Consiglio Nazionale delle Ricerche

Polimi - Financial Engineering AY 2023/2024

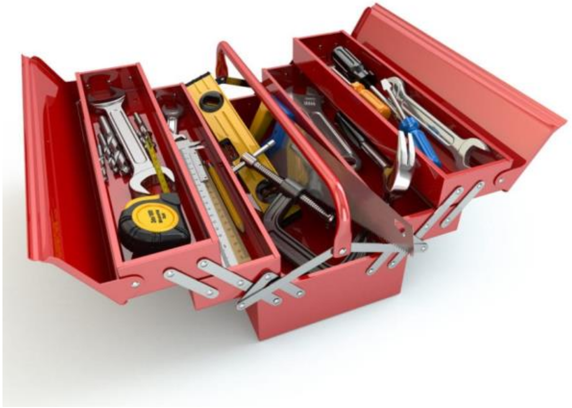
Lab: Electricity Price and Load Forecasting - L4

STIIMA

Alessandro Brusaferri

Codebase structure

- The codebase has been conceived to provide a set of integrated and ready-to-use utilities and functions to develop probabilistic forecasting systems
- It is structured to easily execute the whole PF chain automatically



Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

- Two major usage scenario are foreseen:
  - Use the whole chain by integrating custom models (e.g., add new NN architecture by dedicated class, following the DNN declaration style)
  - Reuse/customize specific classes/functions in brand new projects (e.g., extract the batching procedures, hyperparameter tuning, etc.)

Codebase structure

```
PEPF_lab_v2_L4 ~/PycharmProjects/PEPF_lab_v2_
  data
    datasets
      EM_market__2015-01-03__2017-12-31.csv
      LF_bench_0.csv
  experiments
    tasks
      EM_price
        JSU-DNN
        N-DNN
        point-ARX
        point-DNN
        QR-DNN
          recalib_opt_grid_1_1
            results
            exper_configs.json
            tuned_hyperp-grid_search.json
      NetLoad
  tools
  db.sqlite3
  LICENSE
  main_recalibration.py
  README.md
```

- **Data->** datasets: collection of data (csv)
- **Experiments**: aimed to structure the different experiments configurations (json based) and resuts (pickle)
  - **Task** specific folder (e.g., EM_price, NetLoad). It can be extended.
  - **Model** form specific subfolder (e.g., point-DNN, etc). Further models can be included by dedicating a specific folder (e.g., STU-DNN)
    - It contains a set of folder for running different experiment under the same model by changing the configurations (e.g., recalib_opt_grid_1_1). Further folders can be added (e.g., recalib_opt_grid_1_2)
      - Each run subfolder includes the json configurations of the experiments, the hyperparameter tuning results and the results folder
- **Tools**: structuring the codebase utilities
- **Db.sqlite3**: structuring the different optuna runs
- **main_recalibration.py**: script to run the experiments

main_recalibration

STIIMA

```python
#------------------------------------------------------------------------------------------------------
# Set PEPF task to execute
PF_task_name = 'EM_price'
# Set Model setup to execute
exper_setup = 'JSU-DNN'

#------------------------------------------------------------------------------------------------------
# Set run configs
run_id = 'recalib_opt_grid_1_1'
# Load hyperparams from file (select: load_tuned or optuna_tuner)
hyper_mode = 'load_tuned'
# Plot train history flag
plot_train_history=True
plot_weights=False

#------------------------------------------------------------------------------------------------------
# Load experiments configuration from json file
configs=load_data_model_configs(task_name=PF_task_name, exper_setup=exper_setup, run_id=run_id)

# Load dataset
dir_path = os.getcwd()
ds = pd.read_csv(os.path.join(dir_path, 'data', 'datasets', configs['data_config'].dataset_name))
ds.set_index(ds.columns[0], inplace=True)

#------------------------------------------------------------------------------------------------------
# Instantiate recalibratione engine
PrTSF_eng = PrTsfRecalibEngine(dataset=ds,
                               data_configs=configs['data_config'],
                               model_configs=configs['model_config'])

# Get model hyperparameters (previously saved or by tuning)
model_hyperparams = PrTSF_eng.get_model_hyperparams(method=hyper_mode, optuna_m=configs['model_config']['optuna_m'])

# Exec recalib loop over the test_set samples, using the tuned hyperparams
test_predictions = PrTSF_eng.run_recalibration(model_hyperparams=model_hyperparams,
                                               plot_history=plot_train_history,
                                               plot_weights=plot_weights)
```

