### **Changing the Data Splitting Percentage in MindsDB**

* **Automatic Splitting:**

MindsDB's AutoML engine, Lightwood, automatically splits data into training, validation, and testing sets. The default split ratio is usually 80-10-10, though this can be dynamic.

* **Splitting Method:**

By default, the split is done using random sampling without replacement. It's stratified on the target column to ensure representative distribution across sets.

* **Custom Splitting:**

While the automatic splitting is handled internally, MindsDB also provides data preparation utilities through the **dataprep\_ml** module in **Python** before charging the data in Mindsdb. This module includes data splitters for train-val-test splits, which can be either simple or stratified.

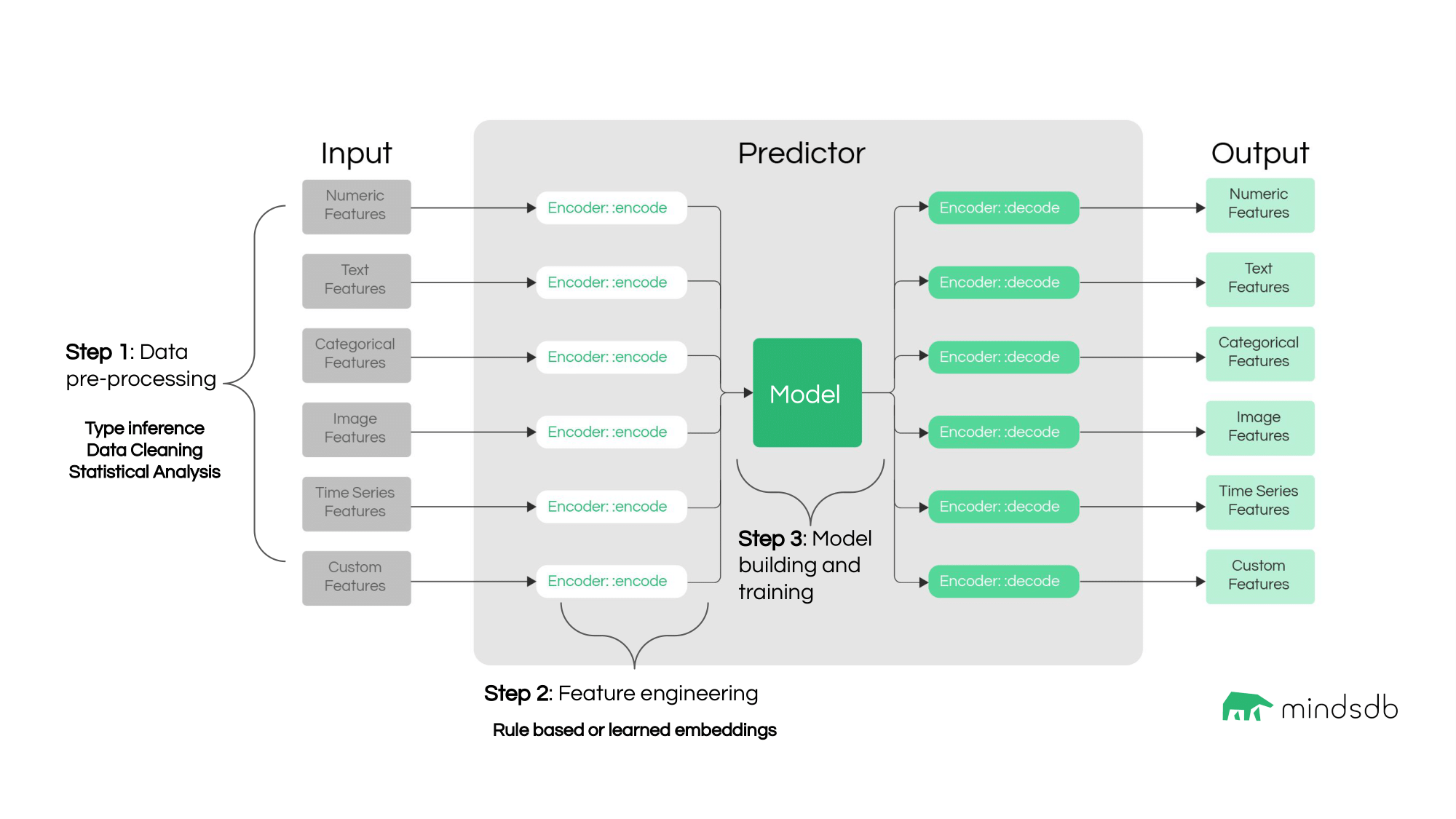
For more information see:

