0 Introduction

This is the official documentation of the code repository of the paper: Patrick Killeen, Iluju Kiringa, and Tet Yeap "UAV Imagery-Based Yield Prediction Using Federated Learning and Model Sharing for Smart Farming Applications".

It includes crop yield prediction via machine learning, deep learning, and federated learning regression models using Python and TensorFlow.

1 Requirements

This is my experiment environment.

- 64-bit Linux (Ubuntu 20.04) desktop with a NVIDIA GeForce RTX 3060 graphics card
- Python 3.8.19
- TensorFlow version 2.12.0
- CUDA version 11.8
- cuDNN version 8.6
- sklearn version 1.3.2
- numpy version 1.24.3
- pandas version 2.0.3
- geopandas version 0.13.2

2 Data Pre-processing

We perform some data pre-processing before feeding the datasets to the models. In this section we briefly explain some of the main logic used for data pre-processing, namely, feature selection via partial least squares regression (PLSR).

Note that I fused the imagery to the yield dataset using the project here https://github.com/patkilleen/geospatial

2.2 Feature Selection

Once a dataset was pre-processed and contains no missing data, feature selection using PLSR is used. This logic is implemented in the plsr.py module, and can be run using the 'runFeatureSelection' function. We explain the API to 'runFeatureSelection''below:

- inPath (string): input path to the dataset to apply feature selection to
- outResPath (string): output path to the result file that will contain PLSR model performance and the selected features, where each feature will have a column in the file such that a 0 indicates the feature was not selected and 1 indicates the feature was selected
- outCoeffPath (string): output path to the file that will contain PLSR coefficients. This is useful for feature important analysis (larger coefficients in absolute value have more impact on N2O). A column is found for each feature and has the coefficient of that feature

- trials (int): number of trials random search is conducted, that is, the number of different hyperparameter sets that are used for hyperparameter tuning when identifying the best hyperparameter choice to use for PLSR
- nFolds (int): number of folds used by cross-validation
- nSamplesPerClient (int): number of random samples to take from the input dataset (-1 includes all the samples)

An example of this feature selection can be run by running 'runFeatureSelection.sh' or:

```
python plsr.py --dataset input/datasets/mono-temporal-DoY201.csv
```

, where the result files will be found in 'output/feature-selection/'

3 Experiment Description

The are 2 types of experiments, cross-validation experiments that use a single dataset and cross-validation to evaluate models, and federated learning experiments that simulated multiple client each running local cross-validation experiments. The scripts take as input a CSV configuration file path and an output directory path. The configuration file controls what types of experiments are run (e.g., the learning model). The output directory is the directory where the output result files will be stored, such for each experiment, directories are created in the output directory and are named after the current time.

The main function is the 'experimenter.py' script for the single dataset cross-validation experiments. The main function is the 'FLExperimenter.py' script for the federated learning experiments. Both scripts take 2 input arguments:

- --inFile: (string) the configuration file path to control the experiments
- --outDirectory: (string) the directory where output files will be stored

3.1 Input file formats

The files input to the logic in this project are as follows

- **configuration file**: The configuration file specifies a batch of experiments to run such that each row specifies the type of experiment to run along with various parameters to tune the experiment. Note that there is a memory management issue hidden somewhere in the code, and if the configuration file contains too many experiments (or too many iterations/folds/execution-trials), TensorFlow may run out of memory. A work around is to separated a configuration file into many single entry configuration files and run all the separated configuration files via a shell script with multiple calls to running 'experimenter.py' (the main file)
- **Input dataset files**: specified by the configuration file, an input dataset file is in CSV format, where the 'clientID' column indicates the client id the sample belongs to (i.e., the field), the 'X' and 'Y' column are location features indicating the UTM easting and northing coordinate, respectively, of the samples. Columns with the naming convention 'MS_<spectral-channel>_<aggregation operation>_<temporalness of feature>' are imagery feature columns where 'spectral-channel' indicates the spectral channel (e.g., NDVI or red), 'aggregation

operation' indicates what type of filter was used to derive the feature (mean, max, or min reflectance in a 4.5 m radius around the yield point), and 'temporalness of feature' indicates whether the feature was from day of year (DoY) 201 imagery, DoY 216 imagery, or was a temporal delta feature (DoY 201 reflectance – DoY 216 reflectance). The last column is the yield data in bu/ac (bushels per acre). Furthermore, the geographic distance between samples should not be smaller than the spatial resolution specified in the configuration file (we discuss this configuration file entry later in the documentation)

