Predict the sow date for winter wheat fields

This notebook details an experiment on training and validating a machine learning model for predicting the sow date.

Experimental setup

Based on the available Sentinel-2 (S2) satellite data and the available DMI climate data, we create feature and target sets consisting of the following:

- Features:
  - Northing (N) and Easting (E) coordinates locating the field
  - Interpolated and re-sampled Sentinel 2 L2A cloud free images (the available images in the timespan (08-01; 11-30) are linearly interpolated to one feature for each 7 days):
    - Raw bands: B01, B02, B03, B04, B05, B06, B07, B08, B8A, B09, B11, B12
    - Vegetation indices: NDVI, NDRE, MSAVI2
  - Daily DMI temperature readings (one aggregated feature for every 7 days in the timespan (08-01; 11-30)):
    - Mean the air temperature
    - Minimum of the minimum temperature
    - Maximum of the maximum temperature
    - Mean of soil temperature
- Target:
  - 3 classes:
    - sow date before September 10
    - sow date between September 10 and September 20
    - sow date after September 20

The cloud free images have been selected using the MSScvm cloud mask.

A random forest classifier model is then trained on data for 37.156 fields and validated on data for 9.290 other fields (a 80/20 train/validation split of the data for all 46.446 fields in the dataset).

The model performance is evaluated using:

- Overall metrics: Accuracy, F1 micro, and F1 macro
- Confusion matrices

Conclusions

- In this first experiment the model achieves accuracy and F1 scores on the order of 80% for the validation set, showing that the model predicts the sow date fairly well on new data.
- For the training set, the accuracy and F1 score are 100%, showing that the model is overfitting. Potentially the model may be improved to reduce overfitting and improve generalization.
- The confusion matrices show that most of the misclassifications are among neighbouring classes, e.g. before September 10 instead of between September 10 and 20.
- There are a lot of potential for improving the model - see the future work section below.

Future work

- An analysis of the misclassifications may provide more insight into how to improve the model.

- The model is trained on field level data. One may try to train a model on precision level and see if it improves the classification. This may require using other models than a random forest that are better suited for batch based training (in order to handle the large training data set).
- A test with a train/validation split based on years, e.g. train on 2017-2020 data and validate on 2021 data, in order to asses the generalization or the model between years.
- A hyperparameter tuning (in particular pruning of the trees) may be a way to improve the model prediction performance and reduce overfitting.
- One may try to train models on more specific classes, e.g. a sow date for each week, or switch to a regression based model that predicts the specific sow date.
- One may try to vary the timespan and frequency of the temporal features (DMI and S2 data).
- One may try to switch to a different model altogether, e.g. a boosting tree classifier or a deep learning based model that is able to learn the temporal aspects of the DMI and S2 data.

In [1]:

```
from pathlib import Path

import dask
import dask.dataframe as dd
import dask.distributed
from dask_ml.wrappers import ParallelPostFit
import geopandas as gpd
import holoviews as hv
import hvplot.pandas
import joblib
from joblib import dump, load
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, f1_score
from sklearn.model_selection import train_test_split
import xarray as xr

client = dask.distributed.Client('localhost:8786')
client
```

Out[1]:

Client
Scheduler: tcp://localhost:8786
Dashboard: http://localhost:8787/status (http://localhost:8787/status)

Cluster
Workers: 20
Cores: 20
Memory: 8.01 TB

In [2]:

```
"""DVC stage configurations."""
FEATURE_CONFIG = {
    'time_column_str_format': '%m-%d - %B %-d'
}

"""DVC stage hyperparameters."""
FEATURE_HYPERPARAM = {
    'time_span': ('{}-08-01', '{}-11-30'),
    'feature_time_delta': '7D',
    'S2_time_series_features': [
        'S2_L2A_B01', 'S2_L2A_B02', 'S2_L2A_B03', 'S2_L2A_B04',
        'S2_L2A_B05', 'S2_L2A_B06', 'S2_L2A_B07', 'S2_L2A_B08', 'S2_L2A_B8A',
        'S2_L2A_B09', 'S2_L2A_B11', 'S2_L2A_B12',
        'S2_L2A_NDVI', 'S2_L2A_MSAVI2', 'S2_L2A_NDRE'],
    'DMI_agg_features': {
        'DMI_air_temperature': ['mean'], 'DMI_maximum_temperature': ['max'],
        'DMI_minimum_temperature': ['min'], 'DMI_soil_temperature': ['mean']},
    'DMI_prognosis_days_delta': None,
    'coord_features': ['N', 'E'],
    'response_name': 'theDate'  # sow_date
}

DATA_PATH = Path('/projects/current/YMP-452_sow_date/')

ML_CONFIG = {
    'random_state': 42,
    'split_field': True,
    'validation_size': 0.2,
    'validation_year': None,
    'model_path': DATA_PATH/'classifier_sow_date.joblib',
    'train_predictions_path': DATA_PATH/'training_predictions.parquet.brotli',
    'val_predictions_path': DATA_PATH/'validation_predictions.parquet.brotli'
    }
ML_HYPERPARAM = {
    'sklearn_classifier': RandomForestClassifier
    }
# The keyword arguments passed to the model.
# NOTE: use the elements from the CONFIG and HYPERPARAM dictionaries above.
MODEL_KEY_ARGS = {
    'random_state': ML_CONFIG['random_state']
}
```

