ML EndSem Report

K.Rishitha, DA24B040

Algorithms:

Logistic Regression(Multivariate):

When I first trained Logisitic Regression with parameters

```
learning rate=0.1 and epochs=1000, i got Accuracy=0.9000 and F1 score= 0.8996
```

Then I hypertuned the parameters to get the following results:

learning_rate	epochs	f1_score	training_loss	validation_loss
0.01	500	0.832952	0.8050	0.8013
0.01	1000	0.860669	0.6161	0.6123
0.01	2000	0.873619	0.4927	0.4918
0.10	500	0.889835	0.3906	0.3979
0.10	1000	0.898810	0.3372	0.3550
0.10	2000	0.907641	0.2949	0.3281
0.50	500	0.910423	0.2828	0.3226
0.50	1000	0.912920	0.2490	0.3118
0.50	2000	0.910079	0.2175	0.3129

Table 1: Hyperparameter Tuning Results

```
Best Performing Model:
learning_rate          0.50000
epochs              1000.00000
f1_score               0.91292
training_loss          0.24900
validation_loss        0.31180
```

- From the above table when I fixed learning rate and vary the epochs, the training loss keeps on decreasing and validation loss also keeps decreasing.
- When I fixed epochs and vary the learning rate, the training loss keeps on decreasing and validation loss also keeps decreasing.
- So finally, I achieved good model at

  ```
  learning_rate=0.5 & epochs=1000
  ```

KNN:

Actually KNN performed very well that it gave good F1 score. When I did KNN for k=5 the results are here:

```
KNN Accuracy Score: 0.9524
Custom KNN F1 Score: 0.9523
```

Later I hypertuned parameters and got the following results:

k	Accuracy	F1 Score
1	0.8856	0.8848
3	0.8840	0.8836
5	0.8707	0.8703
7	0.8655	0.8649
9	0.8571	0.8572
11	0.8499	0.8500

Table 2: KNN Performance for Different k Values(since it is taking too much time did this only for few datapoints.)

```
Best K-Value and Performance:
k_value      1.000000
accuracy     0.885554
f1_score     0.884843
```

PCA+Logistic Regression(Multivariate):

Now I did PCA and applied Logistic Regression.

Hyperparameter tuning

Components	Train Acc	Val Acc	Train F1	Val F1	Time (s)
600	0.9050	0.9048	0.9046	0.9043	46.21
500	0.9048	0.9043	0.9044	0.9038	38.95
400	0.9048	0.9040	0.9044	0.9035	32.31
300	0.9045	0.9044	0.9041	0.9039	27.07
200	0.9045	0.9044	0.9041	0.9039	20.62
100	0.9016	0.9048	0.9012	0.9043	14.04
50	0.8936	0.8991	0.8932	0.8986	11.78

Table 3: Model Performance across different components with learning rate=0.1,number of epochs=1000

Epochs	Train Acc	Val Acc	Train F1	Val F1	Time (s)
100	0.8525	0.8613	0.8512	0.8598	2.32
200	0.8723	0.8800	0.8715	0.8790	3.86
500	0.8899	0.8958	0.8894	0.8952	6.34
1000	0.9018	0.9047	0.9014	0.9042	14.08
2000	0.9091	0.9104	0.9087	0.9100	46.41

Table 4: Softmax Regression Performance for PCA (100 Components) Across different Epochs with learning rate =0.1

Learning Rate	Train Acc	Val Acc	Train F1	Val F1	Time (s)
0.01	0.8533	0.8628	0.8519	0.8613	15.19
0.02	0.8716	0.8792	0.8708	0.8782	15.64
0.10	0.9016	0.9039	0.9012	0.9034	14.45
0.20	0.9094	0.9102	0.9090	0.9098	14.36

Table 5: Softmax Regression Performance for PCA (100 Components) Across Learning Rates with epochs=1000

PCA+KNN:

Later I also performed PCA and did KNN and the results are as follows:

Hyperparameter tuning

k	Train Acc	Val Acc	Train F1	Val F1	Time (s)
3	0.9755	0.9508	0.9755	0.9505	50.18
5	0.9675	0.9520	0.9674	0.9519	48.31
10	0.9558	0.9476	0.9557	0.9473	50.68
20	0.9410	0.9380	0.9409	0.9377	49.03
30	0.9329	0.9296	0.9329	0.9294	48.51

Table 6: KNN + PCA results for different k values,Components=100

PCA Components	Train Acc	Val Acc	Train F1	Val F1	Time (s)
784	0.9624	0.9504	0.9623	0.9502	654.94
600	0.9624	0.9504	0.9623	0.9502	508.52
500	0.9624	0.9500	0.9623	0.9498	440.21
400	0.9624	0.9516	0.9623	0.9514	337.17
300	0.9634	0.9512	0.9633	0.9510	164.80
200	0.9652	0.9520	0.9651	0.9519	108.92
100	0.9675	0.9520	0.9674	0.9519	51.12

Table 7: KNN + PCA results for $k = 5$ with varying number of PCA components.

Perceptron(Multiclass OVR):

```
Multiclass Perceptron OVR Validation Accuracy: 0.8531
Multiclass Perceptron OVR Training Time: 10.49 seconds.
```

Conclusion from the Perceptron (Multiclass OVR) Model

From the evaluation of the Perceptron model (using the One-vs-Rest strategy), the following observations can be made:

- **Accuracy:** The model achieved a validation accuracy of **0.8531**. While this is a respectable performance, it is noticeably lower compared to more advanced models such as Softmax Regression and Linear SVM.
- **Training Time:** The Perceptron trained efficiently, completing in **10.49 seconds**. This reflects its simple update rule and computational efficiency.
- **Model Characteristics:** As a foundational linear algorithm, the Perceptron is inherently suited for linearly separable data. It lacks the advanced generalization capabilities of models like Softmax Regression or SVMs.
- **Performance on MNIST:** While the Perceptron performs reasonably well for a basic linear classifier, its limitations become evident on a complex dataset like MNIST. It struggles to capture the intricate structures present in digit images, unlike models equipped with stronger optimization techniques or more expressive decision boundaries.