- **A selected feature file**: specified by the configuration file, this input CSV file specifies which features are selected to be part of the machine learning experiments. The feature columns in the input dataset file should all be present here. The ClientID, X, and Y column, and target variable column should not be in this file. Only a single row (excluding the header) should be found in this file, where a 0 in a cell means the feature is not included and a 1 means the feature is included. This design allows for a single dataset file to exist, and many different selected feature files can be used to try out different features without having to change the input dataset.

3.2 Cross-validation

We explain cross-validation experiments in this section.

The cross-validation logic performs nested layered K-J-fold block cross-validation on a single input dataset: the dataset is split into client datasaets based on the 'clientID' column, then each client dataset is split into K spatially disjoint outer blocks and these blocks are used to form the K folds, where a fold contains outer-train/test data. The process is done in such a way that an outer fold contains a block from each client. Then each outer fold is split again using an inner block cross-validation loop into train/validation data splits by using KMeans to build J spatially disjoint clusters. Hyperparameters are tuned using the inner-train and validation data, then the model with the best hyperparameters is trained from scratch using the outer-train data and is evaluated against the test data. For each outer fold, J models with different hyperparameters will be evaluated against the test data.

3.2.1 Configuration File Entries

- 'input dataset path' (string): file path to the input dataset file
- 'selected sensors path' (string): file path to the selected feature/sensors file. This specifies which column will be part fed to the machine learning model
- 'spatial resolution (m)' (float): the grid size of the dataset (spatial resolution), where pixels in the input dataset are expected to form a grid and neighbors are expected be spaced out by at least this entry. This entry isn't used if a tabular model is used (only CNN makes use of this to extract input tiles)
- 'apply min-max scaling' (boolean): flag indicating whether to apply min-max scaling (to rescale the data between 0 and 1) or not . Setting this flag to False is useful when the input dataset has already been normalized between 0
- 'input tile size (k x k)' (int): the tile size of the input samples for tensor-based model (e.g., CNN). E.g., setting this value to 3 means the CNN would have 3x3 tiles as input. Should be 0 for non-tensor models

- 'algorithm' (string): machine learning model name. Available options are:
    - ZeroR: a baseline model that ignore input features and simply predicts the average of the target variable from the training data
    - RF: random forest regressor
    - DNN: deep neural network. A multilayer perceptron with multiple hidden layers
    - CNN: 2D convolutional neural networks
    - LR: linear regression
- 'seed' (int): the seed used fro random number generation to enable experiment reproducibility
- 'outer CV split type' (string): the type of cross-validation to use for the outer cross-validation. Available options are:
    - random CV: random cross-validation such that samples are randomly sampled when creating folds
    - blockCV: block cross-validation such that outer folds are created using clustered sampling (via k-means) to form the spatially disjoint blocks that are used to build folds
- 'inner CV split type' (string): the type of cross-validation to use for the inner cross-validation, which should be the same as the 'outer CV split type' entry. Available options are:
    - random CV: random cross-validation such that samples are randomly sampled when creating folds
    - blockCV: block cross-validation such that inner folds are created using clustered sampling (via k-means) to form the spatially disjoint blocks that are used to build folds
- 'iterations' (int): number of iterations that cross-validation should be repeated. More iterations would mean a better understanding of the average model performance. Should be at least 1
- 'number of outer folds' (int): number of folds used for the outer cross-validation. Must be at least 2. Clients datasets are broken into this many blocks.
- 'number of inner folds' (int): number of folds used for the inner cross-validation. Must be at least 2
- 'number of trials' (int): the number of sets of hyperparameters that are tested out when tuning hyperparameters for a given outer-inner fold pair. Must be at least 1.
- 'number of executions per trial' (int): the number of times a model is re-trained from scratch using a set of hyperparameters to reduce the effects of unlucky initial weights affecting hyperparameter selection. Must be at least 1.
- 'output hyperparameter choice' (Boolean): flag indicating whether the choices of hyperparameters are output for each fold.

