# END SEMESTER PROJECT: MNIST CLASSIFICATION

The models I have used for this project are as follows:

1) **XGBoostClassifier**: I have trained XGBoost classifier for each digit in the MNIST dataset. For every digit, we train this classifier to identify whether image belongs to this digit or not. So, it is a One vs All model. Each tree is constructed using a **greedy split-finding strategy**: for every node, the algorithm evaluates a subset of features (feature subsampling) and chooses the split that most improves the model. Regularization is applied using L2 penalties and depth control, making the model more stable.
2) **WeightedKNN:** K-nearest neighbors using weights as **(1/d)**, where d is the euclidean distance between target point and training sample. We take the K closest neighbors, add up the weights per class and choose the class with the highest combined weight. Since WeightedKNN is a non-linear model, it worked the best out of all other linear (logistic regression, SVM) and decision tree models (random_forest, adaboost, XGBoost).
3) **PCA:** Though PCA isn't a machine learning model, it plays an important role in preprocessing data and finding directions along which data has highest variance. This is a crucial linear projection I have used for both x_train and x_val before performing WeightedKNN on it.

I have not included logistic regression model since it gives a maximum accuracy of around **90%.** Testing with **RandomForest** is given further.

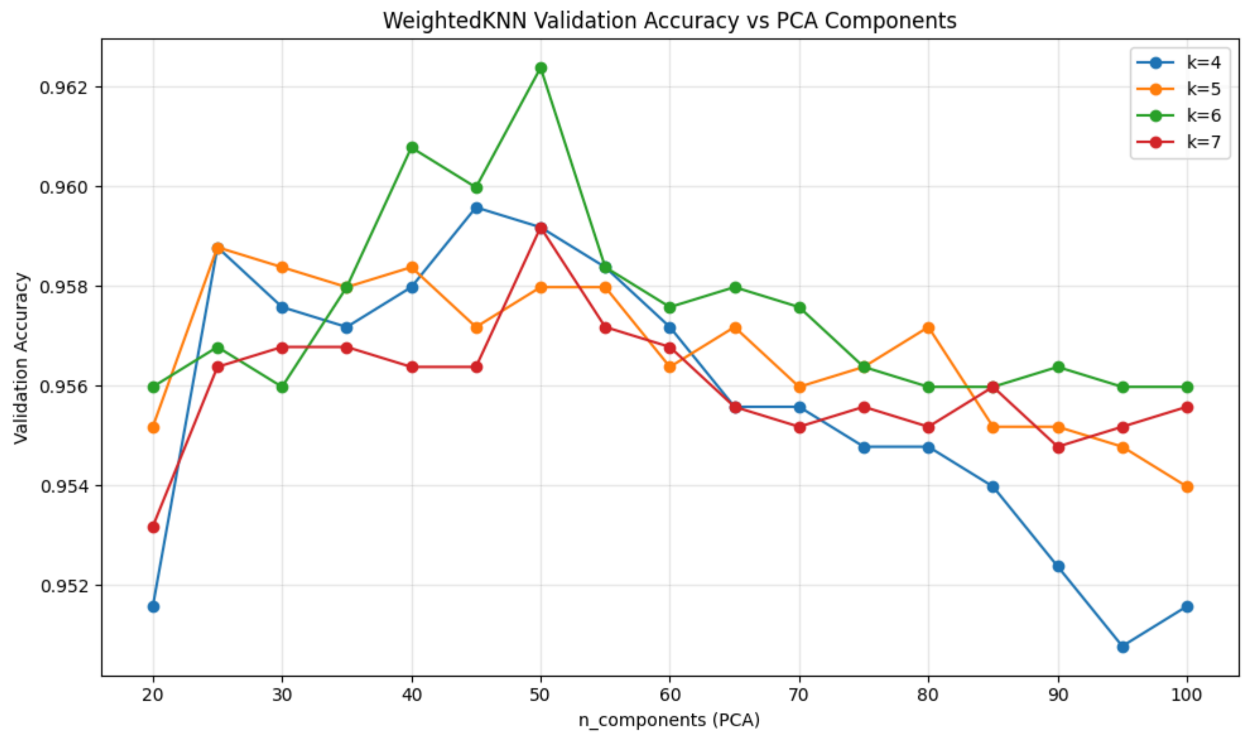
**System Architecture:** I have used a simple voting system for my ensemble model using XGBoost and WeightedKNN. I have used hyperparameter tuning to find out the best weights and parameters to use. I will show the hyperparameter tuning + end results using plots below.