Feature engineering

In [3]:

```
# Copied or modified from https://bitbucket.seges.dk/projects/DDS/repos/yield_map_progn
osis/browse/dvc_pipeline/featurization.py
def create_coord_features(ds, coord_features):
    """Create coordinate features."""
    assert coord_features, 'You must provide at least one coordinate feature.'

    # Create dataframe with all coordinate variables
    df_coords = ds[['N', 'E']].to_dataframe()
    for coord in df_coords.index.names:
        df_coords = df_coords.assign(
            **{coord: df_coords.index.get_level_values(coord)})

    # Create coordinate features dataframe
    df_coord_features = pd.concat(
        [df_coords[[coord]] for coord in coord_features], axis='columns')
    df_coord_features = df_coord_features.rename(
        # Prefix coordinates with "coord_" to avoid ambiguity with index names
        columns={coord: f'coord_{coord}'
                 for coord in df_coord_features.columns})
    df_coord_features = pd.concat(  # Add common index
        [df_coord_features], keys=[ds.ID_DDS_field], names=['ID_DDS_field'])

    return df_coord_features

def create_DMI_agg_features(
        ds, DMI_agg_features, start_date, end_date, feature_time_delta,
        time_column_str_format, prognosis_days_delta):
    """Create aggregated DMI features."""
    assert DMI_agg_features, (
        'You must provide at least one DMI aggregation feature.')

    if prognosis_days_delta is not None:
        end_date = end_date + pd.DateOffset(prognosis_days_delta)

    # Resample and aggregate data within growth season
    df_DMI_features = ds[list(DMI_agg_features.keys())].sel(
        time=slice(start_date, end_date)).to_dataframe().resample(
            feature_time_delta).agg(DMI_agg_features)

    # Format time stamp as string
    df_DMI_features = df_DMI_features.reset_index()
    df_DMI_features['time'] = df_DMI_features.time.dt.strftime(
        time_column_str_format)

    # Create DMI feature set
    df_DMI_features = df_DMI_features.set_index('time').unstack().to_frame().T
    df_DMI_features = pd.concat(  # Replace "0" index with common index
        [df_DMI_features], keys=[(ds.ID_DDS_field, end_date.year + 1)],
        names=['ID_DDS_field', 'harvest_year']).droplevel(-1)
    df_DMI_features.columns = [  # Squash multiindex to single index
        '_'.join(col).strip() for col in df_DMI_features.columns.values]

    return df_DMI_features


def create_empty_feature_set(ds, harvest_years):
    """Create an empty feature set.

    The feature DataFrame has a multi-level index identifying each individual
```

```
    sample (i.e. pixel in a field), by containing the following index levels:
        1. ID_DDS_field: the identifier of each individual field.
        2. harvest_year: the year which the field has recorded harvest data.
        3. N: the Northing coordinate of the pixel within the field.
        4. E: the Easting coordinate of the pixel within the field.

    """
    # Extract N, E indices
    df_empty_feature_set = ds[['N', 'E']].to_dataframe()
    non_dim_coords = set(ds.coords) - set(ds.coords.indexes)
    df_empty_feature_set = df_empty_feature_set[
        df_empty_feature_set.polygon].drop(columns=non_dim_coords)

    # Add harvest_year and ID_DDS_field indices
    df_empty_feature_set = pd.concat(
        [df_empty_feature_set]*len(harvest_years), keys=harvest_years,
        names=['harvest_year'])
    df_empty_feature_set = pd.concat(
        [df_empty_feature_set], keys=[ds.ID_DDS_field], names=['ID_DDS_field'])

    return df_empty_feature_set


def create_S2_time_series_features(
        ds, S2_time_series_features, start_date, end_date, feature_time_delta,
        time_column_str_format):
    """Create S2 time series features."""
    assert S2_time_series_features, (
        'You must provide at least one S2_time_series_feature.')

    # Resample data within growth season
    ds_resampled = ds[S2_time_series_features].sel(
        time=slice(start_date, end_date)).resample(
            time=feature_time_delta).interpolate('linear')

    # Format time stamp as string
    df_resampled = ds_resampled.to_dataframe().reset_index(level='time')
    df_resampled['time'] = df_resampled.time.dt.strftime(
        time_column_str_format)

    # Create feature set dataframe
    df_S2_time_series_features = df_resampled.pivot_table(
        index=['N', 'E'], columns='time', values=S2_time_series_features)
    df_S2_time_series_features = pd.concat(  # Add common index
        [df_S2_time_series_features], keys=[(ds.ID_DDS_field, end_date.year +1)],
        names=['ID_DDS_field', 'harvest_year'])
    df_S2_time_series_features.columns = [  # Squash multiindex to single index
        '_'.join(col).strip()
        for col in df_S2_time_series_features.columns.values]

    return df_S2_time_series_features


def create_response(ds, sow_date, harvest_year):
