "linear":
393         if self.X_train_proc.is_sparse:
394             W = torch.sparse.mm(self.X_train_proc.t().coalesce(), aw)
395         else:
396             W = torch.matmul(self.X_train_proc.T, aw)

```

```

394         if X_test_proc.is_sparse:
395             scores = torch.sparse.mm(X_test_proc, W)
396         else:
397             scores = torch.matmul(X_test_proc, W)
398     else:
399         dist_test = compute_dist_matrix(X_test_proc, self.X_train_proc, self.p_norm)
400         K_test = torch.exp(-self.gamma * dist_test)
401         scores = torch.matmul(K_test, aw)
402     return torch.argmax(scores, dim=1)
403
404 else:
405     num_test = X_test.shape[0]
406     preds = []
407
408     K_Y_test = self.M_tensor[self.y_train]
409     aw = self.alpha.unsqueeze(1) * K_Y_test
410
411     W = None
412     if self.kernel == "linear":
413         if self.X_train_proc.is_sparse:
414             W = torch.sparse.mm(self.X_train_proc.t().coalesce(), aw)
415         else:
416             W = torch.matmul(self.X_train_proc.T, aw)
417
418     for i in range(0, num_test, int(b_size)):
419         X_test_batch = X_test[i : i + int(b_size)]
420         if self.hyperbolic:
421             norms_sq = torch.sum(X_test_batch**2, dim=1, keepdim=True)
422             # REMEDIATION: Mathematically strict projection mapping (removed inner +1.0)
423             X_test_proc = (X_test_batch / (torch.sqrt(norms_sq) + 1.0)) * self.poincare_radius
424         else:
425             X_test_proc = X_test_batch
426
427         if self.kernel == "linear":
428             scores = torch.matmul(X_test_proc, W)
429         else:
430             dist_test = compute_dist_matrix(
431                 X_test_proc, self.X_train_proc, self.p_norm
432             )
433             K_test = torch.exp(-self.gamma * dist_test)
434             scores = torch.matmul(K_test, aw)
435             preds.append(torch.argmax(scores, dim=1))
436
437     return torch.cat(preds)

```

## A.2 hypersphere\_svm\_golfed.py

Note that this optimal 90-byte implementation was specifically code-golfed for the MNIST dataset.

```
1 class UnifiedHypersphereSVM:  
2     def fit(s,X,y):s.predict=lambda z:y[(X@z/X.std(1)).argmax()]
```

Listing 1: Fully unified end-to-end Support Vector Machine implementation compressed to 90 bytes.