OMDS Final Project Report

Christian Buda

0 Targets

The letters used for classification in this project are B and U (for the first 4 questions) and B,U and C (for the last question).

1 PART 1 - MLP

1.1 QUESTION 1 - MLP

The task is to build an MLP for binary classification. Since we have that:

$$\text{softmax}(z_1, z_2)_1 = \frac{e^{x_1}}{e^{x_1} + e^{x_2}} = \frac{1}{1 + e^{x_2 - x_1}} = \text{sigmoid}(x_1 - x_2)$$

I decided to make the network output a single value, and use the sigmoid as the final activation function.

Aside from the details in the implementation, the most important thing to train a neural network model is to find a way to compute the gradient of the loss as a function of the inputs. Let's start by computing some of the derivatives of the scalar functions used throughout this implementation:

$$\begin{aligned}
&\frac{d}{dx} g(x) = \frac{d}{dx} \tanh \sigma x = \frac{d}{dx} \frac{e^{2\sigma x} - 1}{e^{2\sigma x} + 1} = \sigma(1 - g(x)^2) \\
&\frac{d}{dx} \text{sigmoid}(x) = \frac{d}{dx} \frac{1}{1 + e^{-x}} = \text{sigmoid}(x)(1 - \text{sigmoid}(x)) \\
&\frac{d}{dp} \text{CE}(y, p) = -\frac{d}{dp} \left(y \log (p + \epsilon) + (1 - y) \log (1 - p + \epsilon) \right) = \frac{1 - y}{1 - p + \epsilon} - \frac{y}{p + \epsilon}
\end{aligned}$$

where I added a small ϵ to the loss to improve numerical stability. Here I describe the idea needed to explain how the gradient of the loss wrt the inputs is computed in the code, which is nothing more than backpropagation. For convenience in the implementation, I exploited the fact that the network only needs one output neuron, so that the jacobian of the network is effectively just a gradient, and we can compute the gradient of the loss using the chain rule one more time starting from the network gradient. I'll focus on an example, let's suppose that the network only has 4 layers g_i with weights W_i, and let's call z_i the output of layer i. Then we have:

$$\begin{aligned}
\left(\frac{\partial z_4}{\partial W_4}\right)^T &= \left(\frac{\partial g_4}{\partial W_4}(z_3, W_4)\right)^T \mathbf{1} \\
\left(\frac{\partial z_4}{\partial W_3}\right)^T &= \left(\frac{\partial g_3}{\partial W_3}(z_2, W_3)\right)^T \left(\frac{\partial g_4}{\partial z}(z_3, W_4)\right)^T \mathbf{1} \\
\left(\frac{\partial z_4}{\partial W_2}\right)^T &= \left(\frac{\partial g_2}{\partial W_2}(z_1, W_2)\right)^T \left(\frac{\partial g_3}{\partial z}(z_2, W_3)\right)^T \left(\frac{\partial g_4}{\partial z}(z_3, W_4)\right)^T \mathbf{1} \\
\left(\frac{\partial z_4}{\partial W_1}\right)^T &= \left(\frac{\partial g_1}{\partial W_1}(z_0, W_1)\right)^T \left(\frac{\partial g_2}{\partial z}(z_1, W_2)\right)^T \left(\frac{\partial g_3}{\partial z}(z_2, W_3)\right)^T \left(\frac{\partial g_4}{\partial z}(z_3, W_4)\right)^T \mathbf{1}
\end{aligned}$$

Since z_4 is just a scalar, then ∂g_4 is a vector and we only need a way to compute the vector-jacobian products $(\partial g_i)^T v$. Note that there are a lot of common terms across the various rows, this is useful to note since it allows to speed up the computation.