    """Create the response variable."""
    lower = pd.to_datetime(f'{sow_date.year}-09-10').dayofyear
    upper = pd.to_datetime(f'{sow_date.year}-09-20').dayofyear

    target = pd.cut(
        [sow_date.dayofyear], [0, lower, upper, 366], labels=[0, 1, 2])
```

```
    df_response = ds.polygon.to_dataframe(
        name='polygon_1').drop(columns='polygon_1')
    df_response = df_response.assign(
        target=target.to_list()*df_response.shape[0])
    df_response = df_response[df_response.polygon].drop(columns='polygon')
    df_response = pd.concat(  # Add common index
        [df_response], keys=[(ds.ID_DDS_field, harvest_year)],
        names=['ID_DDS_field', 'harvest_year'])

    return df_response


def create_S2_cloud_free_resampled_dataset(ds):
    """Remove cloudy S2 images and resample S2 data to daily images."""
    # Remove images for days with clouds
    S2_das = [da for da in ds.data_vars if da.startswith('S2')]
    cloud_free_dates = ((ds['S2_L2A_MSScvm_B03_B04'].dropna(
        dim='time', how='all').where(
            # Assign pixels outside polygon cloud=False to enable `any` of whole image,
because NaNs are used as False.
            ds['polygon'], other=False)).any(dim=('N', 'E'))).time
    ds_S2_cloud_free_resampled = ds[S2_das].sel(time=cloud_free_dates)

    # Remove samples outside field polygon
    ds_S2_cloud_free_resampled = ds_S2_cloud_free_resampled.where(
        ds_S2_cloud_free_resampled.polygon)

    # Resample to daily values
    ds_S2_cloud_free_resampled = ds_S2_cloud_free_resampled.resample(
        time='1D').interpolate('linear')

    return ds.drop(S2_das).assign(ds_S2_cloud_free_resampled)


def field_samples(
        field_id, sow_date, file_path, hyperparam, config):
    """Return pixel-wise samples from field raster dataset."""
    # Loaded as a dask array with a single chunk
    ds = xr.load_dataset(file_path, engine='zarr')

    ds.attrs['ID_DDS_field'] = field_id
    ds_cloud_free = create_S2_cloud_free_resampled_dataset(ds)
    # Build list of feature dataframes
    feature_dfs = []

    # Year specific features
    start_date = pd.to_datetime(hyperparam['time_span'][0].format(sow_date.year))
    end_date = pd.to_datetime(hyperparam['time_span'][1].format(sow_date.year))

    # S2 time series features
    if hyperparam['S2_time_series_features'] is not None:
        df_S2_time_series_features = create_S2_time_series_features(
            ds_cloud_free, hyperparam['S2_time_series_features'], start_date,
            end_date, hyperparam['feature_time_delta'],
            config['time_column_str_format'])
        feature_dfs.append(df_S2_time_series_features)

    # DMI features
    if hyperparam['DMI_agg_features'] is not None:
        df_DMI_features = create_DMI_agg_features(
            ds_cloud_free, hyperparam['DMI_agg_features'], start_date,
```

```
            end_date, hyperparam['feature_time_delta'],
            config['time_column_str_format'],
            hyperparam['DMI_prognosis_days_delta'])
        feature_dfs.append(df_DMI_features)

    # Response
    df_response = create_response(ds, sow_date, sow_date.year + 1)
    feature_dfs.append(df_response)

    # Coordinate features
    if hyperparam['coord_features'] is not None:
        df_coord_features = create_coord_features(
            ds_cloud_free, hyperparam['coord_features'])
        feature_dfs.append(df_coord_features)

    # Create empty feature set with correct index.
    df_field_features = create_empty_feature_set(
        ds_cloud_free, harvest_years=[sow_date.year + 1])

    # Aggregate features from pixel to fields-level.
    assert len(df_field_features) > 0, f'First has no pixels {field_id=}'
    df_field_features = df_field_features.iloc[[0]]  # Just the first index row, to kee
p one index per field
    mean_feature_dfs = []
    for feature_df in feature_dfs:
        mean_feature_dfs.append(
            # Cast all features to float32.
            feature_df.astype(np.float32, copy=False).median(
                level='ID_DDS_field'))

    # Join all feature/response dataframes
    for feature_df in mean_feature_dfs:
        df_field_features = df_field_features.join(
            feature_df, how='left', on=feature_df.index.names)

    # Sort feature columns alphabetically based on lowercase
    df_samples = df_field_features[
        sorted(df_field_features.columns, key=str.lower)]

    # Drop rows for pixels outside the field polygon
    # NOTE: drop polygon as coord to keep only one polygon column
    df_polygon = ds.polygon.drop('polygon').to_dataframe()
    # NOTE: swap N and E index to the same order as in df_samples
    df_polygon = df_polygon.swaplevel('N', 'E')
    df_samples = df_samples[df_samples.reset_index(
        level=['ID_DDS_field', 'harvest_year'], drop=True).index.isin(
            df_polygon[df_polygon.polygon].index)]

    return df_samples
```