* [mindsdb - dataprep\_ml (github)](https://github.com/mindsdb/dataprep_ml/tree/staging)
* [Build an AI/ML Handler - MindsDB](https://docs.mindsdb.com/contribute/ml-handlers#mindsdb-ml-ecosystem)
* [Build your own training/testing split](https://mindsdb.github.io/lightwood/tutorials/custom_splitter/custom_splitter.html)
* **Usage in ML Handlers:**

For developers creating new ML handlers for MindsDB, it's recommended to use the dataprep\_ml module for data splitting. This helps avoid reimplementing common AutoML functionalities.

* **Evaluation:**

After splitting and training, MindsDB determines the accuracy of the model by evaluating on the held-out test set.

It's worth noting that these splitting mechanisms are primarily used internally by MindsDB's AutoML processes. When querying data or using MindsDB's SQL interface, you typically don't need to manually split the data.

Users can either use [Lightwood’s default mixers/models](https://docs.mindsdb.com/integrations/ai-engines/lightwood#model-key) or create their own approaches inherited from the BaseMixer class.

To learn more about Lightwood philosophy, follow [this link](https://mindsdb.github.io/lightwood/lightwood_philosophy.html).

The code for Python is the following:



| from typing import List, Dict, Union  import numpy as np import pandas as pd from type\_infer.dtype import dtype  from dataprep\_ml.helpers import log   def splitter(  data: pd.DataFrame,  tss,  dtype\_dict: Dict[str, str],  seed: int,  pct\_train: 0.7,  pct\_dev: 0.2,  pct\_test: 0.1,  target: str ) -> Dict[str, Union[pd.DataFrame, list]]:  """  Splits data into training, dev and testing datasets.    The proportion of data for each split must be specified (JSON-AI sets defaults to 80/10/10). First, rows in the dataset are shuffled randomly. Then a simple split is done. If a target value is provided and is of data type categorical/binary, then the splits will be stratified to maintain the representative populations of each class.   :param data: Input dataset to be split  :param tss: time-series specific details for splitting  :param dtype\_dict: Dictionary with the data type of all columns  :param seed: Random state for pandas data-frame shuffling  :param pct\_train: training fraction of data; must be less than 1  :param pct\_dev: dev fraction of data; must be less than 1  :param pct\_test: testing fraction of data; must be less than 1  :param target: Name of the target column; if specified, data will be stratified on this column   :returns: A dictionary containing the keys train, test and dev with their respective data frames, as well as the "stratified\_on" key indicating which columns the data was stratified on (None if it wasn't stratified on anything)  """ # noqa  pct\_sum = pct\_train + pct\_dev + pct\_test  if not (np.isclose(pct\_sum, 1, atol=0.001) and np.less(pct\_sum, 1 + 1e-5)):  raise Exception(f'The train, dev and test percentage of the data needs to sum up to 1 (got {pct\_sum})')   # Shuffle the data  np.random.seed(seed)  if not tss.get('is\_timeseries', False):  data = data.sample(frac=1, random\_state=seed).reset\_index(drop=True)   # Check if stratification should be done  stratify\_on = []  if target is not None:  if dtype\_dict[target] in (dtype.categorical, dtype.binary) and not tss.get('is\_timeseries', False):  stratify\_on = [target]  if tss.get('is\_timeseries', False) and isinstance(tss.get('group\_by', None), list):  stratify\_on = tss['group\_by']   # Split the data  if stratify\_on:  reshuffle = not tss.get('is\_timeseries', False)  train, dev, test = stratify(data, pct\_train, pct\_dev, pct\_test, stratify\_on, seed, reshuffle)  else:  train, dev, test = simple\_split(data, pct\_train, pct\_dev, pct\_test)   # Final assertions for time series  if tss.get('is\_timeseries', False) not in (None, False):  window = tss.get('window', 1) if tss.get('window', 1) else 1  horizon = tss.get('horizon', 1) if tss.get('horizon', 1) else 1   if