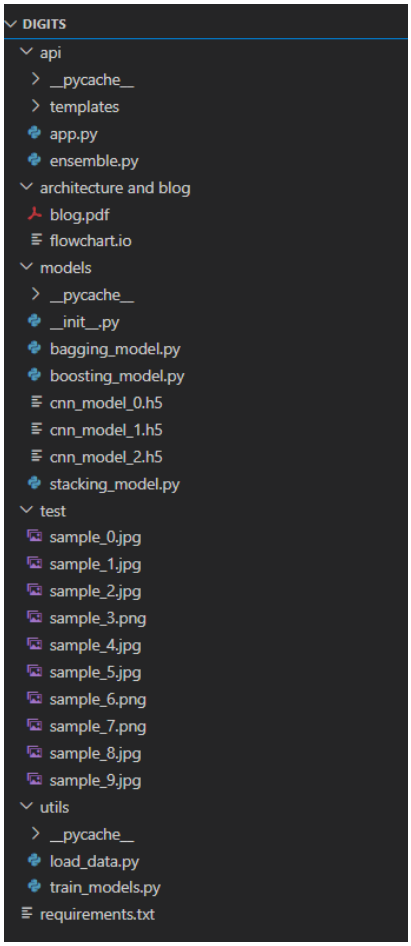
Multi-Model and Ensemble Deployment Strategies

Digits Dataset-MNIST

1. **Introduction**

   In this project, we set out to explore and implement advanced ensemble modelling techniques to improve the reliability and performance of predictive systems. Rather than relying on a single model, we combined multiple models using methods like bagging, boosting, and stacking. This allowed us to not only enhance prediction accuracy but also understand how to mitigate potential issues like bias and overfitting. We also tried A/B testing to evaluate the performance of different models and ensure that we could select the most effective ones in real-time. Additionally, we considered fallback mechanisms to handle any model failures smoothly and focused on achieving low latency for faster predictions. By combining these techniques, we gained a deeper understanding on how to design systems that are both robust and efficient, capable of providing more reliable predictions while maintaining performance.

2. Project structure



3. Building and Saving the Base Models

The function **load_mnist_data()** loads the MNIST dataset (handwritten digits), **normalizes** the pixel values (scaling them between 0 and 1), and **reshapes** the images to include a channel dimension (for grayscale). It then returns the **processed** training and test data.

utils > load_data.py > ...

```python
import tensorflow as tf

def load_mnist_data():
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
    x_train = x_train[..., None].astype('float32') / 255.0  # Add channel dimension
    x_test = x_test[..., None].astype('float32') / 255.0
    y_train = y_train.flatten()
    y_test = y_test.flatten()
    return (x_train, y_train), (x_test, y_test)

```

The **build_model()** function defines a simple Convolutional Neural Network (CNN) for digit classification. It consists of **two convolutional layers**, followed by **max-pooling layers**, a **flattening layer**, and **two fully connected layers (Dense)**. The model is compiled with the Adam optimizer and sparse categorical cross-entropy loss.
The **train_and_save_models()** function trains three separate models on the MNIST data, each for 10 epochs, and saves the trained models as **.h5 files** for later use.

utils > train_models.py > ...

```python
import tensorflow as tf
from load_data import load_mnist_data

def build_model():
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
        tf.keras.layers.MaxPooling2D((2,2)),
        tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
        tf.keras.layers.MaxPooling2D((2,2)),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')
    ])
    model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

def train_and_save_models():
    (x_train, y_train), (x_test, y_test) = load_mnist_data()

    for i in range(3):
        model = build_model()
        print(f"Training model {i+1}...")
        model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test, y_test))
        model.save(f'models/cnn_model_{i}.h5')

if __name__ == "__main__":
    train_and_save_models()

```

4. Defining the techniques

- **Bagging Model**

  The BaggingModel class uses the bagging technique, where multiple models are trained independently, and their predictions are averaged to make a final prediction. The **(__init__)** constructor accepts a list of model paths and loads each model using **tf.keras.models.load_model().**

  The predict method generates predictions for each model in the ensemble and averages them. It then uses **np.argmax** to select the most probable class based on the average prediction.

models > bagging_model.py > ...

```python
import numpy as np
import tensorflow as tf

class BaggingModel:
    def __init__(self, model_paths):
        self.models = [tf.keras.models.load_model(path) for path in model_paths]

    def predict(self, x):
        preds = [model.predict(x) for model in self.models]
        preds = np.array(preds)
        avg_preds = np.mean(preds, axis=0)
        final_preds = np.argmax(avg_preds, axis=1)
        return final_preds

