ARISTOTLE UNIVERSITY OF THESSALONIKI

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

Neural Networks: Using SVM for the Classification of Dementia through MRI

Aphrodite Latskou

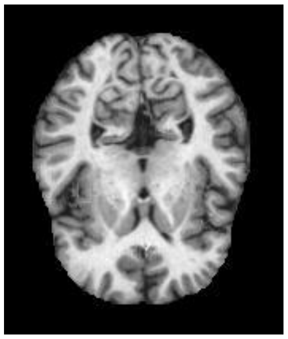
December 2024

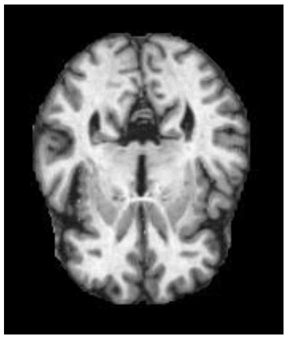
Table of Contents

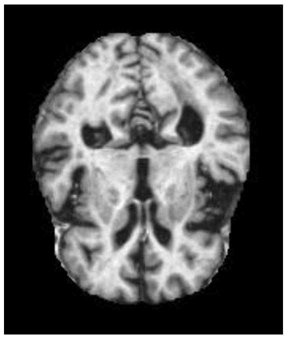
Chapter I: Dataset & Resources

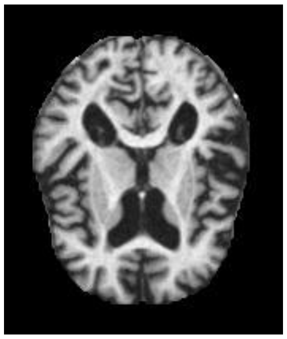
i. Dataset

For this assignment the dataset chosen was the Augmented Alzheimer MRI Dataset from Kaggle. It contains both real and augmented MRI imagery from patients with various dementia levels.


Not Demented


Very Mildly Demented


Mildly Demented


Moderately Demented

More specifically, the dataset contains the following files:

	Original	**Augmented**	**In Total**
Not Demented	3200	9600	12800
Very Mildly Demented	2240	8960	11200
Mildly Demented	896	8960	9856
Moderately Demented	64	6464	6528

Therefore, giving a total of 40384 samples. An important thing to note at this point, is that the samples belonging to each class are not equal. Between the non-demented patients and the moderately demented ones especially, there is a difference of approximately 6000 samples- *half* of the first class. When dealing with imbalanced datasets, it is very likely that the algorithm might have a bias towards the classes with the most samples, therefore learning to favor those categories when it comes to predicting.

ii. Preprocessing

While preprocessing, I used a number of steps to ensure the efficiency of the algorithm and the credibility of our results. I turned the images to true grayscale (since, even though they seem grayscale already, the image arrays had three channels) to have smaller input sizes and since color was non-existent anyways.

Since all the algorithms used in this project take one-dimensional inputs, I also flattened the images. The original images Ire of size 200x190, giving a vector size of 38000 for each sample. That was storage-prohibitive, since NumPy requires a contiguous block of memory to allocate the array, therefore when using the entire images, I would run out of memory often and it would also raise the running time of the algorithm. Through trials, the biggest size for time and space efficiency was found to be 100x95. After the flattening, each sample has a vector size of 2350.

The features Ire also scaled using the Min-Max method, since scaling is known to lead to better training. Finally, I split the data into training and testing sets with a percentage of 60% and 40% accordingly, using the train_test_split function from sklearn.

iii. Resources

The algorithms were partly run on a local AMD Ryzen 5 3500U CPU with 4 cores and 16GB RAM and partly using Google Collab with 12.7GB RAM available.

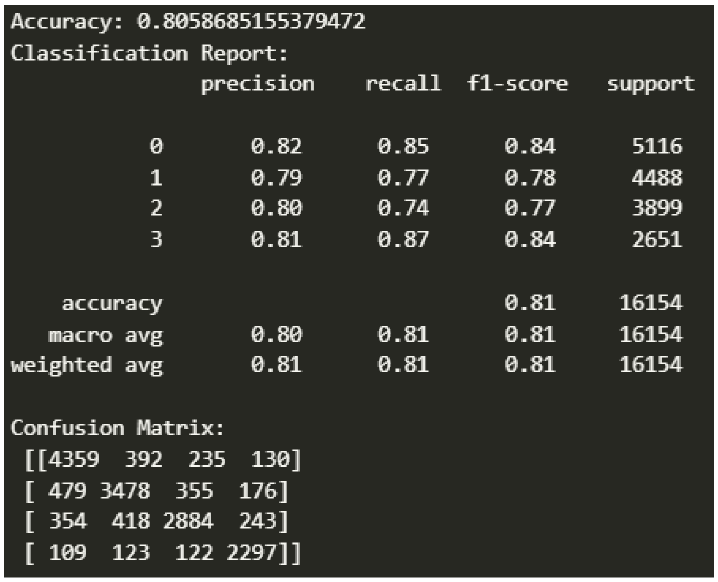
ChatGPT was also partly used for the code production and was found to work very efficiently when using premade functions to create neural network models.