A dense layer (without activation) is of the form $g(z) = \sigma(Wz + b)$, with $W \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ the weights of the layer, and σ the activation function. It's easy to see then, that:

$$\begin{aligned}
(\partial_z g)^T v &= W^T (\sigma'(Wz + b) \odot v) \\
(\partial_W g)^T v &= z(\sigma'(Wz + b) \odot v)^T \\
(\partial_b g)^T v &= \sigma'(Wz + b) \odot v
\end{aligned}$$

With these facts in mind, it's easy to build an algorithm that, starting from the last layer, computes the gradient of the whole network one step at a time by using the vector-jacobian products above, and this is exactly the one used in the code. One important thing to notice is that, in this approach, all the intermediate outputs z_i need to be recomputed (or stored, in a more efficient setting) each time the gradient is computed. Now that we have a way to compute the gradients, the network is optimized using the function *minimize* from the *scipy* library. The optimization method used is L-BFGS-B with a tolerance of 10^{-4}. The hyperparameters were chosen using a 5-fold cross validation and the optimization was successful in all the tested configurations. The final hyperparameters are:

$$N = 8 \qquad H = 1 \qquad \sigma = 1$$

chosen among $N \in \{1, 2, 4, 8, 16\}$, $H \in \{1, 2, 3, 4\}$, $\sigma \in \{10^{-2}, 10^{-1}, 1, 10\}$. The average cross validation accuracy and error were, respectively, 0.996 and 0.00363.

After the cross validation, the final model was retrained on the entire training set. The optimization was performed in 0.1 seconds and 24 iterations were needed to reach convergence (returned message: "CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH"). The optimizer called the objective function 28 times, and called the gradient of the objective function 28 times. The starting and final values of the objective functions (on the training set) are, respectively, 0.693 and 0.00453.

The starting training accuracy was 0.472, meanwhile the final value is 1. The final test accuracy is also 1, so the model was complex enough to be able to learn and generalize in this task. The starting training error was 0.693 (similar to the regularized one, since the initialization of the weights was done with a gaussian of zero mean and 0.01 standard deviation), while the final value was 0.000915 (a reduction of a factor 10^3!). The final test error is 0.00266, a bit higher than the training one, but still lower than the starting training error; this may indicate a slight tendency to the overfit, but the task was so easy for the model that it's hard to tell.

1.2 QUESTION 2 - RBF

Just as the previous section, I used a sigmoid activation on the output of the RBF network. The output of this network can then be written in closed form as:

$$p(x) = \text{sigmoid}\left(\sum_{i=1}^{N} w_i \phi(\|x - c_i\|)\right)$$

where x is the input vector, N is the number of neurons, w is the weight vector, $\{c_i\}$ are the N centers, and ϕ is the radial basis function. Three different functions were considered (each with an hyperparameter σ):

- Gaussian: $\phi(r) = e^{-\frac{r^2}{\sigma^2}}$, with derivative $\phi'(r) = -2\frac{r}{\sigma^2}\phi(r)$
- Multiquadric: $\phi(r) = \sqrt{r^2 + \sigma^2}$, with derivative $\phi'(r) = \frac{r}{\phi(r)}$
- Inverse Multiquadric: $\phi(r) = (r^2 + \sigma^2)^{-\frac{1}{2}}$, with derivative $\phi'(r) = -r\phi(r)^3$

As before, we need the gradient of the network, we can compute the unactivated output (i.e. the g s.t. $p(x) = \text{sigmoid}(g(x))$) in closed form here:

$$\nabla_w g(x) = \begin{pmatrix} \phi(\|x - c_1\|) \\ \vdots \\ \phi(\|x - c_N\|) \end{pmatrix}, \qquad \nabla_{c_i} g(x) = w_i \phi'(\|x - c_i\|) \frac{c_i - x}{\|c_i - x\| + \epsilon}$$

where, again, I added a small ϵ term for numerical stability. Using the chain rule, it is easy to find the gradients of the loss wrt the weights and centers, this is what is done in the actual implementation.

