Machine Learning Classification Project : Weather Classification

Team Name : Code_Wars
Dhairya Gupta - IMT2021005
Sheikh Muteeb - IMT2021008
Suyash Ajit Chavan - IMT2021048

December 17, 2023

1 Introduction

In this report, **Code_wars** tackles the task of building a classification model to predict whether average humidity will surpass 50% the next day. Utilizing a diverse meteorological dataset, our focus is on creating a robust model that precisely classifies instances where humidity crosses this critical threshold. The dataset, ranging from temperature metrics to atmospheric conditions, serves as a valuable resource for training and evaluating our model. As we navigate the intricacies of the data, our goal is to uncover meaningful patterns essential for accurate predictions.

2 Support Vector Machines :

We used two types of SVM from Sklearn that are **SVC**(Support Vector Classification) and **NuSVC**
The results for SVM we got even after hyperparamter tuning were sub optimal as comparared to the ones we trained as we go further into the report. Major drawback while using SVM we encountered is that it took too much time and some time more than that we were able to run on a machine.

2.1 Hyper Parameters

The hyperparamters we used to tune the model were:

1. kernel : Different types of kernel like rbf, polynomial, linear etc.
2. degree : Used in degree of polynomial kernel if kernel is chosen to be poly(ignored by other kernels).
3. gamma(kernel coefficient) : value can be set to 'auto', 'scale' or 'float'

4. decision_function_shape : can be changed between 'ovr' and 'ovo'

5. nu(for NuSVC) : An upper bound on the fraction of margin errors.

So, we used SVM to train a model with a f1 score of 0.65 but this was suboptimal to other scores we achieved. The hypertuning we did for SVM was more through brute force because each try took a substantial amount of time ($\geq$ 1hr) for us to use grid search.

3 Ensemble Models - Part 1 :

3.1 Ensemble Models with XGBoost, LGBM, Random Forest, and CatBoost

To enhance predictive performance and capture diverse patterns in the data, ensemble models were employed using a combination of XGBoost, LightGBM (LGBM), Random Forest, and CatBoost classifiers. Three ensemble techniques, namely **voting**, **stacking**, and **blending**, were explored to leverage the strengths of individual models and achieve a more robust and accurate prediction.

3.1.1 Voting Classifier

A **voting classifier** was implemented to combine the predictions from multiple base models. Different combinations of XGBoost, LGBM, Random Forest, and CatBoost were experimented with to achieve a diversified ensemble.

3.1.2 Stacking Classifier

The **stacking classifier** involved training a meta-model to learn how to best combine the predictions of the base models. The diverse set of base models, including XGBoost, LGBM, Random Forest, and CatBoost, contributed to the meta-model's ability to generalize well.

3.1.3 Blending Classifier

In the **blending classifier**, predictions from multiple base models, such as XGBoost, LGBM, Random Forest, and CatBoost, were blended using weighted averaging. The weights were optimized to maximize predictive accuracy on the validation set.

This ensemble approach aimed to harness the complementary strengths of different algorithms, providing a more robust and accurate solution for the binary classification task at hand. The hyperparameter tuning process was conducted to find the optimal combination of models and their weights, ensuring an effective integration of individual model predictions.

3.2 Optimizing Threshold for Binary Classification

Implemented a binary search algorithm to find the optimal threshold for converting predicted probabilities from ensemble models into binary class labels. The process involved iterative evaluation of performance metrics (e.g., F1 score) at different thresholds, aiming to strike a balance between precision and recall. The selected optimal threshold enhances the interpretability and performance of the ensemble models in the binary classification task.