Chapter II: KNN, Nearest Centroid and MLP

i. KNN

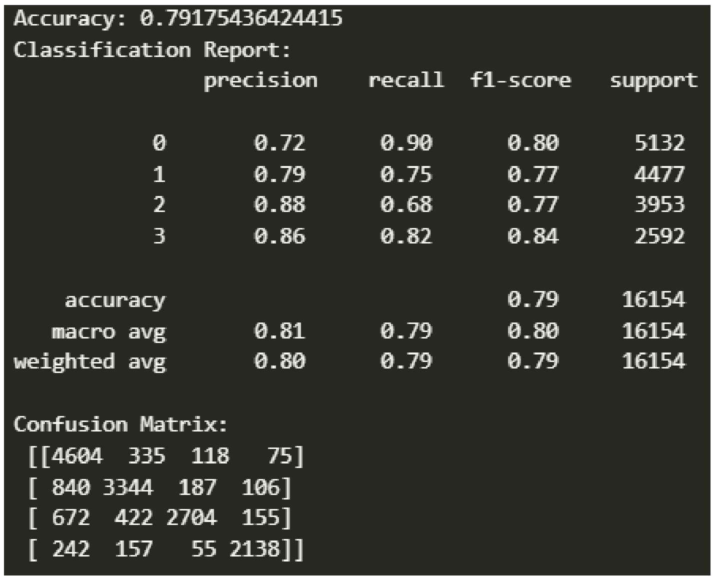
KNN algorithms classify a new sample based on its closest 'neighbors' based on some distance metric. Here, I used the Euclidean distance. The KNearestClassifer from sklearn was used with k = 1 and k = 3 and achieved the following results.

KNN with 1 Neighbor:



```
Accuracy: 0.8058685155379472
Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.85      0.84      5116
           1       0.79      0.77      0.78      4488
           2       0.80      0.74      0.77      3899
           3       0.81      0.87      0.84      2651

    accuracy                           0.81     16154
   macro avg       0.80      0.81      0.81     16154
weighted avg       0.81      0.81      0.81     16154

Confusion Matrix:
 [[4359  392  235  130]
 [ 479 3478  355  176]
 [ 354  418 2884  243]
 [ 109  123  122 2297]]
```

KNN with 3 Neighbors:

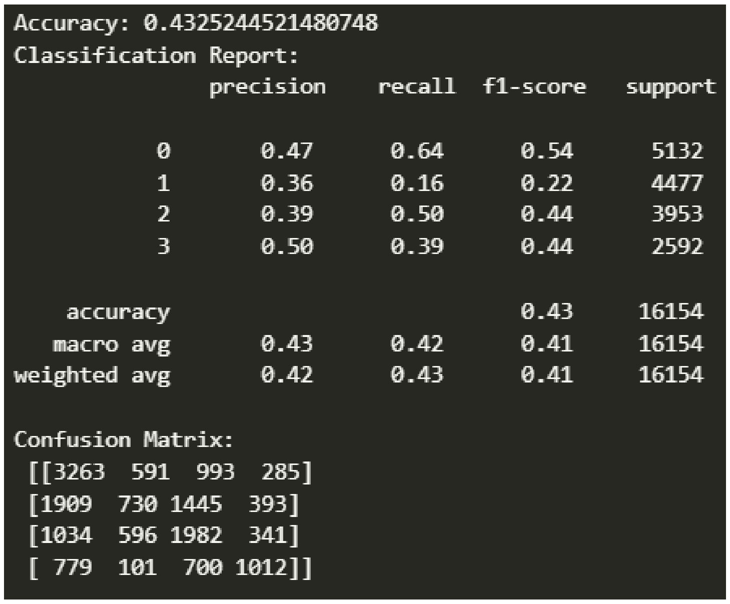


```
Accuracy: 0.79175436424415
Classification Report:
              precision    recall  f1-score   support

           0       0.72      0.90      0.80      5132
           1       0.79      0.75      0.77      4477
           2       0.88      0.68      0.77      3953
           3       0.86      0.82      0.84      2592

    accuracy                           0.79     16154
   macro avg       0.81      0.79      0.80     16154
weighted avg       0.80      0.79      0.79     16154

Confusion Matrix:
 [[4604  335  118   75]
 [ 840 3344  187  106]
 [ 672  422 2704  155]
 [ 242  157   55 2138]]
```

The KNN algorithm performs surprisingly well, and notably, similarly either with 1 or 3 neighbors. Specifically, from the confusion matrices, it is clear that the best metrics are achieved on the Non-Demented and Moderately Demented classes, especially on the recall metric that describes the algorithm's ability to correctly classify a sample belonging to that class. That is always important when dealing with medical data.

ii. Nearest Centroid

Nearest Centroid algorithms calculate the centroid of every class (mean of the samples) and classify every new sample based on the distance from those points. The NearestCentroid from sklearn was used and achieved the following results.