The optimization routine consists of 2 steps:

1. First, keeping the centers fixed, we minimize the loss wrt w
2. Then, keeping the weights fixed, we perform a single minimization step of gradient descent for the centers

The first step is performed, as before, using the function *minimize* from the *scipy* library. The optimization method used is L-BFGS-B with a tolerance of 10^{-4}. The second step is performed manually, and an Armijo line search is used to set the step size. The line search was implemented as described here, and the corresponding suggested $\beta = \frac{1}{2}$ and $c_1 = 10^{-4}$ were used. The algorithm stops when the norm of the gradient wrt both the weights and the centers is smaller than a certain tolerance.

The weights are initialized as in the previous steps (i.e. randomly from a normal distribution with 0 mean and 0.01 standard deviation), while the centers are initialized from a standard normal (this is to mimic the fact that the centers are part of the dataset in the standard RBF network, since the input data is scaled to have zero mean and unit variance). An initialization that seemed to give a head start to the training consisted, instead, of choosing the centers as N random elements in the dataset (this is done at training time by reinitializing the centers before running the algorithm).

A tolerance of 10^{-3} was used (with a limit of 10^3 iterations) for the algorithm, as it was observed to work well for convergence while not taking too long to stop. The hyperparameters were chosen using a 5-fold cross validation and the optimization was successful in all the tested configurations. The final hyperparameters are:

$$N = 16 \qquad \sigma = 10 \qquad \phi(r) = (r^2 + \sigma^2)^{-\frac{1}{2}} \quad \text{(Inverse Multiquadric)}$$

chosen among $N \in \{1, 2, 4, 8, 16\}$, $\sigma \in \{10^{-2}, 10^{-1}, 1, 10\}$, and ϕ as a Gaussian, Multiquadric, or Inverse Multiquadric. The average cross validation accuracy and error were, respectively, 0.993 and 0.0767.

After the cross validation, the final model was retrained on the entire training set. The optimization was performed in 13 seconds and 266 iterations were needed to reach convergence. The objective function was called 985 times, the gradient wrt the weights was called 850 times, and the gradient wrt the centers was called 266 times. The starting and final values of the objective functions (on the training set) are, respectively, 1.78 and 0.07998.

The starting training accuracy was 0.487, meanwhile the final value is 0.997. The final test accuracy is also 0.997, showing how, even though the accuracy was not exactly 1, the generalization capabilities of the RBF are really high. The starting training error was 0.693, while the final value was 0.0313 (a reduction of a factor 10^3!). The final test error is 0.0299, even slightly lower than the training one, showing again the great generalization capability of this type of networks.

1.3 Comparisons

Here is a summary table for the two models trained above:

Ex		Settings				Final Error in		Final Accuracy in		Optimization	Call to the	
		H	N	σ	ρ	Train	Test	Train	Test	Time	Objective	Iterations
Q1.1	Full MLP	1	8	1	10^{-4}	0.000915	0.00266	1	1	0.1 s	28	24
Q1.2	RBF		16	10	10^{-4}	0.0313	0.0299	0.997	0.997	12 s	985	266

As we can see, the MLP was able to reach perfect accuracy in both training and testing, all with less neurons and a much faster optimization. The RBF has an higher test error but the value is very consistent with the training value, indicating no sign of overfit; this may be more important than all the other disadvantages in some applications. Since the accuracies are both so high, the confusion matrices are not really meaningful, as we can see below.

