COMP 3105 – Assignment 2 Report
– Fall 2025 –

Due: Sunday October 19, 2025 23:59.
Group 51
Andrew Wallace - 101210291
Christer Henrysson - 101260693

Getting started
Note that Python 3.11 is used for this assignment. Please install requirements using virtual environment via:

```
python3.11 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Question 1 (5%) Binary Classifier (Primal Form)

In this question, you will implement binary classification with different losses from scratch, in Python using NumPy/SciPy, and evaluate their performances on the synthetic datasets from above with different regularization hyper-parameters. You will learn some essential built-in functions like `scipy.optimize.minimize` to solve unconstrained problems and `cvxopt.solvers.qp` to solve quadratic programmings. The input vectors are assumed to be **un-augmented** in this question (i.e. we do not add a constant feature of 1 to it). All of the following functions must be able to handle arbitrary $n > 0$ and $d > 0$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (1%) Implement a Python function

```
w, w0 = minExpLinear(X, y, lamb)
```

Please see `A2codes.py` for the implementation of `minExpLinear(X, y, lamb)`

(b) (1%) Implement a Python function

```
w, w0 = minHinge(X, y, lamb, stablizer=1e-5)
```

Please see `A2codes.py` for the implementation of `minHinge(X, y, lamb, stabilizer=1e-5)`

(c) (1%) Implement a Python function

```
yhat = classify(Xtest, w, w0)
```

Please see `A2codes.py` for the implementation of `classify(Xtest, w, w0)`

(d) (1%) In this part, you will evaluate your implementation with different regularization hyper-parameters. Implement a Python function

```
train_acc, test_acc = synExperimentsRegularize()
```

Please see `A2codes.py` for the implementation of `synExperimentsRegularize()`
The averages over 100 runs for each accuracy (one for training and the other for test) can be seen in the following tables.

λ	ExpLinear			Hinge		
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3
0.001	0.9996	0.5928	0.8779	0.999	0.6169	0.8717
0.01	0.9982	0.602	0.8743	0.996	0.6172	0.8691
0.1	0.9944	0.607	0.8776	0.9917	0.6206	0.8736
1.0	0.9891	0.603	0.8694	0.9806	0.6182	0.8664

Table 1: Training accuracies with different hyper-parameters

λ	ExpLinear			Hinge		
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3
0.001	0.98998	0.56133	0.8811	0.98865	0.59616	0.87799
0.01	0.98773	0.57185	0.88047	0.98587	0.60043	0.87767
0.1	0.98633	0.57489	0.88127	0.9839	0.60185	0.87848
1.0	0.9812	0.57157	0.87411	0.97397	0.60293	0.87171

Table 2: Test accuracies with different hyper-parameters

(e) (1%) Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reasons behind them.

We can see that the training accuracies are higher when λ approaches zero. This is because when lambda approaches zero, the regularization term $\frac{1}{2}\lambda||\mathbf{w}||_2^2$ also approaches zero. This term is used to prevent overfitting by penalizing complex models. That is, the greater $||\mathbf{w}||_2^2$ is (more complex), the greater the value of our objective function will be. The parameter λ determines how complex our model is allowed to be. When λ is large, the regularization term dominates, and our objective turns into minimizing the complexity of our model. When λ is small, the loss function dominates, and our objective turns into minimizing the loss of our model. So, we see the accuracies of our training data increase (from model to model) as our λ decreases, since we primarily focus on minimizing the loss. Conversely, as λ gets larger (closer to 1), our accuracies decrease.

Note that this is not necessarily true when it comes to testing. If our model is overfitted to our training data, then our accuracies for test data will suffer. See the test accuracy for `ExpLinear`, Model 2. Here we can see that $\lambda = 0.01, 0.1,$ and 1.0 all have higher accuracies compared to $\lambda = 0.001$. This is likely due to the simplified models used (as λ increases) being less overfitted to the training data and thus a better fit for the test data.

Question 2 (5%) Binary Classification (Adjoint Form)

In this question, you will implement binary classification with different losses (again) using the adjoint formula coming from the representer theorem, and evaluate their performances on the synthetic datasets from above with different kernels. The input vectors are assumed to be **un-augmented** in this question (i.e. we do not add a constant feature of 1 to it). All of the following functions must be able to handle arbitrary $n > 0$ and $d > 0$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (1%) Implement a Python function

```
a, a0 = adjExpLinear(X, y, lamb, kernel_func)
```

Please see `A2codes.py` for the implementation of `adjExpLinear(X, y, lamb, kernel_func)`

(b) (1%) Implement a Python function