**XGBoost hyperparameter tuning:** The parameters I have used for my ensemble model are mostly obtained through discrete tuning. I have considered different values of subsampling and learning rate only, since lambda seemed to not have a big effect on accuracy. Since subsampling is

```
n_estimators=50,
lamda=3,
learning_rate=0.3,
subsample_features=0.10
```

made randomly on features, the accuracy seemed to vary wildly from 94.8% to 95.2% on each run. Finally I decided to settle on the above hyperparameters. It takes around 150-160 seconds to fit on MNIST_train.csv dataset given.

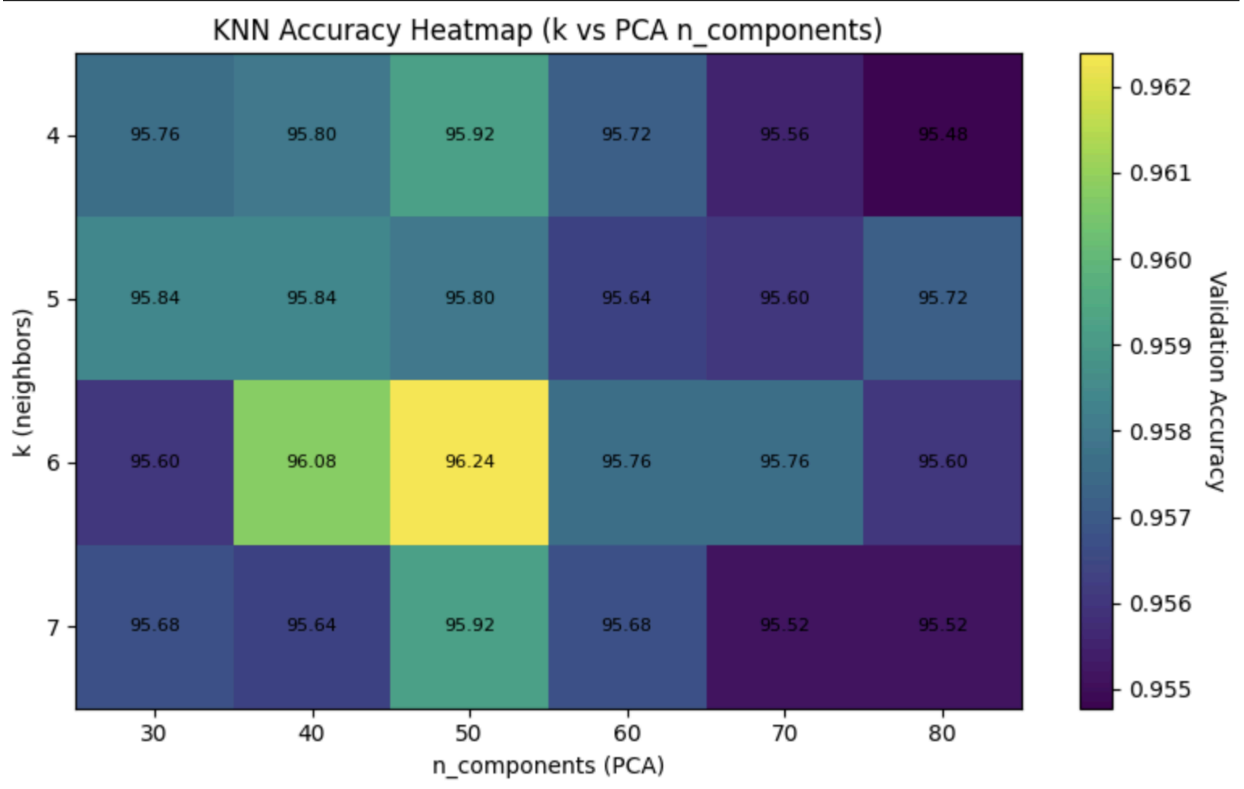
**WeightedKNN ensemble tuning:**



Above plot shows how validation accuracy varies as we change the number of PCA components we use and the value of k-nearest neighbors. It can