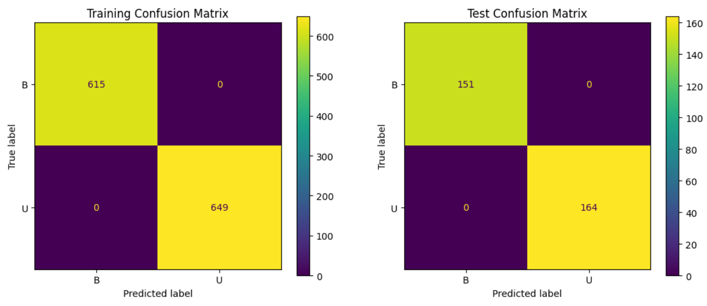

Figure 1: MLP confusion matrices

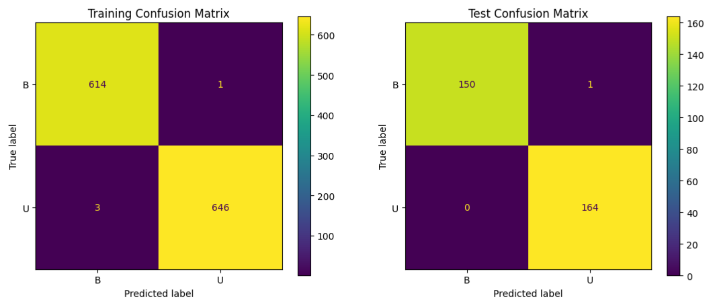

Figure 2: RBF confusion matrices

2 PART 2 - SVM

2.3 QUESTION 3 - CVXOPT

Let's start by writing the dual problem of the kSVM; assume we have a training set $\mathcal{T} = \{(x_i, y_i) | x_i \in \mathbb{R}^p, y_i \in \{0, 1\}, i \in \{1, 2, \ldots, N\}\}$, then we have to solve:

$$\max_{\lambda \in \mathbb{R}^N} -\frac{1}{2} \sum_{i=1}^{N} \lambda_i \lambda_j y_i y_j k(x_i, x_j) + \sum_{i=1}^{N} \lambda_i \qquad \text{subject to} \qquad \sum_{i=1}^{N} y_i \lambda_i = 0 \qquad , \qquad 0 \leq \lambda_i \leq C \quad \forall i$$

where $k(x, y)$ is the kernel function chosen. We need to rewrite it in such a way that we can use the *CVXOPT* routines, in particular, the problem above is equivalent to:

$$-\min_{\lambda \in \mathbb{R}^N} \frac{1}{2} \lambda^T P \lambda + q^T \lambda \qquad \text{subject to} \qquad A\lambda = b \qquad , \qquad (G\lambda)_i \leq h_i \quad \forall i$$

where:

- $P \in \mathbb{R}^{N \times N}$ and $q \in \mathbb{R}^N$ are such that $P_{ij} = y_i y_j k(x_i, x_j)$ and $q_i = -1$ for every i, j
- $A \in \mathbb{R}^{1 \times N}$ and $b \in \mathbb{R}$ are such that $A_{1,i} = y_i$ for all i, and $b = 0$
- $G \in \mathbb{R}^{2N \times N}$ is such that $G_{ij} = -\delta_{ij}$ for all j and for all $i \leq N$, and $G_{ij} = \delta_{ij}$ for all j and for all $i > N$
- $h \in \mathbb{R}^{2N}$ is such that $h_i = 0$ for all $i \leq N$, and $h_i = C$ for all $i > N$

with the problem already set up, we can run the *qp* solver in *CVXOPT* and find the solution we need.

The solver was run with the default settings. The hyperparameters were chosen using a 10-fold cross validation; the final hyperparameters are:

$$C = 0.1 \qquad \qquad \gamma = 0.1$$

chosen among $C \in \{10^{-2}, 10^{-1}, 1, 10\}$, $\gamma \in \{10^{-2}, 10^{-1}, 1, 10\}$. The average cross validation accuracy was 0.990.

After the cross validation, the final model was retrained on the entire training set. The optimization was performed in 2.2 seconds and 11 iterations were needed to reach convergence. The final values of the objective function of the dual is 21.2.

The training accuracy is 0.996, while the test accuracy is 0.994, so the model was able to learn and generalize pretty well.