- **Configurations:** points to the related experiment folder
  - **PF_task_name**: select the task to
  - **Exper_setup:** select the model to run
  - **Run_id:** select the specific run configs to exec
  - **Hyper_mode**: set 'optuna_tuner' for executing optuna, or 'load_tuned' to load a previously saved json hyperparam set
- The first code block loads the selected dataset as a pandas dataframe (**ds**). Here the dataframe can be pre-processed/transformed before running the recalibration procedures.
- The second block instantiate the recalibration engine, get the hyperparams (either running optuna or loaded from experiment path) and execute the recalibration. '**test_predictions**' includes the out of sample recalibration results as dataframe

exper_configs

```json
{
    "data_config": {
        "dataset_name": "EM_market__2015-01-03__2017-12-31.csv",
        "idx_start_train": {
            "y": 2015,
            "m": 1,
            "d": 3
        },
        "idx_start_oos_preds": {
            "y": 2017,
            "m": 1,
            "d": 1
        },
        "idx_end_oos_preds": {
            "y": 2017,
            "m": 1,
            "d": 3
        },
        "keep_past_train_samples": false,
        "steps_lag_win": 7,
        "pred_horiz": 24,
        "preprocess": "StandardScaler",
        "shuffle_mode": "none",
        "num_vali_samples": 100
    },
    "model_config": {
        "PF_method": "qr",
        "model_class": "DNN",
        "optuna_m": "grid_search",
        "target_alpha": [
            0.01,
            0.02,
            0.03,
            0.04,
            0.05,
            0.06,
            0.07,
            0.08,
            0.09,
            0.10
        ],
        "max_epochs": 800,
        "batch_size": 64,
        "patience": 20,
        "num_ense": 1
    }
}
```

- **idx_start_train:** first past date to include in the train set
- **idx_start_oos_preds**: first out of sample test date (i.e., test set start)
- **idx_start_oos_preds**: last out of sample test dat (test set end)
- **keep_past_train_samples**: whether to keep the past samples as the recalibraiton moving window proceeds
- **steps_lag_win**: number of lags employed to build the moving window sampler, defined as multiplier to the pred_horiz. E.g., 7 with pred_horiz: 24 lead to 168 steps.
- **pre_process**: preprocessing class to call (currently only the StandardScaler is implemented)
- **shuffle_mode**: 'none': no shuffle (just conventional train data shuffle in tensorflow fit). 'vali' shuffle validation samples. 'train_vali' shuffle train and validation samples.
- **num_vali_samples**: number of samples in the set between the train start date and the test start date to be employed as validation subset
- **Model_config**:
    - set the **PF_method** and the **model_class** to be called. If new methods are implemented, the related key has to be selected here in the related experiment folder
    - **Optuna_m**: select 'grid_search' or 'random'
    - **Target_alpha**: the list of alpha employed to build the related couple of quantiles for the different 1-alpha coverage degree
    - **Max_epochs**: maximum number of training epochs
    - **Batch_size**: size of the minibatch
    - **Patience:** number of epochs in the early stop patience callback
    - **num_ense**: size of the ensemble (uniform quantile aggregation supported.

PrTsfRecalibEngine

```
class PrTsfRecalibEngine:
    """
    Main class executing the recalibration process
    """
    def __init__(self, dataset,
                 data_configs: PrTsfDataloaderConfigs,
                 model_configs: Dict):...

    @staticmethod
    def __load_dataset_from_file__(dataset_name: str):...


    def __get_global_idx_from_date__(self, date_id, mode='start'):...

    def __store_reindexed_dataset__(self, data_configs: PrTsfDataloaderConfigs):...

    def __build_test_samples_idxs__(self):...

    def __instantiate_preproc__(self):...

    def __build_recalib_dataset_batches__(self, df: pd.DataFrame, fit_preproc: bool):...

    @staticmethod
    def __build_target_quantiles__(target_alpha: List):...

    @staticmethod
    def __build_alpha_quantiles_map__(target_alpha: List, target_quantiles: List):...

    def __transform_test_results__(self, results_df: pd.DataFrame):...

    2 usages
    def get_exper_path(self):...


    def __save_results__(self, test_results_df):...

    1 usage
    def run_hyperparams_tuning(self, optuna_m:str='random', n_trials: int=10):...

    1 usage
    def get_model_hyperparams(self, method, optuna_m='random'):...

    1 usage
    def run_recalibration(self, model_hyperparams:Dict, plot_history=False, plot_weights=False):...
```