be clearly observed that the best (k, n_components) pair for weightedKNN seems to be close to (6, 50). After some finer hyperparameter tuning, I found the optimum pair (6, 48), which gives 96.28% validation accuracy. However, I wanted to also try an ensemble of WeightedKNN models to decrease variance and improve bias. By introducing more models, we can reduce overfitting on just validation data, and also improve bias since few models can predict correctly the digits which other models might get confused doing. The ensemble I used for weightedKNN is of 9 models with combination of (k, n_components) as:

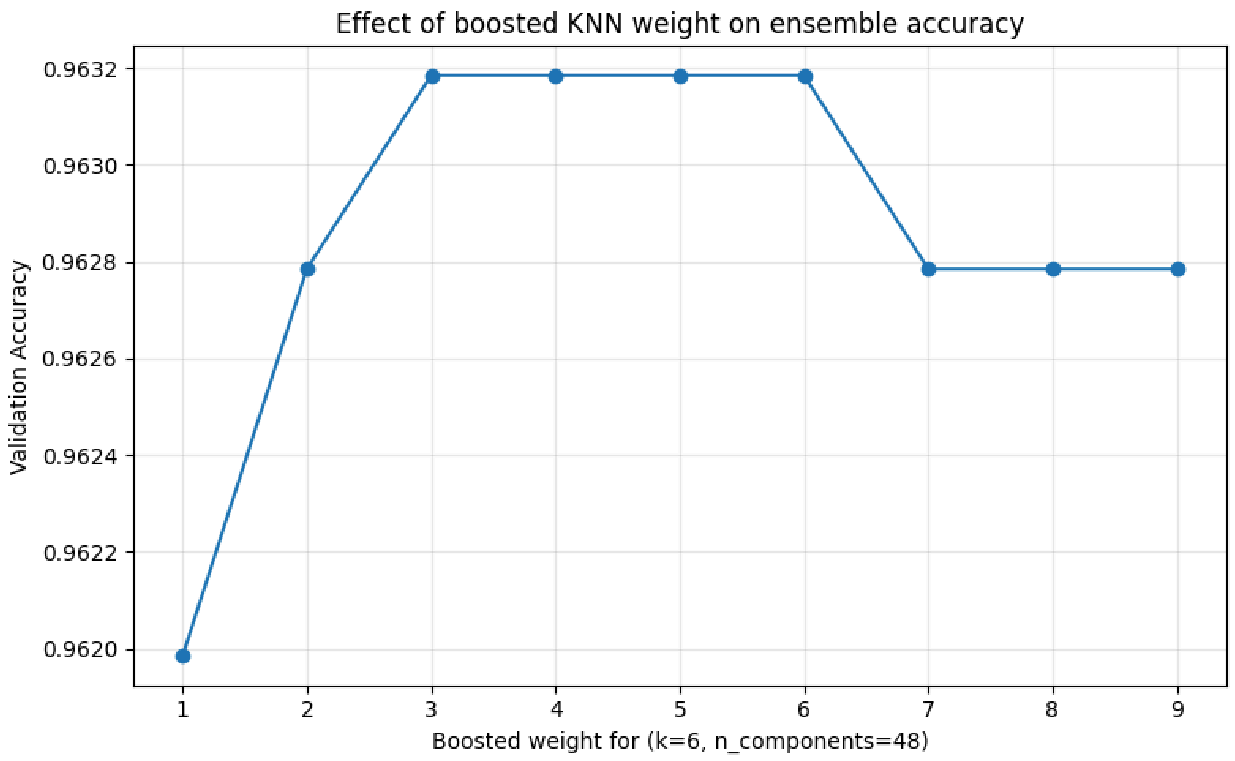
```
k_list = [4, 5, 6]        # K values I am using for WeightedKNN
ncomp_list = [40, 48, 55] # n_component values I am using for PCA in WeightedKNN
```