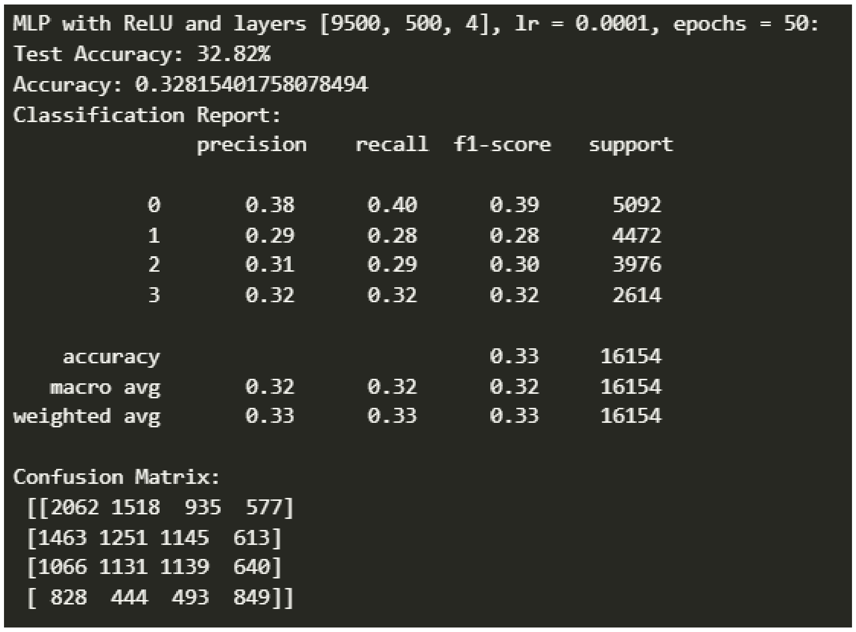


```
Accuracy: 0.4325244521480748
Classification Report:
              precision    recall  f1-score   support

           0       0.47      0.64      0.54      5132
           1       0.36      0.16      0.22      4477
           2       0.39      0.50      0.44      3953
           3       0.50      0.39      0.44      2592

    accuracy                           0.43     16154
   macro avg       0.43      0.42      0.41     16154
weighted avg       0.42      0.43      0.41     16154

Confusion Matrix:
 [[3263  591  993  285]
 [1909  730 1445  393]
 [1034  596 1982  341]
 [ 779  101  700 1012]]
```

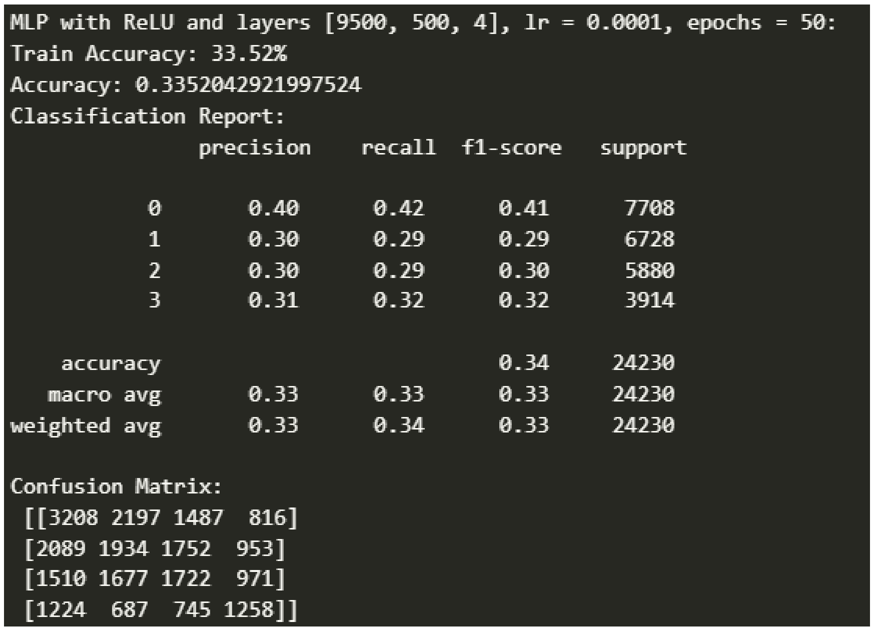
Clearly, the results are very different from the previous algorithms, with Nearest Centroid performing a lot worse in comparison for all 4 classes, and especially for the VeryMildDemented class.