Main class managing and executing the recalibration process

- **utility functions**: automatici moving window batching depending on the json config, preprocessing, quantile building, saving, etc.
- **run_hyperparam_tuning**: execute optuna process
- **get_model_hyperparams**: called by the main script, it loads the store json or run optuna depending on the input config
- **run_recalibration**: execute the recalibration process, depending on the json configs

run_recalibration

```python
# Iterate over test samples
for i_t in range(self.test_set_idxs.shape[0]):
    tf.keras.backend.clear_session()
    print('Recalibrating test sample: ' + str(i_t+1) + '/' + str(self.test_set_idxs.shape[0]))
    test_sample_idx = self.test_set_idxs[i_t]
    # Set index of first train sample, depending on the config
    init_sample = 0 if self.data_configs.keep_past_train_samples else i_t * self.data_configs.pred_horiz

    # Build the current recalibratin batch including preprocessing (preprocess option)
    rec_samples = self.__build_recalib_dataset_batches__(
        self.dataset[init_sample:test_sample_idx+self.data_configs.pred_horiz],
                                fit_preproc=True)

    # Get first rec_block in list
    rec_block = rec_samples.recalibBlocks[0]

    # Merge model configs and hyperparams tuning into the settings dict
    settings = {**self.model_configs, **model_hyperparams}
    # Create ensemble handler
    ensemble = Ensemble(settings=settings)

    # List to store ensemble components preds
    preds_test_e = []

    # Create and fit the ensemble components
    for e in range(settings['num_ense']):
        tf.keras.backend.clear_session()
        model = regression_model(settings=settings,
                                 sample_x=rec_samples.x_test)

        model.fit(train_x=rec_block.x_train, train_y=rec_block.y_train,
                  val_x=rec_block.x_vali, val_y=rec_block.y_vali,
                  plot_history=plot_history
                  )

        # Store ensemble component prediction on test sample
        preds_test_e.append(model.predict(rec_samples.x_test))
        if plot_weights:
            model.plot_weights()

    # Aggregate ensemble predictions
    ensem_preds_test = ensemble.aggregate_preds(preds_test_e)

    # Build and store the prediction quantiles for the current test samples using the selected method
    ens_p = ensemble.get_preds_test_quantiles(preds_test=ensem_preds_test)
    rescaled_PIs = {}
    for i in range(ens_p.shape[-1]):
        rescaled_PIs[self.model_configs['target_quantiles'][i]] = self.preproc['target'].inverse_transform(
            ens_p[:, i:i + 1])[:, 0]
    results_df = pd.DataFrame(rescaled_PIs)
    ensem_test_PIs.append(results_df)
```

Iterate across the oos samples (defined in json):

- Build the related recalibration samples (train,vali,test) through the moving window. Chech this tutorial for more info: https://www.tensorflow.org/tutorials/structured_data/time_series
- Create the ensemble model architecture, following the json configs
- Train each ensemble component and store the test prediction
- Aggregate the ensemble predictions (quantilewise)
- Store and return the oos prediction quantiles as dataframe

Pay attention to your objective

STIIMA

```python
def objective(trial):
    # Clear clutter from previous session graphs.
    tf.keras.backend.clear_session()
    # Update model configs with hyperparams trial
    self.model_configs = self.model_class.get_hyperparams_trial(trial=trial, settings=self.model_configs)

    # Build model using the current configs
    model = regression_model(settings=self.model_configs,
                             sample_x=train_vali_block.x_vali[0:1])

    # Train model
    model.fit(train_x=train_vali_block.x_train, train_y=train_vali_block.y_train,
              val_x=train_vali_block.x_vali, val_y=train_vali_block.y_vali,
              pruning_call=TFKerasPruningCallback(trial,  monitor: "val_loss"),
              plot_history=False)

    # Compute val loss
    results = model.evaluate(x=train_vali_block.x_vali, y=train_vali_block.y_vali)
    return results
```

$$\min_{\Omega} \sum_{n=1}^{N} (f_\Omega(\mathrm{x}_n) - \mathrm{y}_n)^2 + \lambda \sum_{j=1}^{\#\Omega} |\omega_j| + \gamma \sum_{j=1}^{\#\Omega} \omega_j^2$$

- When performing hyper-parameter tuning, return the metric you want to employ to optimize in cross-validation.
- e.g., if your training loss evaluation include regularizer term, perform model.predict() and compute the loss to be optimized by optuna