Since this range of hyperparaters gives consistently high validation accuracies. Here is a heatmap for better visualization:



| k (neighbors) | 30 | 40 | 50 | 60 | 70 | 80 |
|---|---|---|---|---|---|---|
| 4 | 95.76 | 95.80 | 95.92 | 95.72 | 95.56 | 95.48 |
| 5 | 95.84 | 95.84 | 95.80 | 95.64 | 95.60 | 95.72 |
| 6 | 95.60 | 96.08 | 96.24 | 95.76 | 95.76 | 95.60 |
| 7 | 95.68 | 95.64 | 95.92 | 95.68 | 95.52 | 95.52 |

Next, I wanted to vary the weight of the highest validation accuracy hyperparameter pair to see if we can increase accuracy even further. Here is a plot for it:



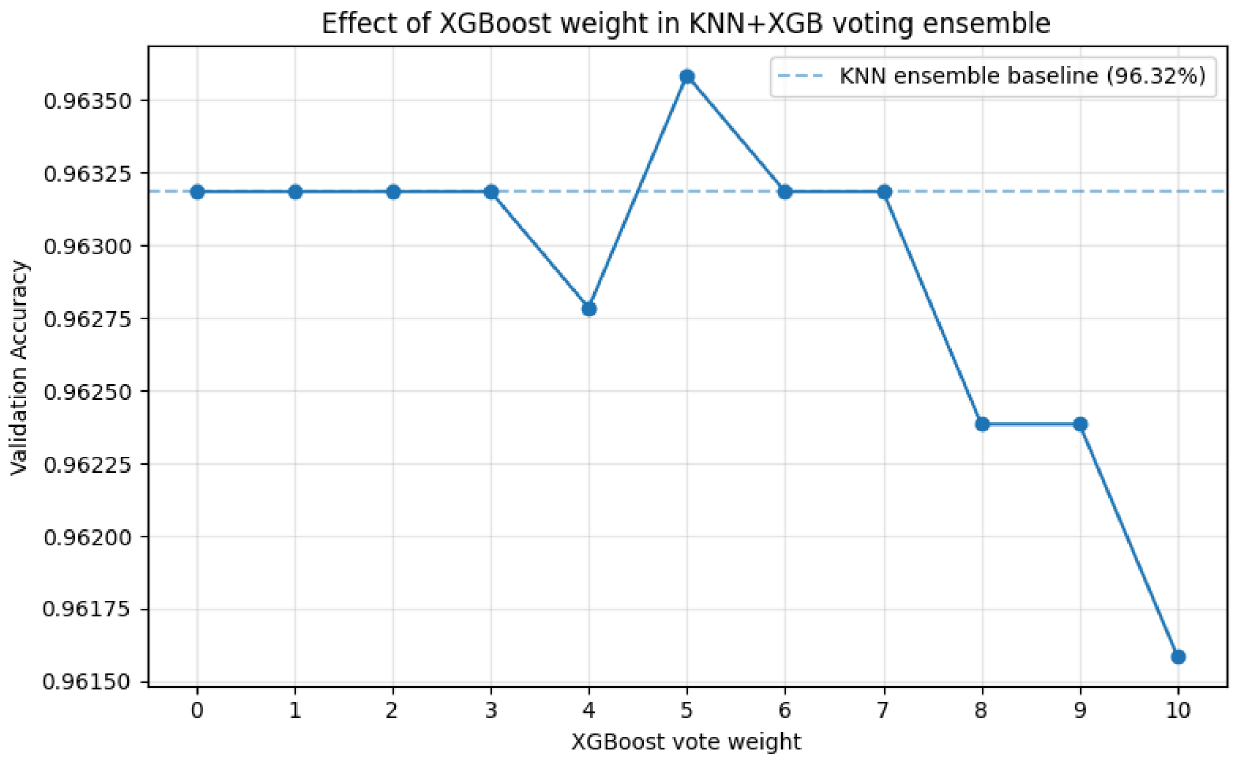
Turns out, having a higher weight for the best weightedKNN model increases validation accuracy by 0.12%.