3.2.2 Output

The output directory provided to the main function is the root directory of all output files. Each time a configuration file is used to run a batch of experiments, the batch of experiments will have their own output directory named using the yyyy-mm-dd_hh_mm_ss (<year>-<month>-<day>_<hour>_<minute>_<second>) date format naming convention. This subdirectory will have its own result files. A batch of experiments' output subdirectory will have 3 main output elements, namely, 2 output files and a subdirectory for each experiment within the provided configuration file. These output elements are named:

- **experiments**: the subdirectory that will hold all the prediction output files for each experiment specified by the input configuration file (one for each row of the configuration file)
- **results**.csv: the CSV file that summarize every iteration and fold's performance metrics for all of the experiments.
- **log**: the log file that output all information and error messages. The amount of stuff being logged can be controlled by changing the 'GLOBAL_LOG_LEVEL' variable in the 'myio' module. Available log levels are:
    - LOG_LEVEL_DEBUG: a lot of debug information is logged. Useful for debugging errors
    - LOG_LEVEL_INFO: (the default) only a small amount of useful information is logged
    - LOG_LEVEL_WARNING: only warning messages are logged
    - LOG_LEVEL_ERROR: only error messages are logged
- **configFile.csv**: a copy of the configuration file used to run the batch of experiments is made in the output directory.

3.2.2.1 The 'experiments' subdirectory

This subdirectory will have subdirectories for each experiments provided in a configuration file. The naming convention of the subdirectories is 'e*X*', where *X* is the experiment id (the row ID, starting from 0, of a configuration file entry). For each of these 'eX' subdirectories, each iteration will produce an output file named 'iY.csv', where Y is the iteration id. The iY.csv prediction files will have a list of the actual yield value, the coordinates (X,Y) of the sample, and the predictions made by each inner fold's model. This way an analyst can keep track of what prediction was made for what sample (useful for time-series plots of emission predictiosn vs. actual emissions)

The prediction 'iY.csv' files have the following columns:

- **sampleid:** the id of the sample in the input dataset
- **X:** the x coordinate of the sample
- **Y:** the y coordinate of the sample
- **index of sample after CV shuffle:** the index of the sample after cross-validation (CV) was applied and the dataset was shuffled. This way you can sort the output file by this column and you will obtain how the dataset was shuffled
- **outer-fold:** the id of the fold the sample belonged to when it was used for testing in the outer-cross-validation loop
- **actual_yield:** the actual yield reading from the original dataset
- **predicted_inner_fold*i*:** the predicted yield made by the model for inner fold *i*
- **predicted_inner_fold_average:** the average predictions over all inner folds for the sample

An example output file structure would be as follows, if /home/user/n2o/2024-ml was provided as the output directory to the main function and the experiment batch had at least 2 experiments (2 configuration row entries) and the experiments each had at least 2 iterations.

```
#>/home/user/n2o/2024-ml/2024-03-11_12_37_42/
#                                                > log
#                                                >results.csv
#                                                >experiments/
#                                                    >e0/
#                                                        >i0.csv
#                                                        >i1.csv
#                                                        >...
#                                                    >e1/
#                                                        >i0.csv
#                                                        >i1.csv
#                                                        >...
#                                                    >...
```

Furthermore, there is an optional file called 'hyperparameter-choices.csv' (controlled by the 'output hyperparameter choice' configuration file entry) that will contain the hyperparameter choices of the best performing model resulting from hyperparameter tuning, for each iteration, and fold. The 3 first headers are 'iteration', 'outer fold', and 'inner fold', and the remaining headers are for each hyperparameter.