2.4 QUESTION 4 - MVP-SMO

To implement the *MVP-SMO* algorithm we need two main ingredients: a way to find the most violating pair, and a way to solve the subproblem involving this pair. The first one is pretty straightforward and can be implemented just by following the definitions. The second one is less trivial and we need to do some computations; let's suppose that the most violating pair is indexed as i, j, we then need to solve the following problem:

$$\min_{\lambda_i, \lambda_j} \frac{1}{2} \sum_{i=1}^{N} \lambda_i \lambda_j y_i y_j k(x_i, x_j) - \sum_{i=1}^{N} \lambda_i \qquad \text{subject to} \qquad \sum_{i=1}^{N} y_i \lambda_i = 0 \qquad , \qquad 0 \leq \lambda_i, \lambda_j \leq C$$

By working out the computations and removing, when possible, the additive terms that are constant in λ_i, λ_j, we get:

$$\frac{1}{2} \sum_{i=1}^{N} \lambda_i \lambda_j y_i y_j k(x_i, x_j) - \sum_{i=1}^{N} \lambda_i =$$

$$\frac{\lambda_i^2}{2} k(x_i, x_i) + \frac{\lambda_j^2}{2} k(x_j, x_j) + \lambda_i \lambda_j y_i y_j k(x_i, x_j) + \lambda_i y_i \sum_{l \neq i,j} \lambda_l y_l k(x_i, x_l) + \lambda_j y_j \sum_{l \neq i,j} \lambda_l y_l k(x_j, x_l) - \lambda_i - \lambda_j$$

The equality constraint gives us $\lambda_i = -y_i y_j \lambda_j - y_i \sum_{l\neq i,j} y_l \lambda_l$. By substituting this in the objective we obtain a simple one-variable quadratic minimization problem that can be solved analitically:

$$\frac{1}{2}k(x_i,x_i)\left(\lambda_j y_j + \sum_{l\neq i,j} y_l\lambda_l\right)^2 + \frac{\lambda_j^2}{2}k(x_j,x_j) + \lambda_j y_j k(x_i,x_j)\left(-y_j\lambda_j - \sum_{l\neq i,j} y_l\lambda_l\right)$$

$$+\left(-y_j\lambda_j - \sum_{l\neq i,j} y_l\lambda_l\right)\sum_{l\neq i,j}\lambda_l y_l k(x_i,x_l) + \lambda_j y_j \sum_{l\neq i,j}\lambda_l y_l k(x_j,x_l) + y_i y_j \lambda_j - \lambda_j =$$

$$\frac{1}{2}\lambda_j^2(k(x_i,x_i)+k(x_j,x_j)) + \lambda_j y_j k(x_i,x_i)\sum_{l\neq i,j} y_l\lambda_l - \lambda_j^2 k(x_j,x_j) - \lambda_j y_j k(x_i,x_j)\sum_{l\neq i,j} y_l\lambda_l$$

$$+\, y_j\lambda_j \sum_{l\neq i,j} y_l\lambda_l k(x_j,x_l) - \lambda_j + y_i y_j\lambda_j - y_j\lambda_j\sum_{l\neq i,j} y_l\lambda_l k(x_i,x_l) =$$

$$\frac{\lambda_j^2}{2}\left(k(x_i,x_i)+k(x_j,x_j)-2k(x_i,x_j)\right) + \lambda_j y_j\left(\sum_{l\neq i,j} y_l\lambda_l\left(k(x_i,x_i)-k(x_i,x_j)\right) + y_i - y_j + \sum_{l\neq i,j} y_l\lambda_l\left(k(x_j,x_l)-k(x_i,x_l)\right)\right)$$