In [4]:

```
# Drop field as it is to small to have any pixels in polygon - thus no features, hence
 fail.
failed_fields = [
    '18-0028160-89-2',
    '18-0035516-7-1',
    '18-0002118-79-1',
    '19-0009981-4-1',
    '19-0003546-2-1',
    '19-0042820-11-1',
    '19-0038961-1-6',
    '19-0002624-28-1',
    '20-0039269-32-3',
    '20-0034796-13-1',
    '20-0023205-1-1',
    '20-0036697-3-1',
    '20-0007915-11-8',
    '21-0014651-1-1',
    '21-0034880-1-22',
    '21-0020585-8-2',
    '21-0023803-89-2',
    '21-0035860-1-3',
    '21-0030390-11-1',
    '21-0036668-1-4']
gdf_sow_date = gpd.read_parquet('output_data/sow_date_data_final.parquet.brotli')
# Show failed fields
gdf_sow_date.loc[failed_fields]
```

Out[4]:

index	JournalNr	Marknr	theDate	varietyName	harvestYear	dataset	geometry	
18-0028160-89-2	18-0028160	89-2	2017-09-26	Benchmark	2018	KLL	POLYGON ((578573.560 6321642.190, 578629.300 6...	/proje 452_
18-0035516-7-1	18-0035516	7-1	2017-10-17	ø Sheriff	2018	KLL	POLYGON ((485036.940 6250268.610, 485043.440 6...	/proje 452_
18-0002118-79-1	18-0002118	79-1	2017-09-26	Sheriff	2018	KLL	POLYGON ((526154.420 6174434.750, 526167.870 6...	/proje 452_
19-0009981-4-1	19-0009981	4-1	2018-09-20	ø Sheriff	2019	KLL	POLYGON ((492118.730 6097932.910, 492115.550 6...	/proje 452_
19-0003546-2-1	19-0003546	2-1	2018-09-13	Benchmark	2019	KLL	POLYGON ((487882.860 6322283.300, 487880.180 6...	/proje 452_
19-0042820-11-1	19-0042820	11-1	2018-09-18	Ohio	2019	KLL	POLYGON ((504641.460 6278426.100, 504650.200 6...	/proje 452_
19-0038961-1-6	19-0038961	1-6	2018-09-20	ø Elixer	2019	KLL	POLYGON ((486144.820 6086389.060, 486153.920 6...	/proje 452_
19-0002624-28-1	19-0002624	28-1	2018-09-20	KWS Lili	2019	KLL	POLYGON ((671328.070 6085146.370, 671329.810 6...	/proje 452_
20-0039269-32-3	20-0039269	32-3	2019-09-20	ø Sort:	2020	KLL	POLYGON ((661751.740 6186873.790, 661753.170 6...	/proje 452_
20-0034796-13-1	20-0034796	13-1	2019-09-25	Sheriff	2020	KLL	POLYGON ((561117.900 6080565.840, 561431.500 6...	/proje 452_
20-0023205-1-1	20-0023205	1-1	2019-10-31	Sheriff	2020	KLL	POLYGON ((488301.050 6125816.540, 488302.600 6...	/proje 452_