all([pct\_train, pct\_dev, pct\_test]) > 0.0:  check\_partitions = [train, dev, test]  elif all([pct\_train, pct\_test]) > 0.0:  check\_partitions = [train, test]  elif all([pct\_train, pct\_dev]) > 0.0:  check\_partitions = [train, dev]  else:  check\_partitions = [train]  partition\_lengths = [len(partition) for partition in check\_partitions]   if min(partition\_lengths) < window:  raise Exception(f"Dataset too small for the specified window size ({window}). Partition length: {partition\_lengths}") # noqa   if min(partition\_lengths) < horizon:  raise Exception(f"Dataset too small for the specified horizon size ({horizon}). Partition length: {partition\_lengths}") # noqa   return {"train": train, "test": test, "dev": dev, "stratified\_on": stratify\_on}   def simple\_split(data: pd.DataFrame,  pct\_train: float,  pct\_dev: float,  pct\_test: float) -> List[pd.DataFrame]:  """  Simple split method to separate data into training, dev and testing datasets.   :param data: Input dataset to be split  :param pct\_train: training fraction of data; must be less than 1  :param pct\_dev: dev fraction of data; must be less than 1  :param pct\_test: testing fraction of data; must be less than 1   :returns Train, dev, and test dataframes  """  train\_cutoff = round(data.shape[0] \* pct\_train)  dev\_cutoff = round(data.shape[0] \* pct\_dev) + train\_cutoff  test\_cutoff = round(data.shape[0] \* pct\_test) + dev\_cutoff   train = data[:train\_cutoff]  dev = data[train\_cutoff:dev\_cutoff]  test = data[dev\_cutoff:test\_cutoff]   return [train, dev, test]   def stratify(data: pd.DataFrame,  pct\_train: float,  pct\_dev: float,  pct\_test: float,  stratify\_on: List[str],  seed: int,  reshuffle: bool,  atol: float = 0.05) -> List[pd.DataFrame]:  """  Stratified data splitter.   The `stratify\_on` columns yield a cartesian product by which every different subset will be stratified  independently from the others, and recombined at the end in fractions specified by `pcts`.   For grouped time series tasks, stratification is done based on the group-by columns.   :param data: dataframe with data to be split  :param pct\_train: fraction of data to use for training split  :param pct\_dev: fraction of data to use for dev split (used internally by mixers)  :param pct\_test: fraction of data to use for test split (used post-training for analysis)  :param stratify\_on: Columns to consider when stratifying  :param seed: Random state for pandas data-frame shuffling  :param reshuffle: specify if reshuffling should be done post-split  :param atol: absolute tolerance for difference in stratification percentages. If violated, reverts to a non-stratified split.   :returns Stratified train, dev, test dataframes  """ # noqa   train\_sts = []  dev\_sts = []  test\_sts = []   fractions = np.array([pct\_train, pct\_dev, pct\_test])  groups = data.groupby(by=stratify\_on)  for \_, df in groups:  train\_st, dev\_st, test\_st = np.array\_split(df, (fractions[:-1].cumsum() \* len(df)).round().astype(int))  train\_sts.append(train\_st)  dev\_sts.append(dev\_st)  test\_sts.append(test\_st)   train\_st = pd.concat(train\_sts)  dev\_st = pd.concat(dev\_sts)  test\_st = pd.concat(test\_sts)   if reshuffle:  train\_st, dev\_st, test\_st = [df.sample(frac=1, random\_state=seed).reset\_index(drop=True)  for df in [train\_st, dev\_st, test\_st]]   # check that stratified lengths conform to expected percentages  emp\_tr = len(train\_st) / len(data)  emp\_dev = len(dev\_st) / len(data)  emp\_te = len(test\_st) / len(data)  if not np.isclose(emp\_tr, pct\_train, atol=atol) or \  not np.isclose(emp\_dev, pct\_dev, atol=atol) or \  not np.isclose(emp\_te, pct\_test, atol=atol):  log.warning(  f"Stratification is outside of imposed tolerance ({atol}) ({emp\_tr} train - {emp\_dev} dev - {emp\_te} test), reverting to a simple split.") # noqa  train\_st, dev\_st, test\_st = simple\_split(data, pct\_train, pct\_dev, pct\_test)   return [train\_st, dev\_st, test\_st] |
| --- |