The final WeightedKNN ensemble model has **96.32% validation accuracy.** Next, I tried to include XGBoost model into my ensemble.

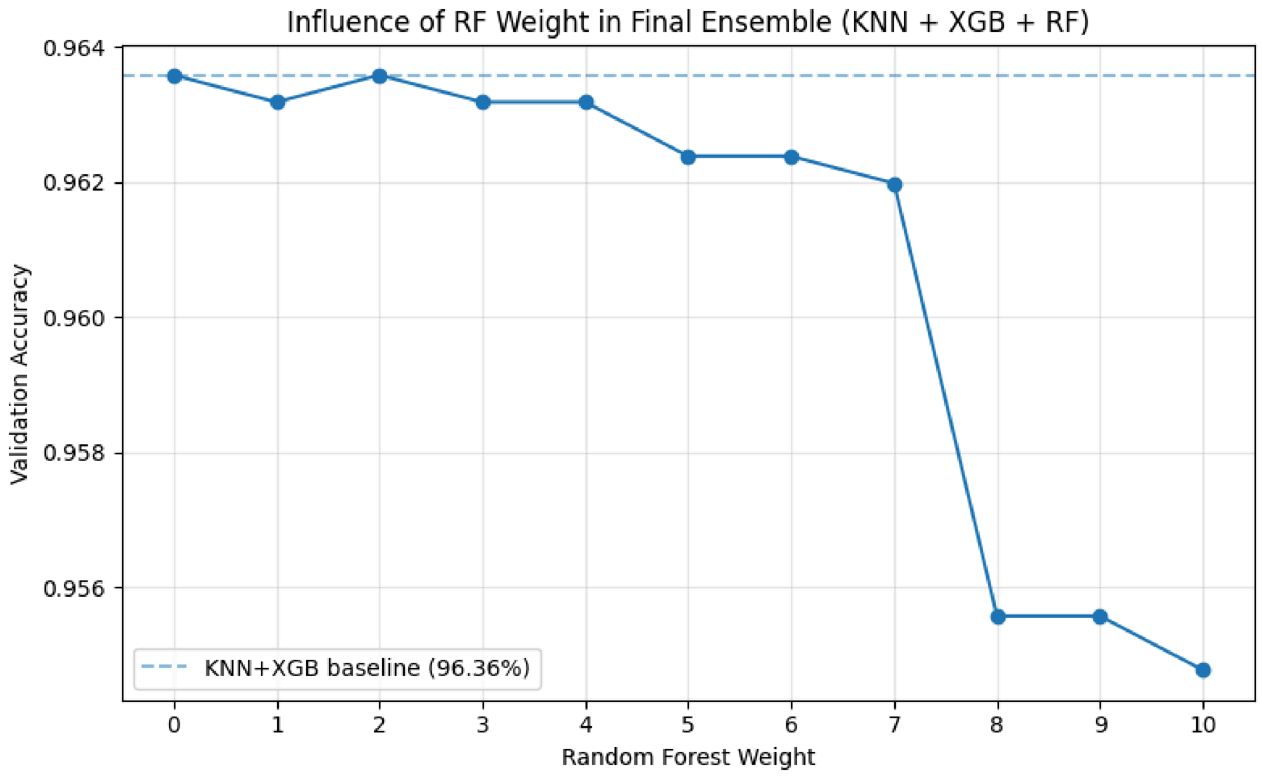
**WeightedKNN + XGBoost ensemble Hyperparameter tuning:**

