DA2401 - Machine Learning Lab
End Semester Project: MNIST Digit Classification

1 Models Used

The algorithms used in this project are:

- Softmax Regression / Multiclass Logistic Regression (with bagging via soft voting)
- XGBoost Classifier (boosting)
- Random Forest Classifier (bagging)
- k Nearest Neighbours (KNN)

These models are finally put together, after training, to form a weighted ensemble.

2 Individual Model Performances

The standalone accuracy and F1 score results of these models (i.e., when used individually to predict the digit classes), for their best tuned hyperparameters are:

Model	**Hyperparameters**
BaggingSoftmaxClassifier	`lr` = 0.05, `n_epochs` = 200, `mini_batch_size` = 256
XGBoostMultiClassifier	`n_estimators` = 30, `lr` = 0.1, `max_depth` = 3
RandomForest	`n_trees` = 50, `max_depth` = 8, `max_features` = 40
KNNClassifier	`k = 5`
Softmax Regression (no bagging)	`lr = 0.01`, `n_epochs` = 1000, `mini_batch_size` = 256

Model	**Acc. (Train)**	**Acc. (Val)**	**F1 (Train)**	**F1 (Val)**
BaggingSoftmaxClassifier	0.9172	0.9044	0.9161	0.9032
XGBoostMultiClassifier	0.9334	0.9087	0.9329	0.9081
RandomForest	0.9358	0.9023	0.9356	0.9015
KNNClassifier	1.0000	1.0000	0.9504	0.9503
Softmax Regression (no bagging)	0.9221	0.8991	0.9210	0.8999

As evident from the table, performing bootstrap aggregation with the softmax classifier, via soft voting (i.e, averaging of predicted probabilities across bootstrap samples), resulted in a significantly better trade-off between the training and validation F1 scores, while keeping the training times nearly the same; thus, I decided to go with the bagged softmax classifier.

3 Bias-Variance Trade-Off for Hyperparameter Tuning & Steps Taken to Limit Run Time

3.1 Softmax Regression

n_epochs	Bias^2	Variance
200	0.0173	0.0100
500	0.0172	0.0100
1000	0.0168	0.0098

From the above table, we may be tempted to go with `n_epochs` = 1000, but this results in the training time exceeding 5 minutes, and hence, is not a feasible choice. I decided to go with `n_epochs` = 200 (keeping the `learning_rate` and `mini_batch_size` constant), because jumping to `n_epochs` = 500 results in only a very small change in bias^2, without affecting variance, while increasing run time.

Note: I also tried other combinations for identifying the best hyperparameters, but the above table, and the ones that follow, do not cover all of them, for brevity, and due to the fact that the bias^2 and variance values either remained the same, or did not significantly vary with those sets of parameters. Additionally, training with those parameters resulted in a worse accuracy and F1 score, without any significant improvement in training time.

3.2 XGBoost Classifier

n_estimators	max_depth	Bias^2	Variance
30	3	0.0260	0.0004
40	3	0.0212	0.0005
30	4	0.0199	0.0005

Although `n_estimators` = 40 and `max_depth` = 3 seems to have the best trade-off, the difference in the training and validation F1 score was significantly greater than when `n_estimators` = 30 was used; also, `n_estimators`= 30 gave a higher F1 score. We do not use `n_estimators` = 30 and `max_depth` = 4, because the model then overfits (also taking 80 seconds longer to train).

Note: The differences in variance when variance = 0.0004 and variance = 0.005 is non-trivial, whereas the same difference in bias^2 is trivial in comparison.

3.3 Random Forest Classifier

max_depth	max_features	Bias^2	Variance
8	40	0.0239	0.0005
10	40	0.0193	0.0006
8	100	0.0219	0.0006

`max_depth` = 10 overfits, while also taking 100 seconds longer to train. Clearly, it is best to go with `max_depth` = 8, `max_features` = 40, due to the constraint in training time, and because the F1 score does not increase significantly when the number of features is 100.

3.4 KNN

k	Bias^2	Variance
1	0.0083	0.0035
5	0.0080	0.0018
10	0.0092	0.0012

From the above table, it is clear that k = 5 has the best trade-off between bias and variance; therefore, we train the `KNNClassifier` with k = 5.

Thus, these hyperparameters are optimal / near-optimal.