It's easy to see that $k(x_i,x_i)+k(x_j,x_j)-2k(x_i,x_j) = k(x_i-x_j,x_i-x_j) \geq 0$, so this problem is convex and it's easy to find the unconstrained solution, which is $-\frac{b}{2a}$, where a,b the coefficients that multiply λ_j^2, λ_j respectively. We now need to impose the two constraints $0 \leq \lambda_i, \lambda_j \leq C$, which can be condensed in the constraint:

$$\max\left\{0, -y_i\sum_{l\neq i,j} y_l\lambda_l - C\right\} \leq \lambda_j \leq \min\left\{C, -y_i\sum_{l\neq i,j} y_l\lambda_l\right\} \qquad \text{if} \quad y_i y_j = 1$$

$$\max\left\{0, y_i\sum_{l\neq i,j} y_l\lambda_l\right\} \leq \lambda_j \leq \min\left\{C, C + y_i\sum_{l\neq i,j} y_l\lambda_l\right\} \qquad \text{if} \quad y_i y_j = -1$$

With this computations, it's easy to implement a solver for the subproblem, which can be found in the code.

The tolerance for this algorithm was set as 10^{-5}; there is also a maximum number of iterations but in practice I observed that this tolerance is always reached in a small number of steps.

Since these this training method should give approximately the same results as the previous one, I used the same hyperparameters as before to be able to make a comparison:

$$C = 0.1 \qquad\qquad \gamma = 0.1$$

The final model was trained on the entire training set. The optimization was performed in 0.26 seconds and 582 iterations were needed to reach convergence. The final values of the objective function of the dual is 21.2. The final difference between $m(\lambda)$ and $M(\lambda)$ is $1 \cdot 10^{-5}$.

The training accuracy is 0.981, while the test accuracy is 0.971, so the model was able to learn and generalize pretty well with this optimization method too.

2.5 QUESTION 5 - MULTICLASS SVM

To be able to use the kSVM as a multiclass classifier, I implemented a one-vs-all strategy in which we train three different models to recognize if the sample belongs to a specific class (i.e. +1) or not (i.e. −1). Using the distance from the hyperplane in the transformed space (i.e. $\frac{|\sum_i y_i\lambda_i^* k(x,x_i)+b^*|}{\sqrt{\sum_{i,j} y_i y_j \lambda_i^*\lambda_j^* k(x_i,x_j)}} = \frac{|\sum_i y_i\lambda_i^* k(x,x_i)+b^*|}{\sqrt{\lambda^T P\lambda}}$), the predicted class can be chosen as the one corresponding to the highest values among these computed distances across the three models. Since the models use the same hyperparameters and the same dataset, the distance should be independent of the specific model and should provide a good indication on which of the models is more "confident" in the prediction.

The models were trained using the *MVP-SMO* method as it turned out to be much faster than using *CVXOPT*, and the hyperparameters were chosen using a 10-fold cross validation; the final hyperparameters are:

$$C = 0.1 \qquad \gamma = 0.1$$

chosen among $C \in \{10^{-2}, 10^{-1}, 1, 10\}$, $\gamma \in \{10^{-2}, 10^{-1}, 1, 10\}$. The average cross validation accuracy was 0.978.

After the cross validation, the final model was retrained on the entire training set. The optimization was performed in 1.8 seconds and $836,\ 792,\ 1128$ iterations were needed to reach convergence in each of the submodels. The final values of the objective function of the dual in each of the submodels are $27,\ 38,\ 36$. The final difference between $m(\lambda)$ and $M(\lambda)$ in the subproblems is $9.94 \cdot 10^{-6},\ 9.25 \cdot 10^{-6},\ 9.63 \cdot 10^{-6}$.

The training accuracy is 0.981, while the test accuracy is 0.976, so the method worked well to solve this multiclass problem.

2.6 Comparisons

Here is a summary table for the three models trained above:

Ex	Settings		Final Accuracy in		Optimization	Final value of		
	C	γ	Train	Test	Time	the Objective	Iterations	KKT Violation
Q2.3	0.1	0.1	0.996	0.994	2.2 s	21.2	11	
Q2.4	0.1	0.1	0.981	0.971	0.26 s	21.2	582	$1 \cdot 10^{-5}$
Q2.5	0.1	0.1	0.981	0.976	1.8 s	$27,\ 38,\ 36$	$836,\ 792,\ 1128$	$9.94 \cdot 10^{-6},\ 9.25 \cdot 10^{-6},\ 9.63 \cdot 10^{-6}$

As we can see, the *CVXOPT* and *MVP-SMO* methods differ slightly in the results, most probably due to a high tolerance in the second algorithm. A percentage point lost in accuracy though, yielded a factor of 10 in speed improvement. The objective functions are identical up to the fifth digit (as you can check in the code).