I have similarly trained one vs rest XGBoost models for each digit and took the digit with maximum probability as my predicted digit. This gave me accuracies ranging from 94.8% to 95.2%. I wanted to check whether including it in my ensemble model would increase the validation accuracy further. Hence, I included it and tuned its weight too.
I observed that for XGBoost weight parameter = 5.0, validation accuracy increases to **96.36%.** However, one thing to note is that it implies that one particular data point/image was predicted correctly by this model which the KNN ensemble model wasn't able to. This doesn't amount to a huge improvement since it almost "overfit" on the validation dataset. However, including XGBoost in the ensemble model does decrease variance since

the model works differently compared to WeightedKNN, hence I included this too in the final model.



I also tried the same ensembling technique for RandomForestClassifier. However, it didn't improve validation accuracy:



**Optimization to reduce run time for WeightedKNN:**

To ensure that KNN runs efficiently on a dataset of 10,000+ samples, I replaced the standard Python loop–based distance computation with fully vectorized NumPy operations. Instead of separately computing Euclidean distances using for loops, This identity is used to facilitate matrix operations:

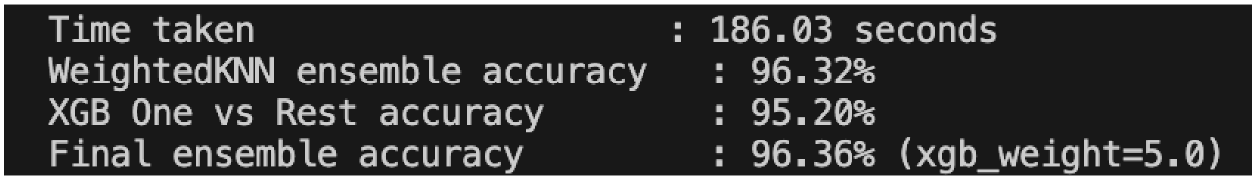
$$\|x-y\|^2 = \|x\|^2 + \|y\|^2 - 2x \cdot y$$

After computing distances, instead of sorting the full distance array—which would cost **O(NlogN)** per sample; I used **np.argpartition**, which finds the k nearest neighbors in **O(N) time**. PCA is applied beforehand to reduce dimensionality, lowering both the memory footprint and the cost of distance computation. Taken together, these optimizations reduce the runtime of KNN from minutes to seconds.

**Optimization to reduce run time for XGBoost Classifier:**

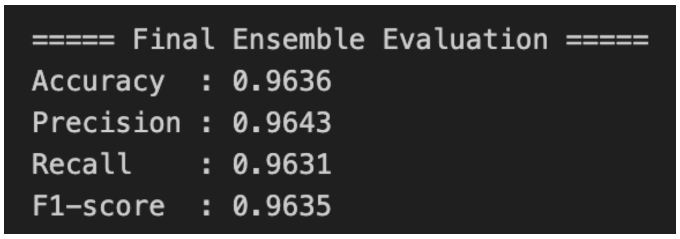
I optimized the scratch XGBoost implementation to stay within the runtime constraints by avoiding unnecessary feature usage and repeated computations. Only a **subset of features** is sampled at each split, which significantly reduces the search space without reducing much accuracy. This allows the model to train 10 OvR trees efficiently under the training time limit.