```

- **Boosting Model**
  The BoostingModel class implements the boosting technique, where multiple models are trained sequentially, and their predictions are weighted based on their performance.
  Similar to the BaggingModel, models are loaded from the provided paths. Additionally, weights for each model can be specified to give more importance to better-performing models.
  The predict method calculates the weighted sum of predictions from each model, using the provided weights. It then selects the class with the highest weighted sum.

models > boosting_model.py > ...

```python
import numpy as np
import tensorflow as tf

class BoostingModel:
    def __init__(self, model_paths, weights=None):
        self.models = [tf.keras.models.load_model(path) for path in model_paths]
        self.weights = weights if weights else [1.0/len(self.models)] * len(self.models)

    def predict(self, x):
        preds = [model.predict(x) * weight for model, weight in zip(self.models, self.weights)]
        preds = np.array(preds)
        weighted_sum = np.sum(preds, axis=0)
        final_preds = np.argmax(weighted_sum, axis=1)
        return final_preds

```

- **Stacking Model**
  The StackingModel class implements the stacking technique, where the predictions from multiple models are used as features to train a meta-model (a logistic regression model in this case).
  Models are loaded, and a meta-model (Logistic Regression) is initialized.
  The **fit_meta_model** method generates predictions from each model and uses them as input features to train the meta-model.
  The predict method generates predictions from each model, concatenates them, and uses the meta-model to make the final prediction.

models > stacking_model.py > ...

```python
import numpy as np
import tensorflow as tf
from sklearn.linear_model import LogisticRegression

class StackingModel:
    def __init__(self, model_paths):
        self.models = [tf.keras.models.load_model(path) for path in model_paths]
        self.meta_model = LogisticRegression(max_iter=1000)

    def fit_meta_model(self, x_val, y_val):
        features = [model.predict(x_val) for model in self.models]
        features = np.concatenate(features, axis=1)
        self.meta_model.fit(features, y_val)

    def predict(self, x):
        features = [model.predict(x) for model in self.models]
        features = np.concatenate(features, axis=1)
        final_preds = self.meta_model.predict(features)
        return final_preds

```

5. Designing the Ensemble Strategies

This code defines a class **EnsembleAggregator** that aggregates predictions from multiple ensemble techniques (Bagging, Boosting, and Stacking) using pre-trained models.

In the **__init__** method, the class is initialized with paths to three pre-trained CNN models. These models will be used by the Bagging, Boosting, and Stacking models. Each ensemble technique is instantiated as an object using the respective model class (BaggingModel, BoostingModel, StackingModel).

api > ensemble.py > EnsembleAggregator > predict

```python
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from models.bagging_model import BaggingModel
from models.boosting_model import BoostingModel
from models.stacking_model import StackingModel
import random
import numpy as np

class EnsembleAggregator:
    def __init__(self):
        self.model_paths = ['models/cnn_model_0.h5', 'models/cnn_model_1.h5', 'models/cnn_model_2.h5']
        self.bagging = BaggingModel(self.model_paths)
        self.boosting = BoostingModel(self.model_paths)
        self.stacking = StackingModel(self.model_paths)
```

This method makes a prediction using one of the ensemble techniques (default is Bagging). It accepts input data (x) and a method argument to choose the ensemble technique. If an invalid method is provided or an error occurs during prediction, it falls back to using Bagging as a default. This is with the help of **fallback Mechanism.**