```
a, a0 = adjHinge(X, y, lamb, kernel_func, stabilizer=1e-5)
```

Please see `A2codes.py` for the implementation of `adjHinge(X, y, lamb, kernel_func, stabilizer=1e-5)`

(c) (1%) Implement a Python function

```
yhat = adjClassify(Xtest, a, a0, X, kernel_func)
```

Please see `A2codes.py` for the implementation of `adjClassify(Xtest, a, a0, X, kernel_func)`

(d) (1%) In this part, you will evaluate your implementation with different kernels. Implement a Python function

```
train_acc, test_acc = synExperimentsKernel()
```

Please see `A2codes.py` for the implementation of `synExperimentsKernel()`.
The results can be seen in the tables below.

Kernel	ExpLinear			Hinge		
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3
Linear	0.998	0.599	0.878	0.996	0.621	0.878
Poly($d = 2$)	1.0	1.0	0.875	1.0	0.999	0.871
Poly($d = 3$)	1.0	1.0	0.999	1.0	1.0	0.999
Gauss($\sigma = 1$)	1.0	1.0	1.0	1.0	1.0	1.0
Gauss($\sigma = 0.5$)	1.0	1.0	1.0	0.999	1.0	1.0

Table 3: Training accuracies with different kernels

Kernel	ExpLinear			Hinge		
	Model 1	Model 2	Model 3	Model 1	Model 2	Model 3
Linear	0.988	0.5561	0.8827	0.9839	0.5875	0.881
Poly($d = 2$)	0.9786	0.9736	0.8792	0.9801	0.9643	0.8771
Poly($d = 3$)	0.9793	0.9693	0.9965	0.9766	0.9636	0.9918
Gauss($\sigma = 1$)	0.9784	0.9664	0.9977	0.9686	0.9616	0.9953
Gauss($\sigma = 0.5$)	0.9648	0.9556	0.9982	0.9595	0.9482	0.9976

Table 4: Test accuracies with different kernels

(e) (1%) Looking at your tables from above, analyze the results and discuss any findings you may have and the possible reasons behind them.

Looking at the tables from above we see that the linear kernel for both ExpLinear and Hinge on Model 2 has poor performance in both training and test accuracies. Since the data in Model 2 is not linearly separable, and we try to use a linear kernel, we can see that the accuracy suffers. That is, it is not able to accurately classify the data when using a linear classifier (on Model 2). This is further supported by the results from Kernels Poly($d = 2$) and Poly($d = 3$) having higher accuracies (for both ExpLinear and Hinge) on Model 2. Furthermore, the accuracies on Model 3 when using a linear kernel are worse than Model 1. Again, this is because the data in Model 3 is not entirely linear separable. So using a linear kernel classifier on this data will not have a high accuracy.
Second, we see that the training accuracies are almost 1 for all three models (for both ExpLinear and Hinge) with a Poly($d = 3$) kernel. This kernel allows for a non linear model that can fit the data more accurately (e.g., fitting curved boundaries), which is suitable for all three models. We also see high accuracies in the test set for the same reason.
Third, the Gauss($\sigma = 1$) appears to have the best overall accuracy across the board. This is likely due to it's flexibility and it's focus on localized points making it robust to outliers.
Lastly, we notice that most of the training accuracies are high. This is likely due to the low value of $\lambda = 0.001$, allowing the model to be more complex, and thus fitting (maybe overfitting) the training data.

Question 3 (5%) Binary Classification (SVM Dual Form)

In this question, you will implement binary classification with the hinge loss (yet again) using the dual formula, and choose the best hyper-parameter and kernel for some real-world problems via cross-validation. The input vectors are assumed to be **un-augmented** in this question (i.e. we do not add a constant feature of 1 to it). All of the following functions must be able to handle arbitrary $n > 0$ and $d > 0$. The vectors and matrices are represented as NumPy arrays. Your functions shouldn't print additional information to the standard output.

(a) (1%) Implement a Python function

```
a, b = dualHinge(X, y, lamb, kernel_func, stabilizer=1e-5)
```

Please see `A2codes.py` for the implementation of `dualHinge(X, y, lamb, kernel_func)`

(b) (1%) Implement a Python function

```
yhat = dualClassify(Xtest, a, b, X, y, lamb, kernel_func)
```

Please see `A2codes.py` for the implementation of `dualClassify(Xtest, a, b, X, y, lamb, kernel_func)`

(c) (2%) The A2files.zip includes an image dataset, A2train.csv, of handwritten digits taken from the MNIST dataset. Each image is either digit 4 or digit 9 (once loaded, you can call the `plotDigit` function to see some samples of the images as in Fig. 4). The first column of the csv file is the class label. Treat digit 4 as the -1 class and digit 9 as the +1 class, your task is to use your `dualHinge` function to learn a good binary classifier. In this part, you need to perform cross-validation and select the best hyperparameters and kernels for this dataset. Implement a Python function