3.2.2.2 The 'results.csv' file

This file contains all the result information of all experiments of a single configuration file, and is useful for creating excel pivot table to analyze average performance over all folds and iterations for all experiments in a batch of experiments specified by a configuration file. This file has the following columns:

- **experiment id:** id of the experiment
- **processing device:** indicates whether the model was trained using a CPU or GPU (deep learning models are expected to be run on GPU, unless the execution environment was poorly configured)
- **algorithm:** the model name
- **spatial resolution (min):** spatial resolution (in meters) of the input dataset (provided by the configuration file entry)
- **input tile size (k x k):** tile size of input samples for tensor-based models (provided by the configuration file entry)
- **number of features:** number of features selected (specified by the selected features input file)
- **number of instances:** number of samples in the input dataset
- **iteration:** the current iteration number
- **seed:** the random number generation seed used (provided by the configuration file entry)
- **outer fold: current** fold number in the outer cross-validation loop
- **inner fold: current** fold number in the inner cross-validation loop
- **MSE:** mean squared error performance of the model on the given outer-inner fold pair
- **RMSE:** root mean squared error performance of the model on the given outer-inner fold pair

- **R2:** coefficient of determination performance of the model on the given outer-inner fold pair
- **MAPE:** mean absolute percentage error performance of the model on the given outer-inner fold pair. Furthermore, note that this is in ratio format and would need to be multiplied by 100 to convert it into percentage error units.
- **execution time(s):** the time (in seconds) taken to train and evaluate the model on the given outer-inner fold pair (does not include the time required to initially load the dataset into memory and pre-process it.

3.2.3 Running the experiment example

Assuming your working directory is setup as follows:

- experimenter.py (and all other python scripts in the same directory)
- input/configs/config.csv
- output/

Run the below line of code in the command line to run the batch of experiments specified by the 'input/configs/config.csv' configuration file, where the results will be output into the 'output' directory:

```
python experimenter.py --inFile input/configs/config.csv  --outDirectory output
```

3.3 Federated learning

These types of experiments simulated distributed learning and various baselines to compare federated learning (FL) to. Centralized learning (CL), local learning (LL), average ensemble (AE) learning, stacked ensemble (SE) learning, and FL are supported. A single dataset is provided and is assumed to have been pre-partitioned by client, where cross-validation is performed on each client.

3.3.1 Configuration File Entries

- 'input dataset path' (string): file path to the input dataset file
- 'selected sensors path' (string): file path to the selected feature/sensors file. This specifies which column will be part fed to the machine learning model
- 'spatial resolution (m)' (float): the grid size of the dataset (spatial resolution), where pixels in the input dataset are expected to form a grid and neighbors are expected be spaced out by at least this entry. This entry isn't used if a tabular model is used (only CNN makes use of this to extract input tiles)
- 'algorithm' (string): machine learning model name. Available options are:
    - ZeroR: a baseline model that ignore input features and simply predicts the average of the target variable from the training data
    - RF: random forest regressor
    - DNN: deep neural network. A multilayer perceptron with multiple hidden layers
    - CNN: 2D convolutional neural networks
    - LR: linear regression

- ‘hyperparameter set’ (string): string used as a lookup key to specify which hyperparameter set to use depending on the ML model specified. (the ‘FLStaticHyperParameterMap’ variable in model.py specifies the hyperparameter choice)
- 'input tile size (k x k)' (int): the tile size of the input samples for tensor-based model (e.g., CNN). E.g., setting this value to 3 means the CNN would have 3x3 tiles as input. Should be 0 for non-tensor models
-
- 'iterations' (int): number of iterations that cross-validation should be repeated. More iterations would mean a better understanding of the average model performance. Should be at least 1
- 'number of outer folds' (int): number of folds used for the outer cross-validation. Must be at least 2. Clients datasets are broken into this many blocks.
- 'number of inner folds' (int): number of folds used for the inner cross-validation. Must be at least 2
- 'number of trials' (int): the number of sets of hyperparameters that are tested out when tuning hyperparameters for a given outer-inner fold pair. Must be at least 1.
- 'number of executions per trial' (int): the number of times a model is re-trained from scratch using a set of hyperparameters to reduce the effects of unlucky initial weights affecting hyperparameter selection. Must be at least 1.
-
- ‘federated aggregation algorithm’ (string): the federated aggregation algorithm, which is only relevant to FL experiments. Available options are (future work is to support more than one aggregation functions):
    - FedAvg: the federated averaging algorithm proposed by McMahan et al., “Communication-Efficient Learning of Deep Networks from Decentralized Data”, in Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, 2017.
- ‘FL experiment type’ (string): the type of learning method to apply over the client datasets. Available options are:
    - centralized learning: all clients simulate sending their train and test data to the cloud, and a single model is trained and evaluated. This is the standard learning approach most works apply that do not take privacy into consideration
    - local learning and ensemble learning: local learning (LL), average ensemble learning (AE), and stacked ensemble learning (SE) experiments are run.
        - LL is first executed, where clients train and evaluate a local model only using their local data.
        - AE and SE simulate each client sending their local model to each other, where AE is evaluated by having a client make a prediction over its test data using each local model and taking the average of all prediction.
        - For SE, a meta linear regression model is trained and evaluate using the predictions of each local model. This is done by using 5-fold cross-validation on a client’s test data to spilt the data into a level 0 meta train set and meta test set, such that the meta train set is fed to each local model to make predictions. These predictions are combined to form a level 1 data that is then used to train