Integration of custom models in the regressor

STIIMA

```python
class TensorflowRegressor():
    """..."""
    def __init__(self, settings, sample_x):...

    1 usage (1 dynamic)
    def predict(self, x):
        return self.output_handler(self.regressor.predict(x))

    4 usages (4 dynamic)
    def fit(self, train_x, train_y, val_x, val_y, verbose=0, pruning_call=None,

    3 usages (3 dynamic)
    def evaluate(self, x, y):...

    1 usage (1 dynamic)
    def plot_weights(self):...

    def __quantiles_out__(self, preds):...

    def __pred_Normal_params__(self, pred_dists: tfp.distributions):...

    def __pred_JSU_params__(self, pred_dists: tfp.distributions):...
```

1. Map the loss of your custom model
2. Add your custom model to the instantiation phase, as for DNN/ARX in the example code
3. Implement and map the function to handle the output quantiles
4. Map your custom model in the config mapper

```python
# Map the loss to be used
if settings['PF_method'] == 'qr':
    loss = PinballLoss(quantiles=settings['target_quantiles'])
elif settings['PF_method']=='point':
    loss = 'mae'
elif (settings['PF_method'] == 'Normal'
      or settings['PF_method'] == 'JSU'
):
    loss = lambda y, rv_y: -rv_y.log_prob(y)
else:
    sys.exit('ERROR: unknown PF_method config!')
```

```python
# Instantiate the model
if settings['model_class']=='DNN':
    # get input size for the chosen model architecture
    settings['input_size']=DNNRegressor.build_model_input_from_series(x=sample_x,
                                                                     col_names=self.x_columns_names,
                                                                     pred_horiz=self.pred_horiz).shape[1]
    # Build the model architecture
    self.regressor = DNNRegressor(settings, loss)
```

```python
# Map handler to convert distributional output to quantiles or distribution parameters
if (settings['PF_method'] == 'Normal'):
    self.output_handler = self.__pred_Normal_params__
elif settings['PF_method'] == 'JSU':
    self.output_handler = self.__pred_JSU_params__
else:
    self.output_handler =self.__quantiles_out__
```

```python
def get_model_class_from_conf(conf):
    """..."""
    if conf == 'ARX':
        model_class = ARXRegressor
    elif conf == 'DNN':
        model_class = DNNRegressor
```

The ensemble manager class

```python
class Ensemble():
    """..."""
    def __init__(self, settings):
        # store configs for internal use
        self.settings = settings
        # map the methods to use for aggretation and quantile building c
        if (self.settings['PF_method'] == 'point'):
            self.ensemble_aggregator = self.__aggregate_de_quantiles__
            self._build_test_PIs = self.__get_qr_PIs__
        elif (self.settings['PF_method'] == 'qr'):
            self.ensemble_aggregator = self.__aggregate_de_quantiles__
            self._build_test_PIs = self.__get_qr_PIs__
        elif (self.settings['PF_method'] == 'Normal'):
            self.ensemble_aggregator = self.__aggregate_de__
            self._build_test_PIs = self.__build_Normal_PIs__
        elif (self.settings['PF_method'] == 'JSU'):
            self.ensemble_aggregator = self.__aggregate_de__
            self._build_test_PIs = self.__build_JSU_PIs__
        else:
            sys.exit('ERROR: Ensemble config not supported!')

    1 usage
    def aggregate_preds(self, ens_comp_preds):...

    1 usage
    def get_preds_test_quantiles(self, preds_test):...

    @staticmethod
    def __aggregate_de__(ens_comp_preds):...

    @staticmethod
    def __aggregate_de_quantiles__(ens_comp_preds):...

    @staticmethod
    def __get_qr_PIs__(preds_test, settings):...

    @staticmethod
    def __build_Normal_PIs__(preds_test, settings):...

    @staticmethod
    def __build_JSU_PIs__(preds_test, settings):...
```

- From point EPF model ensemble to PEPF ensemble (e.g., in case of multiple DNNs, num_ense > 1)
- Simple aggregation by equally weighted (i.e., uniform) quantile averaging
- From distributional NNs samples to prediction quantiles
- If you develop a new probabilistic approach (as you did e.g., during the development of the JSU), include the function to obtain the quantiles (e.g., __build_JSU_PIs) and add it to the mapper in the class init