```
cv acc, best lamb, best kernel = cvMnist(dataset_folder, lamb_list,
                              kernel_list, k=5)
```

Cross-validation results. The average validation accuracies (rows = λ, columns = kernel) are:

λ	Linear	Poly(d=2)	Gauss(σ=1.0)
0.001	0.958	**0.978**	0.493
0.01	0.958	**0.978**	0.493
0.1	0.958	**0.978**	0.493

Table 5: Q3(c) average validation accuracies for MNIST (4 vs 9). Best setting: λ=0.001, Poly(d=2).

Best choice. We select λ=0.001 with Polynomial kernel of degree 2, which acheves the highest average validation acuracy ($\approx$ 0.978) across all tested λ values. The Gaussian kernel with σ=1.0 underperforms on raw pixel features (784-D), likely due to an overly small bandwidth causing near-identity similarities off-diagonal.

(d) (1%) We will evaluate your choices from (c) on a new test dataset (that you don't have access to). You will get full mark here if your chosen hyper-parameters, kernel function, dualHinge and dualClassify can achieve acceptable performance on the test dataset.

References

The following ChatGPT prompt and response was used in this assignment:
Prompt:
How does the Gaussian kernel with sigma =1 work at a high level

Response:
Perfect — let's go step by step through **how the Gaussian (RBF) kernel actually performs classification**, both conceptually and computationally.

—

1. Goal of kernel-based classification

At the core (e.g., in an **SVM**), the classifier wants to find a **decision boundary** that separates two classes with the **maximum margin**.

For linear data, that's just a straight hyperplane:

$f(x) = w^T x + b$

and you classify by the **sign** of (f(x)).

But when the data are **not linearly separable**, we need to find a boundary in a **higher-dimensional space** where it *is* separable.

2. The kernel trick

Instead of explicitly transforming (x) into a high-dimensional feature space (which could be infinite-dimensional), we use the **kernel trick**:

$K(x_i, x_j) = \exp\left(-\frac{|x_i - x_j|^2}{2\sigma^2}\right)$

This lets us compute inner products in that feature space **without ever constructing it**.

So the classifier's decision function becomes:

$f(x) = \sum_{i=1}^{N} \alpha_i y_i K(x_i, x) + b$

where:

   1. (x_i) are the **training samples**,
   2. $(y_i \in +1, -1)$ are their labels,
   3. (α_i) are learned weights (nonzero only for **support vectors**),
   4. $(K(x_i, x))$ measures **similarity** between the new point and each support vector.

You classify by taking the **sign** of (f(x)):

$\text{class}(x) = \text{sign}(f(x))$

3. What happens when you classify a new point

When a new input (x) arrives:

1. **Compute similarity to each support vector:**
   1. If (x) is close to (x_i), $(K(x_i, x) \approx 1)$.
   2. If (x) is far, $(K(x_i, x) \approx 0)$.
   3. With $\sigma = 1$, "close" means within roughly one standard deviation in feature space.
2. **Weight those similarities by their learned importance:**
   1. Each support vector "votes" for its class, weighted by $(\alpha_i y_i)$.
   2. Points near the boundary have higher σ-values — they're the most influential.
3. **Sum the votes** to get (f(x)).

1. If the weighted sum is positive → classify as +1
2. If negative → classify as -1

4. Intuition — “ripples” in feature space
Imagine each **support vector** radiates a **smooth Gaussian bump** of influence over space.
When classifying, you’re effectively adding up all those bumps:

1. Points near positive examples raise (f(x)).
2. Points near negative examples lower (f(x)).
3. The final decision boundary forms where the sum crosses 0 — the **ridge between the bumps**.

Because the Gaussian decays smoothly, the resulting decision boundary is **curved and continuous**, adapting to the data shape.
5. Why this works so well

1. It automatically focuses on **local neighborhoods** (since far-away points have negligible influence).
2. It builds a **smooth but nonlinear** decision boundary.
3. It requires only a few **support vectors** to represent the learned function.
4. It generalizes well because it balances local adaptability with global smoothness (especially when $\sigma = 1$).

Example (conceptually)
Suppose your data look like concentric circles — one class inside, one outside.
A linear classifier can’t separate them.
But with the Gaussian kernel:

1. Points inside the same ring are *close* (high similarity).
2. Points across rings are *far apart* (low similarity).
3. The classifier can now easily draw a **circular decision boundary** in the original space.

Would you like me to show you a **visual animation** (or static plot) of how an SVM with a Gaussian kernel classifies points in 2D? It makes this process crystal clear.