the meta linear regression model. The meta model is then evaluated against the meta test set. This is done for each of the 5 folds.
    - federated learning: this runs FL experiments where clients perform simulated rounds of communication to build the global model progressively by only sharing model weight updates each round.
- 'apply min-max scaling' (boolean): flag indicating whether to apply min-max scaling (to rescale the data between 0 and 1) or not . Setting this flag to False is useful when the input dataset has already been normalized between 0
- ‘client heterogeneity’ (string): determines whether random sampling (without replacement) is applied to each client dataset. Available options are:
    - IID: random sampling without replacement is applied to each client’s dataset, where the number of samples taken is controlled by the ‘number of random samples to take per client’ configuration file entry.
    - non-IID: no random sampling is applied. The entirety of each clients’ datasets is used
- ‘learning rate override’ (float): allows the default learning rate of deep learning algorithms to be overridden. A value of -1 means no override occurs, and another value means a user-defined learning rate can be used.
- ‘epoch override’ (float): allows the default number of epochs of deep learning algorithms to be overridden. A value of -1 means no override occurs, and another value means a user-defined number of epochs can be used.
- ‘batch size override’ (float): allows the default batch size of deep learning algorithms to be overridden. A value of -1 means no override occurs, and another value means a user-defined batch size can be used.
- 'seed' (int): the seed used for random number generation to enable experiment reproducibility
- ‘number of random samples to take per client’ (int): controls how many samples are taken for each client’s data when applying random sampling in the IID dataset split cases (see the ‘client heterogeneity’ configuration file entry). When the ‘client heterogeneity’ is ‘non-IID’ this entry is ignored. When ‘client heterogeneity’ is ‘IID’, then <= 0 means no random sampling and > 0 means random sampling is applied.
- 'cross-validation type' (string): the type of cross-validation to use for the outer cross-validation. Available options are:
    - random CV: random cross-validation such that samples are randomly sampled when creating folds
    - blockCV: block cross-validation such that outer folds are created using clustered sampling (via k-means) to form the spatially disjoint blocks that are used to build folds
- 'iterations' (int): number of iterations that cross-validation should be repeated. More iterations would mean a better understanding of the average model performance. Should be at least 1
- 'number of folds' (int): number of folds used for the cross-validation on client datasets. Must be at least 2
- ‘number of selected clients per round’ (int): the number of clients select in a round of FedAvg during FL experiments. This is unused for CL, LL, AE, and SE. Acceptable value for this are from 1 to the number of clients.
- ‘number of rounds’ (int): number of rounds of communication used in the FL experiments. This is unused for CL, LL, AE, and SE. This should be at least 1.

3.3.2 Output

The same as the cross-validation experiments (see Section 3.2.2)

3.3.2.1 The 'experiments' subdirectory

For CL experiments, this is the same as the cross-validation experiments (see Section 3.2.2.1). For LL and ensemble learning experiments, there are prediction files for each iteration, but instead of being named using the 'iX.csv' notation for iteration *X*, the naming convention is 'iX.avg-ensemble.csv', 'iX.stacked-ensemble.csv', and 'iX.local.csv', for AE, SE, and LL, respectively, for prediction files for iteration *X*. Furthermore, the prediction files have an addition column, 'clientid', that indicate the id of the client the sample belonged to after client dataset partitioning. Furthremore, experiment subdirectories that hold the prediction output files has a 'client' subdirectory containing subdirectories for each client, where the subdirectory of client *Y* is named 'cY'. The client subdirectories similarly have 'iX.avg-ensemble.csv', 'iX.stacked-ensemble.csv', and 'iX.local.csv' output files for each iteration 0 to *X*, but these differ from the files in the parent directory two directories above the client's output directory such that only samples belonging to the client will be part of those prediction files. The prediction files also don't have a 'clientid' column, since all samples belong to the same client.