iii. MLP

The custom-made Multi-Layer Perceptron, as discussed in the previous report, was included for comparison purposes. For this specific task, an MLP with one hidden layer of 500 neurons was used with hinge loss for optimization. The following results were obtained:



```
MLP with ReLU and layers [9500, 500, 4], lr = 0.0001, epochs = 50:
Test Accuracy: 32.82%
Accuracy: 0.32815401758078494
Classification Report:
              precision    recall  f1-score   support

           0       0.38      0.40      0.39      5092
           1       0.29      0.28      0.28      4472
           2       0.31      0.29      0.30      3976
           3       0.32      0.32      0.32      2614

    accuracy                           0.33     16154
   macro avg       0.32      0.32      0.32     16154
weighted avg       0.33      0.33      0.33     16154

Confusion Matrix:
 [[2062 1518  935  577]
 [1463 1251 1145  613]
 [1066 1131 1139  640]
 [ 828  444  493  849]]
```

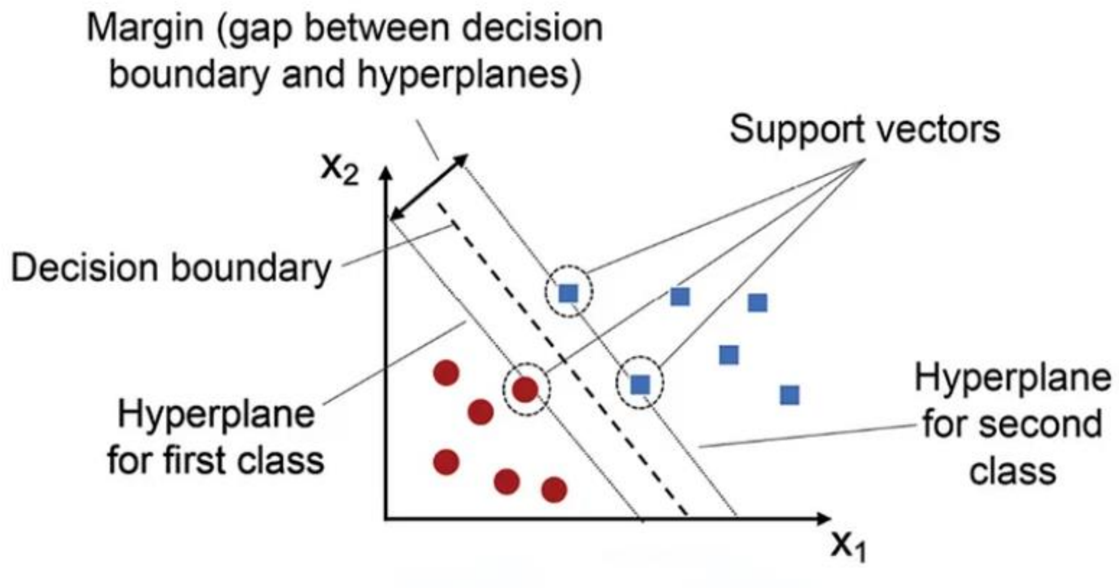


```
MLP with ReLU and layers [9500, 500, 4], lr = 0.0001, epochs = 50:
Train Accuracy: 33.52%
Accuracy: 0.3352042921997524
Classification Report:
              precision    recall  f1-score   support

           0       0.40      0.42      0.41      7708
           1       0.30      0.29      0.29      6728
           2       0.30      0.29      0.30      5880
           3       0.31      0.32      0.32      3914

    accuracy                           0.34     24230
   macro avg       0.33      0.33      0.33     24230
weighted avg       0.33      0.34      0.33     24230

Confusion Matrix:
 [[3208 2197 1487  816]
 [2089 1934 1752  953]
 [1510 1677 1722  971]
 [1224  687  745 1258]]
```

We can see that it performs rather poorly, which, taking into account that I used the custom model I built myself and that one hidden layer does not yield great results in it, is not surprising. The running time however was adequately faster than the SVM algorithm we will see later.

Chapter III: SVM using sklearn

SVMs work by finding the optimal hyperplane that separates data points into different classes. A hyperplane is a decision boundary that separates data points into different classes in a high-dimensional space. The data closest to the hyperplane (or line when we are in 2D spaces as below) are the support vectors, since they define the boundary.