index	JournalNr	Marknr	theDate	varietyName	harvestYear	dataset	geometry	
20-0036697-3-1	20-0036697	3-1	2019-09-15	Kvium	2020	KLL	POLYGON ((485243.770 6205476.050, 485244.720 6...	/proje 452_
20-0007915-11-8	20-0007915	11-8	2019-09-08	Graham	2020	KLL	POLYGON ((693219.470 6103389.090, 693221.360 6...	/proje 452_
21-0014651-1-1	21-0014651	1-1	2020-09-30	ø Elixer	2021	KLL	POLYGON ((485855.620 6235027.790, 485837.700 6...	/proje 452_
21-0034880-1-22	21-0034880	1-22	2020-10-24	Sheriff	2021	KLL	POLYGON ((660053.920 6068227.060, 660054.990 6...	/proje 452_
21-0020585-8-2	21-0020585	8-2	2020-09-20	Sort:	2021	KLL	POLYGON ((504601.490 6305767.780, 504593.340 6...	/proje 452_
21-0023803-89-2	21-0023803	89-2	2020-10-13	Benchmark	2021	KLL	POLYGON ((578629.300 6321837.880, 578635.500 6...	/proje 452_
21-0035860-1-3	21-0035860	1-3	2020-09-06	Sortsblanding	2021	KLL	POLYGON ((566852.180 6126808.110, 566868.690 6...	/proje 452_
21-0030390-11-1	21-0030390	11-1	2020-09-15	Informer	2021	KLL	POLYGON ((504662.100 6278421.730, 504714.500 6...	/proje 452_
21-0036668-1-4	21-0036668	1-4	2020-09-24	KWS Extase	2021	KLL	POLYGON ((719378.130 6100274.070, 719445.660 6...	/proje 452_

In [5]:

```
field_dfs = [
    dask.delayed(field_samples)(
        field_id, field_data.theDate, field_data.zarr_path, FEATURE_HYPERPARAM, FEATURE
_CONFIG)
    for field_id, field_data in gdf_sow_date.drop(index=failed_fields).iterrows()]
# Drop rows with NaNs, as some fields is missing S2 features due to no-cloud-free image
s in start or end of time interval.
df_feature_set = dask.delayed(pd.concat)(field_dfs).dropna()
```

In [6]:

```
futures = client.compute([df_feature_set])
dask.distributed.progress(futures, notebook=False)
```

```
/home/fogh/miniconda3/envs/py38_v5/lib/python3.8/site-packages/distribute
d/worker.py:3387: UserWarning: Large object of size 2.47 MB detected in ta
sk graph:
  (['field_samples-c44705f9-2370-4971-bf51-e8ccc8dea ... 0899f21eb43'],)
Consider scattering large objects ahead of time
with client.scatter to reduce scheduler burden and
keep data on workers

    future = client.submit(func, big_data)    # bad

    big_future = client.scatter(big_data)     # good
    future = client.submit(func, big_future)  # good
  warnings.warn(

[########################################] | 100% Completed |  1hr  3min 2
1.4s
```

Train model

In [7]:

```
# Copied from gaffa
@dask.delayed
def from_dask(dask_collection):
    """Convert dask collection to non-dask collection."""
    try:
        non_dask_collection = dask_collection.compute()
    except AttributeError:
        # Something in dask seems to have changed
        # The compute seems to be implicit now
        non_dask_collection = dask_collection

    return non_dask_collection


@dask.delayed
def partition_delayed(df_or_ser):
    """Repartition a delayed Pandas DataFrame or Series."""
    # Rule of thumb: Create partions of 100 MiB each
    # https://stackoverflow.com/a/45988488
    partition_target_mem_use = 100 * 1024**2
    mem_use = np.sum(df_or_ser.memory_usage(deep=True))

    # We require at least 100 partitions - five for each core in Ahsoka
    # May be an overkill - one should monitor the dask data transfer overhead
    optimal_npartitions = max(100, int(1+mem_use/partition_target_mem_use))
    partitioned = dd.from_pandas(df_or_ser, npartitions=optimal_npartitions)

    return partitioned
```

Create features and target

In [8]:

```
df_features, df_target = df_feature_set.drop(columns='target'), df_feature_set[['targe
t']]
```