3.3.2.2 The 'results.csv' file

This file is almost identical to the result file of cross-validation experiments (see Section 3.2.2.2): the differences are as follows:

- There is an added 'FL aggregator' column that indicates what federated learning aggregator method (e.g., FedAvg) was used (only relevant for deep learning models and and federated learning experiments. This column is meaningless otherwise)
- There is an added 'client heterogeneity' column that indicates whether IID or non-IID client data partitioning was used.
- The 'number of instances' column was renamed to 'number of global instances'
- There is an added 'number of local instances' column that indicates how many instances were part of the current client's dataset for a given iteration and fold.
- There is an added 'experiment type' column to indicated whether CL, LL, SE, AE, or FL was used to evaluate the current fold, iteration, and client.
- There is an added 'clientid' column that indicates the id of the client whose dataset is used for model training and evaluation. In the CL case, client id is set to -1, since all clients simulated sending their data to the cloud.
- The 'outer fold' column was removed, since there is no outer cross-validation loop, and the 'inner fold' column was renamed to 'fold'.
- There is an added 'network communication cost (KB)' column that indicates how much data was sent over the simulated network to train and evaluate the model for the current client, fold, and iteration. Note that currently only deep learning models have network cost calculation support. The ML models we didn't implement how much bandwidth it takes to send them over the network

3.3.3 Running the experiment example

Assuming your working directory is setup as follows:

- FLExperimenter.py (and all other python scripts in the same directory)

- input/configs/configMLP.csv
- output/

Run the below line of code in the command line to run the batch of experiments specified by the 'input/configs/configMLP.csv' configuration file, where the results will be output into the 'output' directory:

```
python FLExperimenter.py --inFile input/configs/configMLP.csv  --outDirectory output
```

4 Adding a new Machine Learning Model

This section explains what parts of the code need to be changed to create and implement a new model. The model.py module contains all the model logic and needs to be changed. Below are the list of changes that must be made to model.py:

- Add the name/acronym of the new model as a constant variable using the naming convention 'ALG_' as a prefix to the variable name (e.g., 'ALG_RANDOM_FOREST'). The value of this variable will be the acronym of the model that is specified in the 'algorithm' entry of the configuration file.
- Add an entry to the 'deepLearningModelMap' variable to indicate whether the new model is a deep learning model or not using the new algorithm acronym as the key
- Add an entry to the 'modelsWithTensorsMap' variable to indicate if the model has multi-dimensional input tiles (tensor-based input samples), like CNN, using the new algorithm acronym as the key
- Add an empty dictrionary to the 'FLStaticHyperParameterMap' variable using the new algorithm acronym as the key
- Add the 3 static hyperparameter set choices for each year to the 'FLStaticHyperParameterMap' variable using the new algorithm acronym as the key
- In the constructor of the Model class (the __init__ function), add to the algorithm name if condition structure the new model's name, and call the appropriate function to build the model given the hyperparameters
- Create a new function that takes as input the set of hyperparameters and the function creates the model, initializes it, and returns the new model. The name of the function should following the naming convention 'buildModel_name', where 'Model_name' is the name of the model.
- Modify the 'generateHyperparameterSets' function to add the new model's name to the if structure to define the random search hyperparameter search range of each parameter. The logic is as follows to add a hyperparameter
    - Add an empty list to the 'paramDict' map using the hyperparameter name as the key (e.g., paramDict["units"]=[])
    - Use a for loop to populate the list. The list contains all possible values that random search can sample from. So appending 'i' in the loop 'for i in range(8,256,1):' would mean that i starts at 8, is incremeneted by 1 up until 256.
    - Repeat this for each hyperparameter