Margin (gap between decision boundary and hyperplanes)
Support vectors
x_2
Decision boundary
Hyperplane for first class
Hyperplane for second class
x_1

When building an SVM model, certain hyperparameters are chosen:

- ***Kernel***: A kernel finds the similarity of the points. Instead of finding f(x,y) directly, it computes the similarity of the image of these points. In other words, instead of finding f(x1,y1) and f(x2,y2) we take the points (x1,y1) and (x2,y2) and compute how similar would their outputs be using a function f(x,y). The kernels tested with on this project were: Linear, RBF and Polynomial
- ***Gamma (γ)***: Defines how far the influence of a single training point reaches in the feature space. It also represents the linearity of the model. A high value leads to more complex decision boundaries while a low one leads to bigger linearity.
- ***Regularization parameter (C)***: Determines how much weight should be given to minimizing classification errors versus maximizing the margin. A low C makes the decision surface smooth, while a high C aims at classifying all training examples correctly by giving the model freedom to select more samples as support vectors.
- ***Degree:*** Only relevant for the polynomial kernel

For my implementation, I used premade functions from sklearn. As such, the following model was used in all the experiments.

svm = SVC(*degree*=degree, *kernel*=kernel, *C*=C, *gamma*=gamma)

In order to visualize the decision boundaries two function were created using ChatGPT and specifically reduce_to_2d_tsne(*X*) and plot_decision_boundaries(*X*, *y*, *model*, *title*="SVM Decision Boundary") that implement the t-SNE method for dimensionality reduction. The code uses t-SNE to reduce high-dimensional data to 2D, making it easier to visualize. After this, it shows how an SVM model separates the data by creating and plotting decision boundaries. A grid is made over the 2D space, and the model predicts class labels for each grid point. The results are displayed as colored regions (decision boundaries) with the original data points plotted on top.

Finally, cross-validation was used in some cases to produce more dependable results and for hyperparameter fine-tuning. The function applied was GridSearchCV() that compared models with predetermined parameter combinations.

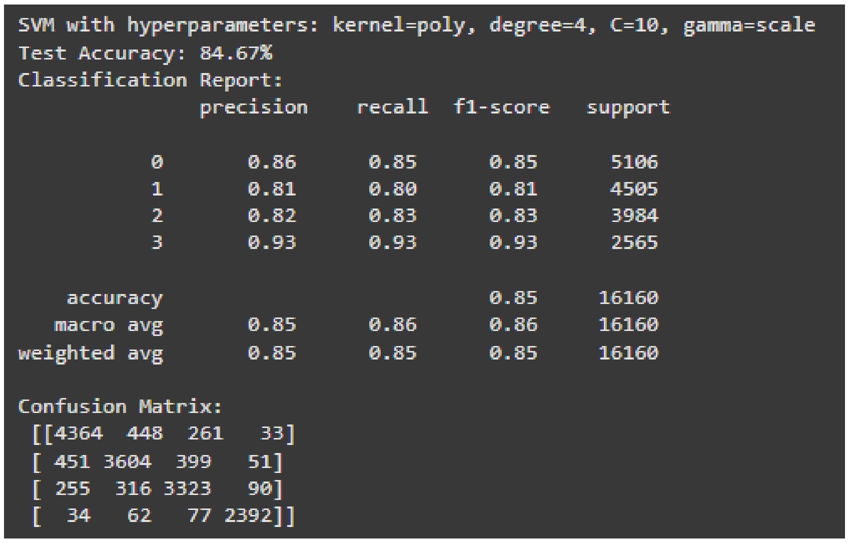
Chapter IV: Results

i. Kernel ranking

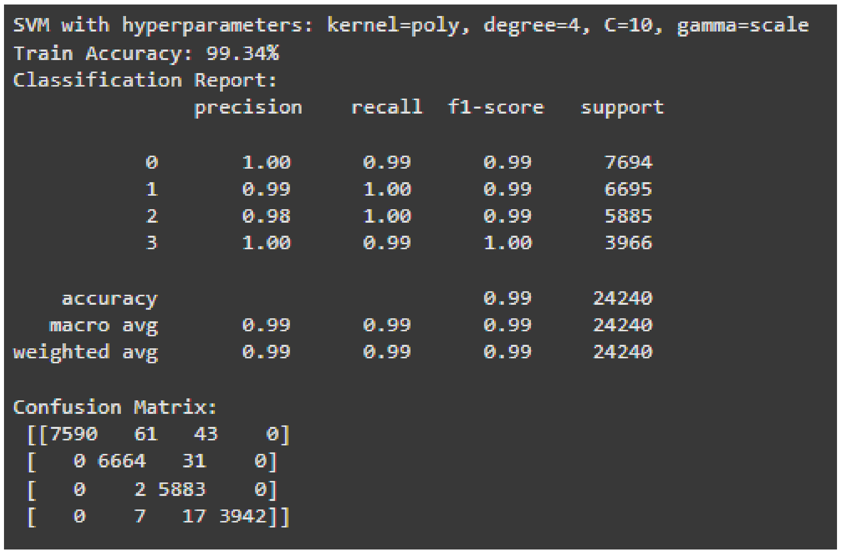
As mentioned, I trained various models with the three kernels linear, polynomial and rbf. Below we have a comparison for all three with parameters C=10, γ='scaled' (this means it is calculated as 1 / (n_features * X.var()) and here is equal to 0.0011) and degree=4 for the polynomial.