Split features and target into training and validation sets

In [9]:

```
# Copied from maize_prognosis_from_satellite_images/dev_notebooks/MPSI_15_ML_utils.py
def run_split_data_pipeline(df_features, df_target, config, client):
    """Split features and target into training and validation sets."""
    supported_configs = ['validation_size', 'validation_year']
    try:
        not_none_configs = [
            config[value] is not None for value in supported_configs]
    except KeyError:
        raise KeyError(
            f'"config" dictionary must contain the keys: {supported_configs}.')
    assert not_none_configs.count(True) == 1, (
        'One and only one of the data split functionalities (i.e. config  ' +
        f'keys: "{supported_configs}" has to be not None!')

    if config['split_field']:
        ser_ids = df_target.index.get_level_values('ID_DDS_field').unique()
        ser_train_ids, ser_val_ids = dask.delayed(
            train_test_split, nout=2)(
                ser_ids, test_size=config['validation_size'],
                random_state=config['random_state'])
        df_train_features = df_features.loc[ser_train_ids]
        df_val_features = df_features.loc[ser_val_ids]
        df_train_target = df_target.loc[ser_train_ids]
        df_val_target = df_target.loc[ser_val_ids]
        print('Splitting data into training and validation samples per field...')
    elif config['validation_size'] is not None:
        (df_train_features, df_val_features,
         df_train_target, df_val_target) = dask.delayed(
            train_test_split, nout=4)(
                df_features, df_target, test_size=config['validation_size'],
                random_state=config['random_state'])
        print(f'Splitting data randomly into "{1-config["validation_size"]}/' +
              f'{config["validation_size"]}" training and validation set, ' +
              'respectively...')
    elif config['validation_year'] is not None:
        ser_feature_years = df_target.index.get_level_values('harvestYear')
        df_train_features = df_features[
            ser_feature_years < config['validation_year']]
        df_val_features = df_features[
            ser_feature_years == config['validation_year']]
        ser_target_years = df_target.index.get_level_values('harvestYear')
        df_train_target = df_target[
            ser_target_years < config['validation_year']]
        df_val_target = df_target[
            ser_target_years == config['validation_year']]
        print('Splitting data into training samples before ' +
              f'"{config["validation_year"]}" and validation samples ' +
              f'only from "{config["validation_year"]}"...')
    else:
        raise NotImplementedError(
            'Other data split functionality than config keys: ' +
            f'"{supported_configs}" is not implemented. You must ' +
            'provide them!')

    (df_train_features, df_val_features, df_train_target, df_val_target) = client.persi
st(
        [df_train_features, df_val_features, df_train_target, df_val_target])

    future_tasks = client.compute(
```

```
        [df_train_features, df_val_features, df_train_target, df_val_target])
    dask.distributed.progress(future_tasks, notebook=False)

    assert df_train_features.index.equals(df_train_target.index).compute(), (
        'Index do not match for df_train_features and df_train_target')
    assert df_val_features.index.equals(df_val_target.index).compute(), (
        'Index do not match for df_val_features and df_val_target')

    print('\nFirst 5 rows in training feature DataFrame:')
    display(df_train_features.head().compute())
    print('\n\nFirst 5 rows in training target DataFrame:')
    display(df_train_target.head().compute())
    print('\n\nFirst 5 rows in validation feature DataFrame:')
    display(df_val_features.head().compute())
    print('\n\nFirst 5 rows in validation target DataFrame:')
    display(df_val_target.head().compute())

    return (df_train_features, df_val_features, df_train_target, df_val_target)
```

In [10]:

```
(df_train_features, df_val_features,
 df_train_target, df_val_target) = run_split_data_pipeline(
    df_features, df_target, ML_CONFIG, client)
```

```
Splitting data into training and validation samples per field...
[########################################] | 100% Completed |  9.8s
First 5 rows in training feature DataFrame:
```

				coord_E	coord_N	DMI_air_temperature_mean 01 - Augu
ID_DDS_field	**harvest_year**	**E**	**N**			
19-0009634-3-0	**2019**	**517345**	**6128335**	517575.0	6128380.0	19.86
21-0025170-4-1	**2021**	**535185**	**6088505**	535345.0	6088655.0	18.19
21-0032084-21-1	**2021**	**448975**	**6264705**	449025.0	6264600.0	16.89
21-0038903-29-0	**2021**	**594205**	**6101445**	594405.0	6101365.0	19.09
19-0007128-12-0	**2019**	**477375**	**6163685**	477470.0	6163755.0	19.38