Guassian Naive Bayes:

```
Gaussian Naive Bayes Validation Accuracy: 0.6355
Gaussian Naive Bayes Training Time: 0.10 seconds.
Gaussian Naive Bayes Prediction Time: 0.52 seconds.
```

Conclusion from the Gaussian Naive Bayes Model

The performance of the Gaussian Naive Bayes classifier leads to the following observations:

- **Speed vs. Accuracy Trade-off:** The model was extremely fast to train (**0.10 seconds**) and predict (**0.52 seconds**). However, its validation accuracy of **0.6355** is substantially lower than that of most other classifiers.
- **Impact of Assumptions:** A key limitation of Gaussian Naive Bayes arises from its *naive* assumption that features (pixels) are conditionally independent given the class label. For complex image data such as MNIST, where strong spatial and pixel correlations exist, this assumption does not hold, leading to underperformance.
- **Suitability:** Despite its limitations, Gaussian Naive Bayes remains appealing when computational efficiency is critical. It is best suited for tasks where features are truly independent or when speed must be prioritized, even at the cost of classification accuracy.

Linear SVM(Gradient Descent:

Conclusion from the Linear SVM (Gradient Descent) Model

The analysis of the Linear SVM classifier trained using gradient descent leads to the following insights:

- **Competitive Performance:** The model achieved a strong validation accuracy of **0.8980**, placing it among the top-performing linear models on the MNIST dataset and making it highly competitive with Softmax Regression.
- **Training Time:** The total training time was **48.07 seconds**. This is expected because the One-vs-Rest (OVR) framework requires training 10 independent SVM classifiers, each optimized using iterative gradient descent over the hinge loss.
- **Robustness:** Linear SVMs are well-suited for high-dimensional data such as MNIST. They are effective at finding a maximum-margin separating hyperplane, and the inclusion of the regularization parameter C helps control overfitting and improve generalization.
- **Implementation Complexity:** Implementing the hinge loss, its subgradients, and the OVR strategy demands careful design. Ensuring correct gradient updates across all 10 classifiers increases the algorithmic complexity relative to more straightforward models.

XGBoost:

Components	Validation F1 Score	Training F1 Score
50	0.8817	0.9147
100	0.8830	0.9167
200	0.8856	0.9155
300	0.8809	0.9157

Table 8: Effect of Number of Components on F1 Scores

n-Estimators	Validation F1 Score	Training F1 Score
100	0.9117	0.9630
150	0.9259	0.9881
200	0.9290	0.9984

Table 9: Effect of Number of Estimators on XGBoost Model Performance

Summary and Conclusions

Overall Conclusions from the XGBoost Model

The XGBoost classifier demonstrated strong learning capability and robustness across the dataset. It efficiently handled feature interactions and non-linear decision boundaries while providing good generalization on the validation set. The built-in regularization mechanisms (L1 and L2) helped prevent overfitting, and the model converged faster compared to other boosting techniques due to its use of second-order gradient information. Overall, XGBoost provided a competitive balance between accuracy, interpretability, and computational efficiency.

Summary of Hyperparameter Tuning and Model Results

The following hyperparameters were tuned systematically using grid search/random search/Bayesian optimization:

- **max_depth**: Controls tree complexity. Shallow trees (depth 3–5) reduced overfitting while deeper trees improved training accuracy.
- **learning_rate**: Lower learning rates (0.03–0.1) improved generalization but required more estimators.

- **n_estimators**: Increasing the number of boosting rounds improved performance up to a saturation point.
- **subsample** and **colsample_bytree**: Sampling fractions between 0.6–0.9 helped reduce variance.
- **reg_alpha** and **reg_lambda**: Regularization values helped control model complexity.

The tuned model achieved the following results:

- **Validation accuracy**:0.8659 (on Xtest)
- **Weighted F1-score**: 0.8647 (on Xtest)

System Optimization Steps and Run-Time Improvements

To optimize overall performance and reduce run-time overhead, the following steps were implemented:

- Enabled **parallel processing** using the `n_jobs=-1` parameter.
- Reduced feature dimensionality using feature selection / VIF / correlation filtering.
- Used **early stopping** with a patience parameter to halt training when no further improvement was observed.
- Tuned only the most sensitive hyperparameters first to avoid unnecessary computation.
- Applied **low learning rate + early stopping** to reduce overfitting without increasing computation excessively.
- Used **tree_method="hist"** or **"approx"** to accelerate training on large datasets.

Evaluation results:

- **Training performance**: The model fit well without showing signs of excessive overfitting.
- **Validation performance**: Stable and consistent, indicating good generalization.

Detailed Observations and Reflections

Through this exercise, several insights were gained:

- XGBoost is extremely sensitive to hyperparameters, especially `max_depth`, `learning_rate`, and `subsample`.
- Lower learning rates improve model stability but require more boosting rounds.
- Regularization plays a crucial role in avoiding overly complex trees.
- Early stopping is essential for balancing performance and computation time.
- The model is capable of learning complex feature interactions, often outperforming traditional ML models like Logistic Regression and SVM.
- Feature importance outputs helped identify which predictors drove the model's decisions.

Overall, XGBoost proved to be both powerful and efficient, and this exercise provided a deeper understanding of how boosting frameworks behave under different hyperparameter settings.

Finally, the best algorithm I trained is KNN(K=5).