	RBF	*Linear*	*Poly*
Time	*2*	*2.5*	*1*
Train Accuracy	*98.34%*	*96.38%*	*99.34%*
Test Accuracy	*82.28%*	*74.94%*	*84.67%*

As we can see, the polynomial kernel performed the best and in the least amount of time. More specifically:



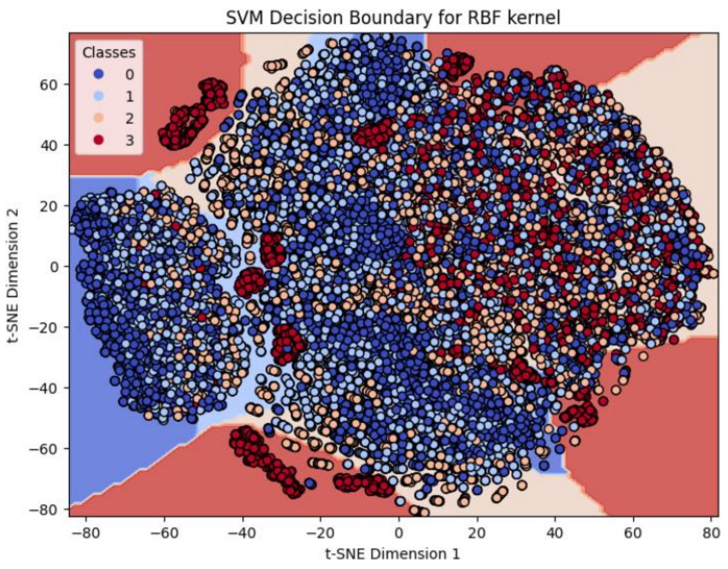
```
SVM with hyperparameters: kernel=poly, degree=4, C=10, gamma=scale
Test Accuracy: 84.67%
Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.85      0.85      5106
           1       0.81      0.80      0.81      4505
           2       0.82      0.83      0.83      3984
           3       0.93      0.93      0.93      2565

    accuracy                           0.85     16160
   macro avg       0.85      0.86      0.86     16160
weighted avg       0.85      0.85      0.85     16160

Confusion Matrix:
 [[4364  448  261   33]
 [ 451 3604  399   51]
 [ 255  316 3323   90]
 [  34   62   77 2392]]
```

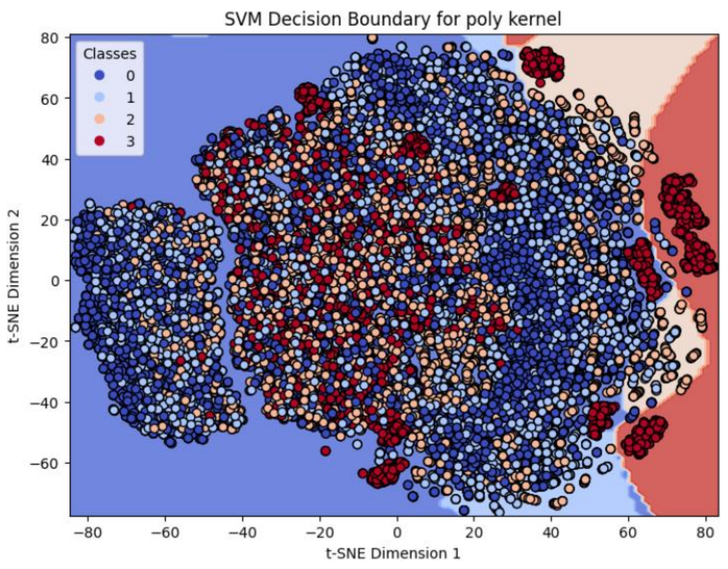


```
SVM with hyperparameters: kernel=poly, degree=4, C=10, gamma=scale
Train Accuracy: 99.34%
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99      7694
           1       0.99      1.00      0.99      6695
           2       0.98      1.00      0.99      5885
           3       1.00      0.99      1.00      3966

    accuracy                           0.99     24240
   macro avg       0.99      0.99      0.99     24240
weighted avg       0.99      0.99      0.99     24240

Confusion Matrix:
 [[7590   61   43    0]
 [   0 6664   31    0]
 [   0    2 5883    0]
 [   0    7   17 3942]]
```

The training accuracy is larger by more than 10%, which means an amount of overfitting could not be avoided, still the results were adequately high, and the metrics reached satisfactory, which can be appointed to how the polynomial kernel works best with very dense datasets. The linear kernel naturally performed the worst, as it is better suited for datasets that already are distributed in a linear and sparse fashion, which in our instance is not the case.

To further understand this point, the data were transformed into the 2D space with the functions mentioned. Since our data are flattened images, that amounts to big dimensionality (9500 specifically) and thus, this reduction led to substantial information loss, resulting in lower performance and accuracy metrics. Still, it is a good way to visualize our dataset as well as how different kernels behave in the 2D space. Those experiments are depicted below.