5 rows × 344 columns

```
First 5 rows in training target DataFrame:
```

				target
ID_DDS_field	**harvest_year**	**E**	**N**	
19-0009634-3-0	**2019**	**517345**	**6128335**	1.0
21-0025170-4-1	**2021**	**535185**	**6088505**	2.0
21-0032084-21-1	**2021**	**448975**	**6264705**	1.0
21-0038903-29-0	**2021**	**594205**	**6101445**	0.0
19-0007128-12-0	**2019**	**477375**	**6163685**	1.0

```
First 5 rows in validation feature DataFrame:
```

ID_DDS_field	harvest_year	E	N	coord_E	coord_N	DMI_air_temperature_mean 01 - Augu
21-0015861-1-0	2021	549535	6232455	549760.0	6232130.0	17.11
20-0032547-21-0	2020	567185	6126675	567265.0	6126680.0	18.36
20-0022495-7-0	2020	467505	6293755	467680.0	6294015.0	17.55
20-0022245-42-1	2020	530945	6326395	531015.0	6326435.0	16.30
20-0018246-77-0	2020	579995	6354305	580155.0	6353995.0	16.41

5 rows × 344 columns

First 5 rows in validation target DataFrame:

ID_DDS_field	harvest_year	E	N	target
21-0015861-1-0	2021	549535	6232455	0.0
20-0032547-21-0	2020	567185	6126675	2.0
20-0022495-7-0	2020	467505	6293755	1.0
20-0022245-42-1	2020	530945	6326395	2.0
20-0018246-77-0	2020	579995	6354305	1.0

Train machine learning model

In [11]:

```
@dask.delayed
def _train_predict_save(
        sklearn_model, model_args, df_feature_set, df_target_set,
        output_model_path):
    """Train, predict and save model.

    Train a scikit-learn (interface compatible) model, predict on the training
    dataset, and save the model.

    Take from Bitbucket:
        - repository: "perceptive_sentinel"
        - file:dvc_pipeline/training.py"
        - commit: "23847227595"

    """
    model = sklearn_model(**model_args)
    with joblib.parallel_backend('dask'):
        model.fit(df_feature_set, df_target_set)
    ddf_feature_set = partition_delayed(
        # Dask dataframe does not implement MultiIndex,
        # so we drop the index since it is lost in sklean anyway.
        df_feature_set.reset_index(drop=True)).compute()
    da_predictions = ParallelPostFit(estimator=model).predict(ddf_feature_set)
    dump(model, output_model_path)

    return da_predictions

def run_training_pipeline(
        df_train_features, df_train_target, model_obj, model_key_args, config,
        client):
    """Train and save predictive model."""
    da_train_predictions = _train_predict_save(
        model_obj,
        model_key_args,
        df_train_features, df_train_target,
        config['model_path'])
    arr_train_predictions = from_dask(da_train_predictions)
    df_train_predictions = df_train_target.assign(
        predictions=arr_train_predictions)
    df_train_predictions = client.persist(df_train_predictions)

    # Save predictions on training set
    save_train_predictions = df_train_predictions.to_parquet(
        config['train_predictions_path'], compression='brotli')

    print('Training and predicting on training set...')
    future_tasks = client.compute([save_train_predictions])
    dask.distributed.progress(future_tasks, notebook=False)

    print('\nFirst 5 training predictions.')
    display(df_train_predictions.head().compute())

    return df_train_predictions
```

In [12]:

```
df_train_predictions = run_training_pipeline(
    df_train_features, df_train_target, ML_HYPERPARAM['sklearn_classifier'],
    MODEL_KEY_ARGS, ML_CONFIG, client)
```

```
Training and predicting on training set...
[########################################] | 100% Completed |  1min  3.8s
First 5 training predictions.
```

				target	predictions
ID_DDS_field	harvest_year	E	N		
19-0009634-3-0	**2019**	**517345**	**6128335**	1.0	1.0
21-0025170-4-1	**2021**	**535185**	**6088505**	2.0	2.0
21-0032084-21-1	**2021**	**448975**	**6264705**	1.0	1.0
21-0038903-29-0	**2021**	**594205**	**6101445**	0.0	0.0
19-0007128-12-0	**2019**	**477375**	**6163685**	1.0	1.0