**Evaluation results on Validation dataset:**

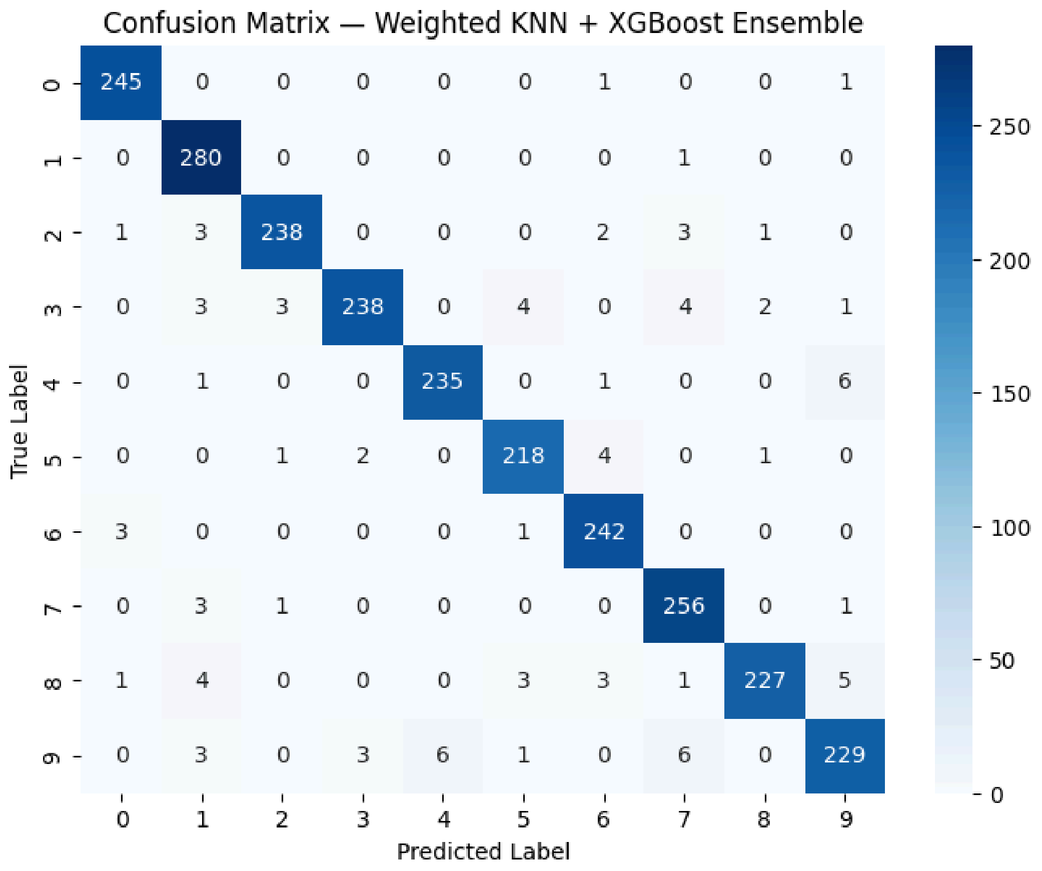


```
  Time taken                      : 186.03 seconds
  WeightedKNN ensemble accuracy     : 96.32%
  XGB One vs Rest accuracy          : 95.20%
  Final ensemble accuracy           : 96.36% (xgb_weight=5.0)
```

Total training + Validation time is just over 3 minutes for the ensemble model. I have listed the validation accuracies for the WeightedKNN ensemble model, XGB OvR model and finally, the WeightedKNN + XGboost ensemble model. After the hyperparameter tuning as given above, I was able to achieve **96.36% validation accuracy.** Final Confusion matrix and different statistics like F1score, precision and recall is as follows:

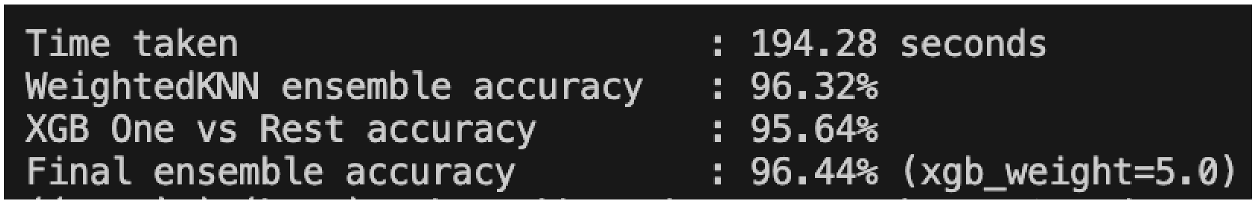


```
===== Final Ensemble Evaluation =====
Accuracy  : 0.9636
Precision : 0.9643
Recall    : 0.9631
F1-score  : 0.9635
```



Final F1_score of **96.35%** is very close to accuracy, and precision and recall scores are also within 0.05% of accuracy, indicating a well-balanced model.