4 Ensemble Models - Part 2 :

Stacking Ensemble Models

1. Stacking (Random Forest, Gradient Boosting, XGBoost) - Without Date Column:

- **Base Models:**
  - Random Forest (RF)
  - Gradient Boosting (GB)
  - XGBoost (XGB)
- **Stacking Model:**
  - Final Estimator: Random Forest
- **Accuracy:** 0.9184

2. Stacking (Random Forest, Gradient Boosting, XGBoost) - With Date Column:

- **Base Models:**
  - Random Forest (RF)
  - Gradient Boosting (GB)
  - XGBoost (XGB)
- **Stacking Model:**
  - Final Estimator: Random Forest
- **Accuracy:** 0.9128

3. Stacking (Random Forest, Gradient Boosting, XGBoost) - With XGBoost Hyperparameters Tuned:

- **Base Models:**
  - Random Forest (RF)
  - Gradient Boosting (GB)
  - XGBoost (XGB) - Hyperparameters Tuned
- **Stacking Model:**
  - Final Estimator: Random Forest
- **Accuracy:** 0.9174

4. Stacking (Logistic Regression, Decision Tree, KNN):

- **Base Models:**
  - Logistic Regression (LR)
  - Decision Tree (DT)
  - K-Nearest Neighbors (KNN)
- **Stacking Model:**
  - Final Estimator: Logistic Regression
- **Accuracy:** 0.8732

5. Stacking (Random Forest, Decision Tree, Gradient Boost) - With TPOT Genetic Algorithm:

- **Base Models:**
  - Random Forest (RF)
  - Decision Tree (DT)
  - Gradient Boosting (GB)
- **Stacking Model:**
  - Final Estimator: Logistic Regression (TPOT Optimized)
- **F1 Score (Kaggle):** 0.6324

Voting Ensemble Models

1. Voting (Random Forest, Gradient Boosting, XGBoost) - With XGBoost Hyperparameters Tuned:

- **Individual Models:**
  - Random Forest (RF)
  - Gradient Boosting (GB)
  - XGBoost (XGB) - Hyperparameters Tuned
- **Voting Model:**
  - Hard Voting
- **Accuracy:** 0.9141

2. Voting (Logistic Regression, Decision Tree, KNN) - With Hyperparameters Tuned (RandomizedSearchCV):

- **Individual Models:**
  - Logistic Regression (LR)
  - Decision Tree (DT)
  - K-Nearest Neighbors (KNN)
- **Voting Model:**
  - Hard Voting
- **F1 Score (Kaggle):** 0.6333

3. Voting (Random Forest, Decision Tree, Gradient Boost, AdaBoost, Logistic Regression):

- **Individual Models:**
  - Random Forest (RF)
  - Decision Tree (DT)
  - Gradient Boosting (GB)
  - AdaBoost (AB)
  - Logistic Regression (LR)
- **Voting Model:**
  - Hard Voting
- **Individual Model Accuracies:**
  - Random Forest Accuracy: 0.8793

- Decision Tree Accuracy: 0.8054
- Logistic Regression Accuracy: 0.7885
- AdaBoost Accuracy: 0.8020
- Gradient Boosting Accuracy: 0.8193
- Voting Classifier Accuracy: 0.8320

5 Neural Networks :

5.1 Introduction

This report presents a neural network architecture designed for binary classification tasks. The implemented code leverages the TensorFlow and Keras libraries, offering a flexible and customizable solution for various datasets.

5.2 Initialization

- The class initializes with the dimensions of the input and output layers.

5.3 Model Building

- The `build_model` method constructs the neural network architecture based on hyperparameters provided through the Keras Tuner.
- It dynamically creates hidden layers with specified units, activations, initializers, and regularizers.
- Optional Batch Normalization and Dropout layers are added based on boolean hyperparameters.
- The output layer uses a sigmoid activation function for binary classification.
- The model is compiled with binary crossentropy loss, the chosen optimizer, and accuracy as the metric.