Validate trained model

In [13]:

```
@dask.delayed
def _load_predict(path_model, df_feature_set):
    """Load and predict.

    Load a scikit-learn (interface compatible) model and predict on a dataset.

    Take from Bitbucket:
        - repository: "perceptive_sentinel"
        - file:dvc_pipeline/training.py"
        - commit: "23847227595"

    """
    model = load(path_model)
    ddf_feature_set = partition_delayed(
        # Dask dataframe does not implement MultiIndex,
        # so we drop the index since it is lost in sklean anyway.
        df_feature_set.reset_index(drop=True)).compute()
    da_predictions = ParallelPostFit(estimator=model).predict(ddf_feature_set)

    return da_predictions


def run_prediction_pipeline(df_val_features, df_val_target, config, client):
    """Validate trained model."""
    # Predict on validation set
    da_val_predictions = _load_predict(
        config['model_path'], df_val_features)
    arr_val_predictions = from_dask(da_val_predictions)
    df_val_predictions = df_val_target.assign(
        predictions=arr_val_predictions)

    # Save predictions on validations set
    save_val_predictions = df_val_predictions.to_parquet(
        config['val_predictions_path'], compression='brotli')

    print('Predicting on validation set using trained model...')
    future_tasks = client.compute([save_val_predictions])
    dask.distributed.progress(future_tasks, notebook=False)

    print('\nFirst 5 validation predictions.')
    display(df_val_predictions.head().compute())

    return df_val_predictions
```

In [14]:

```
df_val_predictions = run_prediction_pipeline(
    df_val_features, df_val_target, ML_CONFIG, client)
```

```
Predicting on validation set using trained model...
[########################################] | 100% Completed | 49.3s
First 5 validation predictions.
```

				target	predictions
ID_DDS_field	harvest_year	E	N		
21-0015861-1-0	**2021**	**549535**	**6232455**	0.0	0.0
20-0032547-21-0	**2020**	**567185**	**6126675**	2.0	2.0
20-0022495-7-0	**2020**	**467505**	**6293755**	1.0	1.0
20-0022245-42-1	**2020**	**530945**	**6326395**	2.0	2.0
20-0018246-77-0	**2020**	**579995**	**6354305**	1.0	1.0

Present model scores

In [19]:

```
def _compute_classification_scores(target, prediction, data_name):
    """Return classification scores."""
    field_target = target.groupby('ID_DDS_field').agg(pd.Series.mode)
    field_prediction = prediction.groupby('ID_DDS_field').agg(pd.Series.mode)

    df = pd.DataFrame({
        'Accuracy_field': accuracy_score(field_target, field_prediction),
        'F1_micro_field': f1_score(field_target, field_prediction, average='micro'),
        'F1_macro_field': f1_score(field_target, field_prediction, average='macro'),
        'Samples_field': len(field_prediction)},
        index=[data_name])

    return df

def _compute_confusion_matrices(target, prediction):
    arr_confusion = confusion_matrix(target, prediction)
    labels = ['(<10Sep, 10Sep]','(10Sep, 20Sep]', '(20Sep, 20Sep<]']
    df_confusion_train = pd.DataFrame(
        arr_confusion, columns=labels, index=labels)
    return df_confusion_train


def present_model_scores(
        df_train_predictions, df_val_predictions, client):
    """Compute regression model scores."""
    df_train_scores = dask.delayed(_compute_classification_scores)(
        df_train_predictions['target'],
        df_train_predictions['predictions'],
        'Training')
    df_val_scores = dask.delayed(_compute_classification_scores)(
        df_val_predictions['target'],
        df_val_predictions['predictions'],
        'Validation')
    df_scores = dask.delayed(pd.concat)(
        [df_train_scores, df_val_scores])
    df_scores = client.persist(df_scores)

    print('Computing model scores on the training and validation set...')
    future_tasks = client.compute([df_scores])
    dask.distributed.progress(future_tasks, notebook=False)

    display(df_scores.compute())

    print('Computing model confusion matrix on the training and validation set...')
    plot_confusion_matrices = []
    for sample_split in ['field']:
        split_plot = []
        for dataset in ['Training', 'Validation']:
            if dataset == 'Training':
                target = df_train_predictions['target']
                prediction = df_train_predictions['predictions']
            else:
                target = df_val_predictions['target']
                prediction = df_val_predictions['predictions']

            if sample_split == 'field':
                df_confusion = dask.delayed(_compute_confusion_matrices)(
                    target.groupby('ID_DDS_field').agg(pd.Series.mode),
                    prediction.groupby('ID_DDS_field').agg(pd.Series.mode)
```

```
            ).compute()
        else:
            df_confusion = dask.delayed(_compute_confusion_matrices)(
                target, prediction).compute()
        plot_confusion_matrices.append(
            df_confusion.replace(0, np.nan).hvplot.heatmap(
                xlabel='Prediction', ylabel='Target',
                title=f'{dataset} sampled per {sample_split}'))

display(hv.Layout(plot_confusion_matrices).cols(1))

return df_scores
```