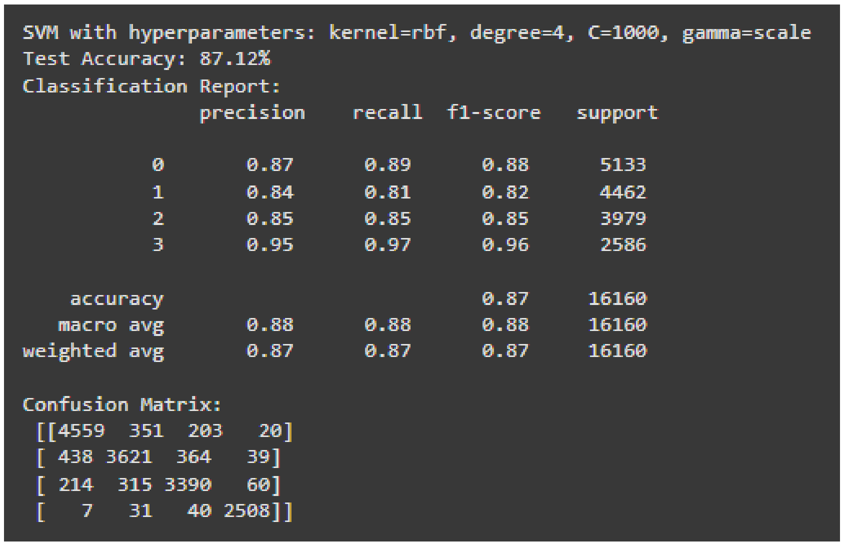
The comparison was focused primarily on the RBF and poly kernels, since they performed very similarly, and I wanted to see the difference in their exact behavior. As we can see, the RBF adapted better to the existence of multiple classes so densely close to one another without overfitting to the data. Specifically, it is a lot better at finding clusters of data within the same class, as well as drawing more complex and flexible lines between the classes. That fact unfortunately also makes it more prone to overfitting. On the other hand, the polynomial kernel creates smoother boundaries -'looking at the bigger picture', one might say- and is a lot more likely to underfit.

ii. C ranking

To see the effect of the C regularization parameter, we used the RBF kernel with the same gamma value as before.

	0.1	*1*	*10*	*100*	*1000*
Time	*3*	*3.5*	*2*	*2*	*1.5*
Train Accuracy	*59%*	*82.28%*	*98.34%*	*99.34%*	*99.56%*
Test Accuracy	*57.79%*	*73.56%*	*82.28%*	*84.67%*	*87.12%*

A tendency for the running time to reduce as the C value increased was noticed and of course, with larger C value we obtain better metrics, accuracy among them.



```
SVM with hyperparameters: kernel=rbf, degree=4, C=1000, gamma=scale
Test Accuracy: 87.12%
Classification Report:
              precision    recall  f1-score   support

           0       0.87      0.89      0.88      5133
           1       0.84      0.81      0.82      4462
           2       0.85      0.85      0.85      3979
           3       0.95      0.97      0.96      2586

    accuracy                           0.87     16160
   macro avg       0.88      0.88      0.88     16160
weighted avg       0.87      0.87      0.87     16160

Confusion Matrix:
 [[4559  351  203   20]
 [ 438 3621  364   39]
 [ 214  315 3390   60]
 [   7   31   40 2508]]
```

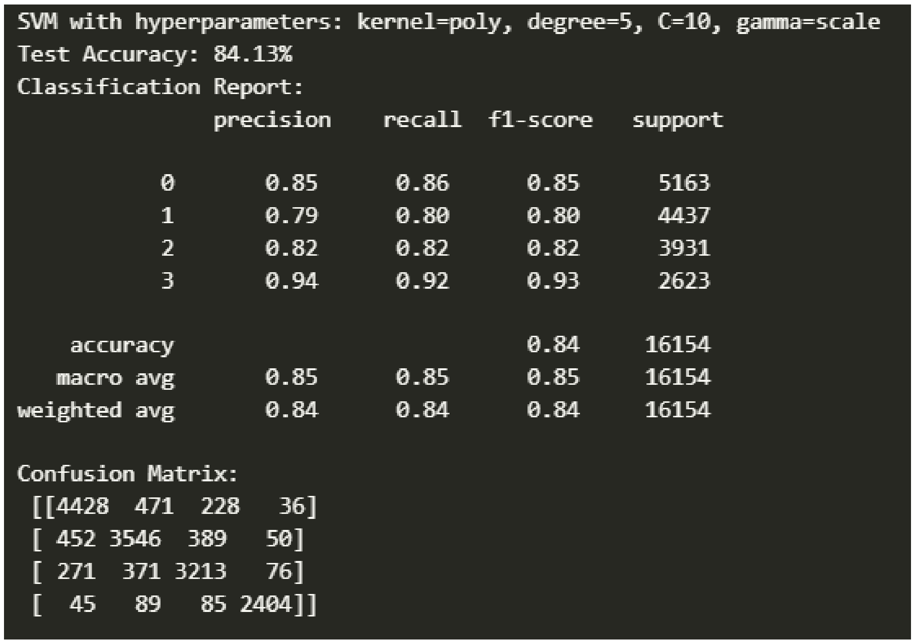
Clearly, having more samples decide on the boundaries leads to a more correct classification in this case, since clusters are often found in the dataset. More specifically, in overlapping clusters, the SVM must heavily focus on accurately classifying points near the boundaries and a large C ensures these misclassifications are minimized by allowing the decision boundary to fit the data more closely. In our case, tight and intricate boundaries fit the best.