5.4 Hyperparameter Tuning

- The `tune_hyperparameters` method utilizes Keras Tuner's RandomSearch to explore hyperparameter combinations.
- The search objective is set to maximize validation accuracy.
- The hyperparameters explored include the number of hidden layers, units per layer, activations, initializers, regularizers, Batch Normalization, Dropout rates, optimizer, learning rate, number of epochs, and batch size.
- The possible values for each hyperparameter are as follows:

5.4.1 Number of Hidden Layers:

1, 2, 3

5.4.2 Number of Units in each Hidden Layer:

[64, 128, 256]

5.4.3 Activation Functions:

['relu', 'tanh', 'sigmoid']

5.4.4 Kernel Initializer:

['glorot_uniform', 'he_normal', 'random_normal']

5.4.5 Kernel Regularizer:

[None, 'l1', 'l2']

5.4.6 Batch Normalization:

[True, False]

5.4.7 Dropout Rate:

[0.2, 0.3, 0.4]

5.4.8 Optimizer:

['adam', 'rmsprop', 'sgd']

5.4.9 Learning Rate:

[0.001, 0.0001, 0.00001]

5.4.10 Number of Epochs:

[5, 10, 15]

5.4.11 Batch Size:

[32, 64, 128]

5.5 Training

- The model is trained using the hyperparameters obtained from the tuning process.
- Training involves fitting the model to the provided training data for a specified number of epochs and batch size.

5.6 Keras Tuner for Hyperparameter Tuning

The hyperparameter tuning process was facilitated by the Keras Tuner library, which provides a powerful and efficient way to search through hyperparameter space and identify optimal configurations for neural network models.

5.6.1 Hyperparameters Explored

The hyperparameters explored during the tuning process, as determined by Keras Tuner, include:

- **Number of Hidden Layers:** 2
- **Number of Units in each Hidden Layer:** 128, 64
- **Activation Functions:** 'relu' for both layers
- **Kernel Initializer (Layer 0):** 'he_normal'
- **Kernel Initializer (Layer 1):** 'glorot_uniform'
- **Kernel Regularizer (Layer 0):** None
- **Kernel Regularizer (Layer 1):** 'l2'
- **Batch Normalization (Layer 0):** True
- **Batch Normalization (Layer 1):** False
- **Dropout Rate (Layer 0):** 0.3
- **Dropout Rate (Layer 1):** 0.2
- **Optimizer:** 'adam'
- **Learning Rate:** 0.0001

5.6.2 Best Configuration

The Keras Tuner identified the combination of these hyperparameters that resulted in the highest validation accuracy, and this configuration was subsequently used for training the neural network model.

5.6.3 Conclusion

The use of Keras Tuner significantly streamlines the hyperparameter tuning process, allowing for efficient exploration of the hyperparameter space and ultimately leading to the discovery of configurations that enhance model performance.

5.7 Wide and Deep Neural Architecture

5.7.1 Architecture Overview

The neural network architecture is designed with two main branches: a **wide branch** and a **deep branch**.

Wide Branch - A single dense layer captures simple patterns.

Deep Branch - Multiple dense layers extract complex features.

5.7.2 Advantages

1. **Pattern Flexibility:**
    - Wide branch captures simple patterns.
    - Deep branch extracts complex features.
2. **Enhanced Performance:**
    - Balances memorization and generalization.
    - Performs well on diverse datasets.
3. **Versatility:**
    - Suitable for data with varied complexity.
    - Combines strengths of both architectures.

In summary, the wide and deep architecture blends simplicity and complexity, optimizing model performance across a range of datasets.

5.8 Multilayered Perceptron (MLP)

The MLP, a foundational neural network, excels in learning complex patterns through its layered architecture. Trained via gradient descent and backpropagation, key hyperparameters include layer configuration and activation functions. While versatile, MLP faces challenges like overfitting and sensitivity to hyperparameters.

5.8.1 Advantage

- Capacity for Complex Patterns: MLP demonstrates a strong ability to learn intricate patterns in data, making it versatile for various tasks.

5.8.2 Disadvantage

- Overfitting Sensitivity: MLP, especially with large architectures, is prone to overfitting, requiring careful regularization.