The best run recorded (since XGBoost varies due to feature subsampling) is of validation accuracy: **96.44%**:



```
Time taken                        : 194.28 seconds
WeightedKNN ensemble accuracy     : 96.32%
XGB One vs Rest accuracy          : 95.64%
Final ensemble accuracy           : 96.44% (xgb_weight=5.0)
```

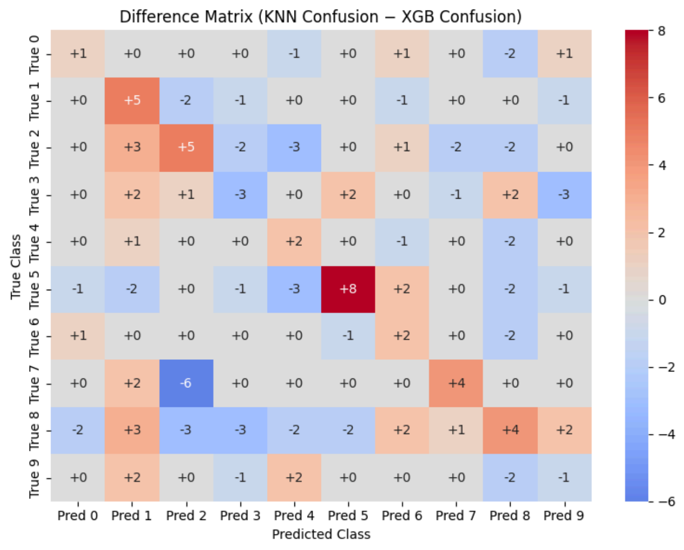
**Comment on Bias-Variance tradeoff for the models I used:**

**Weighted KNN** tends to have *low bias and high variance*, because it is a non-parametric model that memorizes the training set and adapts locally to the data. This is consistent with my results: KNN performed extremely well on MNIST after PCA compression but can overfit when the number of components is very high. Hence, the use of PCA before WeightedKNN.

XGBoost showed nearly **100% accuracy on the training data,** but its **validation accuracy plateaued around ~94–95%, indicating clear signs of overfitting.** This suggests that the model has **very low bias** (it fits the training distribution extremely well) but **high variance**.

**Final Thoughts and Observations:**

I observed that including XGBoost along with WeightedKNN ensemble didn't increase validation accuracy by much and so tried to check differences in the predictions made by the two models as a confusion difference matrix:



For the main-diagonal terms, positive or red regions are where KNN performs better than XGBoost, for off diagonal terms this is reversed. For the off-diagonal terms, the highest positive value where XGBoost does better than KNN is +3. Red regions are sparse and low in off-diagonal regions, explaining why adding XGBoost didnt really help the ensemble model. Also, the confusion matrix is mostly neutral with very few extreme values (bright red or blue), implying that both WeightedKNN and XGBoost are not complementary enough to cancel each other's errors and increase validation accuracy.

Attempts to improve beyond 96.5% were challenging.

I attempted several ensemble strategies includings **stacking ensemble with softmax.** It is a logistic regression model which takes predictions of different models like KNN, XGBoost, Logistic Regression, Random Forest, etc. as input and trains to give output probabilities for each digit based on these input features. Initially validation accuracy was **~96.0–96.1%**, but it plateaued. The issue was that training data needed to be split and K-fold CV needed to be done to train the logistic regression stack. But this reduced training data for the ensemble models, and K-fold CV took too much time.

Despite multiple ensemble strategies like stacking methods and weighted voting I have described above, validation accuracy doesn't seem to cross **96.5%.** This suggests that with the restrictions on training time and no non-linear preprocessing techniques, it is very difficult to reach 97% or more validation accuracy. (I was able to achieve **97.5% validation accuracy** using Sobel Gradients preprocessing instead of linear PCA on data).