iii. Degree ranking

To test different degree values, the polynomial kernel was naturally used with C=10 and γ='scale'.

	3	*4*	*5*
Time	*2*	*1.5*	*1*
Train Accuracy	*89.69%*	*99.34%*	*99.35%*
Test Accuracy	*70.54%*	*84.67%*	*84.13%*

The performance of the algorithm improves significantly after the value of 4, which is easily explained since we have 4 classes, and a 4-degree polynomial is a lot better suited to tackle this classification problem. Below, the more elaborate results for a 5-degree polynomial kernel is included:



```
SVM with hyperparameters: kernel=poly, degree=5, C=10, gamma=scale
Test Accuracy: 84.13%
Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.86      0.85      5163
           1       0.79      0.80      0.80      4437
           2       0.82      0.82      0.82      3931
           3       0.94      0.92      0.93      2623

    accuracy                           0.84     16154
   macro avg       0.85      0.85      0.85     16154
weighted avg       0.84      0.84      0.84     16154

Confusion Matrix:
 [[4428  471  228   36]
 [ 452 3546  389   50]
 [ 271  371 3213   76]
 [  45   89   85 2404]]
```

iv. K-fold Cross-Validation and Hyperparameter Tuning

In order to have more robust results and find the best parameters without trying every different combination, 3-fold cross-validation was used. Since it is a very time-consuming process, I ran it only a couple of times and for limited parameter choices. Thus, the function made combinations using grid search among:

- kernel: RBF, polynomial
- C: 0.01, 10, 100, 1000
- γ: 0.001, 0.1, ‘scale’
- degree: 3, 4, 5

Of all the possible combinations, the best one based on accuracy scores reached was an **RBF kernel with C=1000 and γ=’scale’**, giving an accuracy of **86.64%.**

Chapter V: Conclusions

i. Comparison and Discussion

In the end, the observations from the various tests performed can be summarized by the following:

- The RBF kernel is the most effective at tackling complex, dense classification problems, something that comes to no surprise seeing as it is the most widely used kernel trick.
- Choosing the best C and gamma values is one of the most important steps, and can lead to big improvements in our results

Overall, the SVM algorithm managed to tackle the classification problem with great efficiency and reached as high as **87.12% accuracy** on the test set and 99% on the training set. That makes it the best algorithm yet to tackle the classification of the MRI images in our Dementia dataset, since it surpassed the KNN and Nearest Centroid algorithms and did a lot better than the MLP, both using PyTorch and the custom-made on from the previous report.

ii. Limitations

While the results were satisfactory, my approach still has shortcomings that should be acknowledged. I decided to run my neural networks on my local CPU, which proved to be a difficult and sub-optimal task. Since the running times were high and it was very heavy on the RAM, there was a limit to the number of things I had time to try.

More hyperparameter tuning with a wider variety of values could also possibly lead to better metrics. Even better, an extensive search outside of pre-determined values would have been preferred but would have been very time-consuming.

PCA could also have been used to reduce the complexity and the running time, but since good metrics were achieved, it was not deemed as necessary.

Finally, due to the high dimensionality of the data, feature engineering in order to find better features with actual importance rather than simply the value of every pixel could not only offer a better solution but would also increase the interpretability of our results and point to more specific characteristics that describe each dementia level.

References

baeldung. (2020, October 7). *Multiclass Classification Using Support Vector Machines | Baeldung on Computer Science*. Www.baeldung.com. https://www.baeldung.com/cs/svm-multiclass-classification

GeeksforGeeks. (2024, February 23). *How to Choose the Best Kernel Function for SVMs*. GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/how-to-choose-the-best-kernel-function-for-svms/

SVC. (2024). Scikit-Learn. https://scikit-learn.org/1.5/modules/generated/sklearn.svm.SVC.html

Tibrewal, T. P. (2023, September 15). *Support Vector Machines (SVM): An Intuitive Explanation*. Low Code for Data Science. https://medium.com/low-code-for-advanced-data-science/support-vector-machines-svm-an-intuitive-explanation-b084d6238106