The Multiclass kSVM cross-validation chose the same values of C and γ for the optimization; the performances are pretty good both in training and testing, while the overhead in training time is pretty significant (three different models trained on a 50% larger datasets).

All three models have pretty similar accuracy in training and testing, so we cannot identify any clear sign of overfit. They also have pretty good values for the accuracy, so there is no sign of underfit either.

From the confusion matrices we can see that the second model tends to predict U more often than necessary. We can also see that this happens with the multiclass case too, even though here it's more reasonable, since C and U can be regarded as pretty similar even by us.

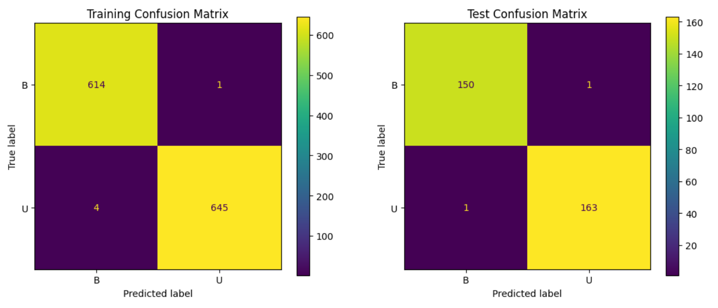

Figure 3: *CVXOPT*-trained kSVM

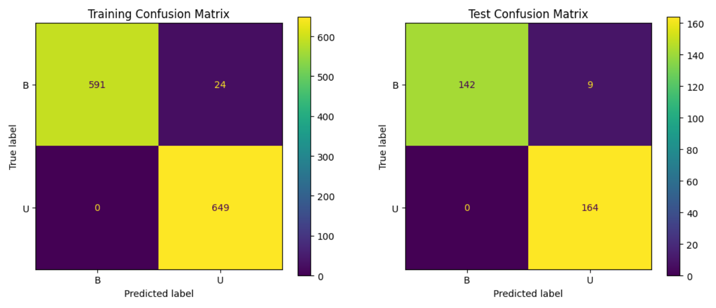

Figure 4: *MVP-SMO*-trained kSVM

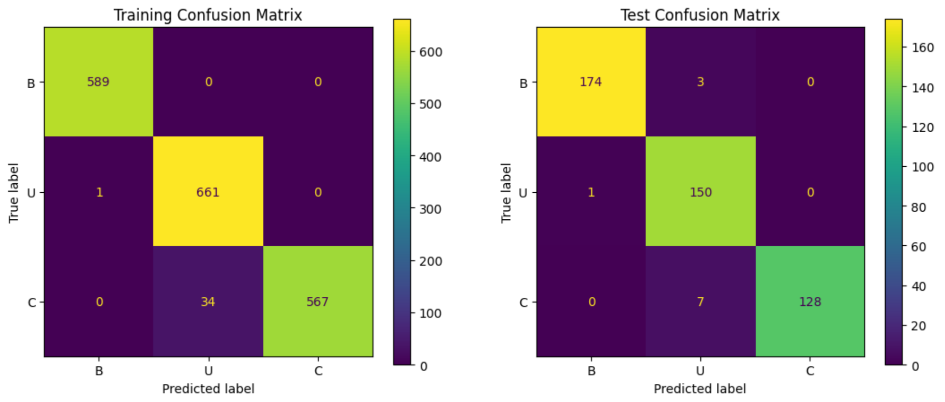

Figure 5: Multiclass kSVM Confusion Matrices