Summarily, the steps taken to optimize performance and run time are:

1. Limiting the number of trees built to 30-50
2. Limiting the depth of the `XGBoostMultiClassifier` to 3 and that of `RandomForest` to 8. This also optimizes performance by ensuring overfitting does not happen, by not letting the trees grow to greater depths
3. Vectorizing array operations using `numpy`'s built-in functions wherever possible
4. Ensuring code is modular and not repeated unnecessarily
5. Optimizing hyperparameters in such a manner that the training time does not blow up

3.5 Training Times

Model	**Training Time (s)**
BaggingSoftmaxClassifier	25.655
XGBoostMultiClassifier	118.4639
RandomForest	65.9721
KNNClassifier	0.3097

The total training time is nearly 220 seconds.

4 Other Algorithms Trained

In addition to the above four algorithms, I also trained a `k-Means` Clustering model, albeit with poor results, which are given below:

n_clusters	Acc. (Train)	Acc. (Val)	F1 (Train)	F1 (Val)
10	0.6929	0.6899	0.6912	0.6905
64	0.7462	0.7443	0.7477	0.7464
128	0.7932	0.7921	0.7928	0.7923
196	0.8011	0.8002	0.8013	0.7999

The metrics for `n_clusters` = 196 were the best I could achieve, which is very poor in comparison to the supervised models trained. Therefore, I decided to go with `KNN` over `k-Means` for the purpose of this project.

5 System Architecture

The final model on which the predictions are made combine the four algorithms (trained on their best hyperparameters) in the following manner:

Model	Weight
BaggingSoftmaxClassifier	35%
XGBoostMultiClassifier	10%
RandomForest	10%
KNNClassifier	45%

The predictions were initially made in two different ways:

- **Soft voting**
  For every digit, each model predicts the probability of the digit belonging to each of the 10 classes (i.e, $p(y = k \mid x),\ k = 0, 1, 2, \ldots, 9$); these probabilities are then weighted averaged (according to the weights from the above table) across all models
- **Hard voting**
  For every digit, its class is assigned to be the one it belongs to with the highest probability (i.e,$k = \arg\max_{k\in\{0,1,2,\ldots,9\}} p(y = k \mid x)$); a voting is then done, based on the model weights, and each digit is assigned the class with the highest vote. That is, say, if 3 models predict for one sample as follows:
  - Model A: predict class 1 (weight = 0.5)
  - Model B: predict class 1 (weight = 0.3)
  - Model C: predict class 0 (weight = 0.2) Then, the votes for the sample are: {1:0.8, 0: 0.2}; thus, the sample is assigned to class 1.

For the above weights:

Voting Method	**Acc. (Train)**	**Acc. (Val)**	**F1 Score (Train)**	**F1 Score (Val)**
Soft Voting	0.9645	0.9452	0.9643	0.9447
Hard Voting	0.9643	0.9642	0.94677	0.94672

It is observed that hard voting gives a higher accuracy and F1 score than soft voting for the same digit, and, because the difference between the training and validation metrics are smaller with hard voting, I decided to go with **hard voting** to make the class predictions.

6 Weight Tuning for Ensemble

Weights (Softmax, XGBoost, RF, KNN)	Acc. (Val)	F1 (Val)
(35, 10, 10, 45)	0.94638	0.94633
(40, 10, 10, 40)	0.9307	0.9299
(45, 10, 10, 35)	0.9240	0.9233

- I chose to give only 10% weightage to the tree-based models because their training-validation F1 scores differed significantly; although this difference is smallest in the case of softmax regression, I gave a slightly higher weight to `KNN`, because it had the best validation F1 score.
- Clearly, (35, 10, 10, 45) gives the best validation result, seeing as there is nearly a 2% difference in F1 score across all weight combinations.
- We see that by ensembling, the validation F1 score (0.94633) is higher than when each of the models is trained individually but `KNN` (but we can choose to ignore this, because 3 of the models give a validation F1 score of around 0.9, whereas even though `KNN`'s validation F1 score is 0.95, its training F1 score is 1.0, meaning the training-validation difference is higher here than in all the other models).
- Therefore, ensembling different models gives us better predictions overall.

7 Notes

- Performed `PCA`, with `n_components` = 100, for `BaggingSoftmaxClassifier`, because training on 784 features, for even a single model, exceeded the time limit, with only a marginal improvement in the evaluation metrics.
- No data preprocessing for `XGBoostMultiClassifier` and `RandomForest` (tree based methods do not require preprocessing).
- Normalized by dividing with 255 (i.e, maximum pixel value in the dataset) for `KNN`, because of potential problems with curse of dimensionality in higher dimensions.
  - Interestingly, standardising the data by mean centering and dividing by the variance resulted in a worse F1 score (0.913).