The model class

```python
class DNNRegressor:
    def __init__(self, settings, loss):
        self.settings = settings
        self.__build_model__(loss)

    def __build_model__(self, loss):...

    4 usages (4 dynamic)
    def fit(self, train_x, train_y, val_x, val_y, verbose=0, pruning_call=None):...

    1 usage (1 dynamic)
    def predict(self, x):...

    3 usages (3 dynamic)
    def evaluate(self, x, y):...

    5 usages
    @staticmethod
    def build_model_input_from_series(x, col_names: List, pred_horiz: int):...

    1 usage (1 dynamic)
    @staticmethod
    def get_hyperparams_trial(trial, settings):...

    1 usage (1 dynamic)
    @staticmethod
    def get_hyperparams_searchspace():...

    1 usage (1 dynamic)
    @staticmethod
    def get_hyperparams_dict_from_configs(configs):...

    1 usage (1 dynamic)
    def plot_weights(self):...
```

Follow the DNN template to develop your custom models:

- Define the model architecture in __build_model__()
- Customize the input feature construction in build_model_input_from_series()
- Declare the hyperparamenter in the handlers:
  - get_hyperparams_trial: define the ranges to be used by the optuna sampler
  - get_hyperparams_searchspace: discrete set for the grid search (must be contained in the ranges of the trial definition above)
  - get_hyperparams_dict_from_configs: map to the json config

The model class

```
def __build_model__(self, loss):
    x_in = tf.keras.layers.Input(shape=(self.settings['input_size'],))
    x_in = tf.keras.layers.BatchNormalization()(x_in)
    x = (tf.keras.layers.Dense(self.settings['hidden_size'],
                               activation=self.settings['activation'],
                               )(x_in))
    for hl in range(self.settings['n_hidden_layers'] - 1):...
    if self.settings['PF_method'] == 'point':...

    elif self.settings['PF_method'] == 'qr':...

    elif self.settings['PF_method'] == 'Normal':...

    elif self.settings['PF_method'] == 'JSU':...
    else:
        sys.exit('ERROR: unknown PF_method config!')

    # Create model
    self.model= tf.keras.Model(inputs=[x_in], outputs=[output])
    # Compile the model
    self.model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=self.settings['lr']),
                       loss=loss)
```

- Build the model architecture following the keras functional API
- See lesson 2 and tensorflow documentation
- Map the probabilistic forecasting method to be used
- The model class can be defined to support alternative output layers by a common hidden mapper, as we performed during the course

The model class

```python
def build_model_input_from_series(x, col_names: List, pred_horiz: int):
    # get index of target and past features
    past_col_idxs = [index for (index, item) in enumerate(col_names)
                     if features_keys['target'] in item or features_keys['past'] in item]

    # get index of const features
    const_col_idxs = [index for (index, item) in enumerate(col_names)
                      if features_keys['const'] in item]

    # get index of futu features
    futu_col_idxs = [index for (index, item) in enumerate(col_names)
                     if features_keys['futu'] in item]

    # build conditioning variables for past features
    past_feat = [x[:, :-pred_horiz, feat_idx] for feat_idx in past_col_idxs]
    # build conditioning variables for futu features
    futu_feat = [x[:, -pred_horiz:, feat_idx] for feat_idx in futu_col_idxs]
    # build conditioning variables for cal features
    c_feat = [x[:, -pred_horiz:-pred_horiz + 1, feat_idx] for feat_idx in const_col_idxs]

    # return flattened input
    return np.concatenate(past_feat + futu_feat + c_feat, axis=1)
```

```python
#----------------------------------------------------------------
# Features naming conventions employed to build the samples
#----------------------------------------------------------------
features_keys={
    # Employ just a single value related to the prediction
    'const': 'CONST__',
    # Target variable
    'target': 'TARG__',
    # Employ just values included in the configured moving
    'past': 'PAST__',
    # Employ the series value related to the prediction ste
    'futu': 'FUTU__',
}
```

Function aimed to build the model input from the subseries created by the moving window method:

- The PrTsfRecalibEngine pass through the input dataset using a moving window of size (json):
    - steps_lag_win*pred_horiz: in the past
    - pred_horiz: in the future
- The x input to the method include a batch (first dimention) of all the features subseries built by the moving window
- The aim of the method is to build the model input from the subseries of the current batch
- To this end, it employ a naming convention, defined in the features_keys dict (in data_utils)
- Currently, three class of features are supported:
    - Past: e.g., the target series
    - Futu: samples available also in day-ahead (e.g., load forecast)
    - Const: e.g., calendar
- The current implementation use the whole set of lags for the 'past', the whole future value for the 'futu' and a single step for the 'const'
- You can customize the function to change the input features (e.g., just specific past lags, etc.

Consiglio Nazionale delle Ricerche

Thanks

Alessandro Brusaferri