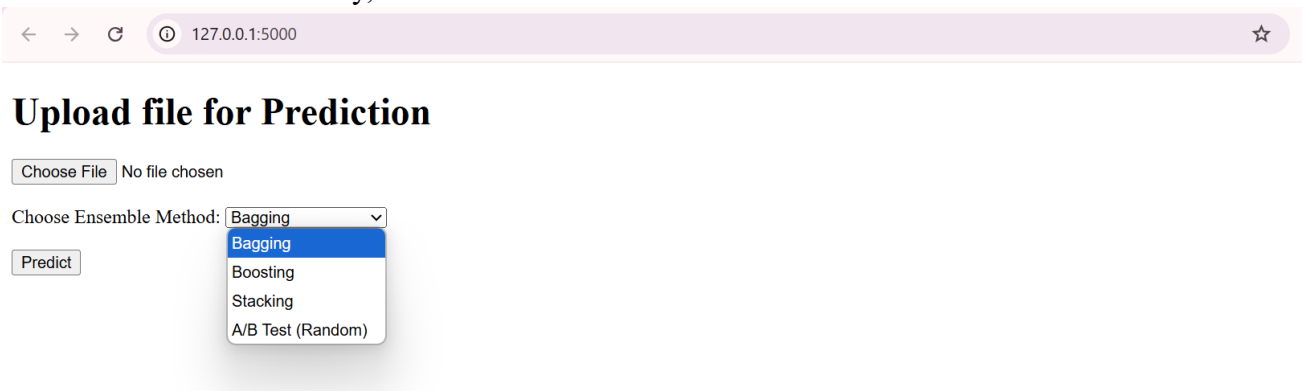
```python
def predict(self, x, method='bagging'):
    try:
        if method == 'bagging':
            return self.bagging.predict(x)
        elif method == 'boosting':
            return self.boosting.predict(x)
        elif method == 'stacking':
            return self.stacking.predict(x)
        else:
            raise ValueError("Invalid method.")
    except Exception as e:
        # fallback to bagging if something fails
        print(f"Fallback due to error: {e}")
        return self.bagging.predict(x)
```

The **a_b_test** method randomly selects one of the three ensemble methods (Bagging or Boosting) to make a prediction. This simulates A/B testing, where different methods are evaluated to see which performs best for the given input data.

```python
def a_b_test(self, x):
    method = random.choice(['bagging', 'boosting', 'stacking'])
    print(f"A/B Testing with method: {method}")
    return self.predict(x, method)
```

6. Building the Flask Web Application

After the backend was ready, we created a Flask web interface.



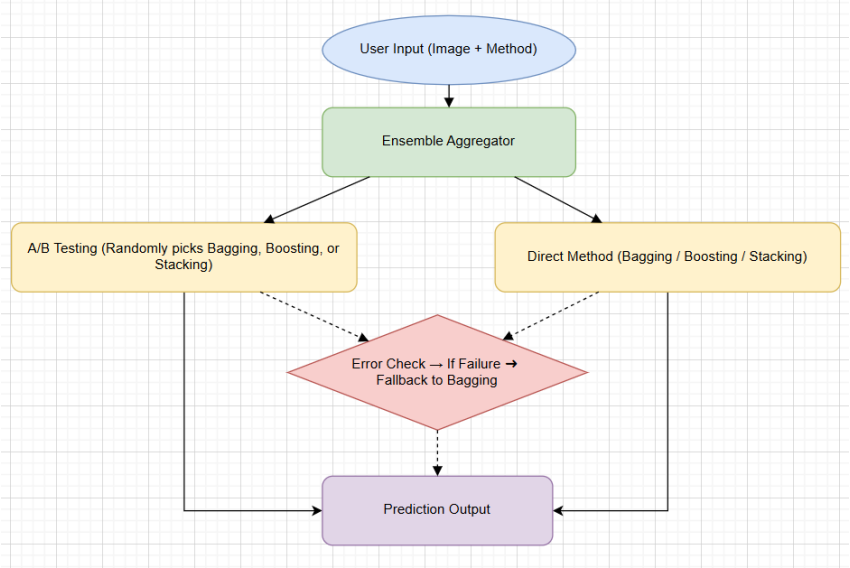
- User can upload an image.
- Select the desired method: Bagging, Boosting, Stacking, or A/B Testing.
- Get the final prediction displayed directly on the browser.

7. Implementing Smart Backend Control

We made sure that:

- Backend routes the request correctly based on user choice.
- Correct ensemble logic runs in the background without the user needing to know the details.
- Fallback happens silently without user disruption.
- A/B Testing behaves randomly but predictably (for proper experimentation).

8. Visualising the workflow



9. Key Features Achieved

- Can work with more than one model
- Supports different ways of combining models (like bagging, boosting, stacking)
- Easy-to-use web interface for users
- Errors in methods has a backup plan if something goes wrong
- Can randomly try different methods to test which one works better

10.Challenges

- Ensuring all CNN models had consistent input shapes and output formats was crucial, especially when combining predictions in ensemble methods.
- Since the MNIST dataset consists of images with digits written with while ink on black paper, it was hard to get proper predictions with normal handwritten digits.
- Implementing a fallback mechanism was important because errors in one model (especially in stacking) could break the pipeline without proper handling.
- Managing randomized method selection while keeping results interpretable and traceable for comparison added complexity.
- Training multiple deep learning models is compute-intensive—doing this with limited hardware or time was a challenge.