# Storage-Based Handlers

These are all of the storage-based handlers we'll be using for our backtest and deployment mechanisms. The should end up in the following categories.

## Models

Model handlers are there to ensure we can store and train models. Both in production, and in backtest. The two major categories of models are:

1. Online learning models
2. Batch Models

Online models can be retrained in production. While batch models won't have that option.

Each stored model will have some attached metadata to it. It will have two attached metadata fields to it. The first will be metadata describing the model, so we can therefore find it without issue. The second will have valuable metrics for the given data type.

### Model-Base Class

For us to actually make this a reality, we'll have a base class for models that will be an inherited version of the `FileHandler`. That extenstion will have the functions:

1. partial_fit - This trains the model we have by epoch. For fit once models this won't be done very often.
2. fit - fit the model using the partial_fit function above.
3. store - this will be an inherited function from the FileHandler
4. load - this will be an inherited function from the FileHandler
5. monitor - this will take the model that we have an run through a predefined monitoring procedure. We'll have default monitoring procedures for each kind of model by type.
    - Ultimately this montior function will have a metric class that we'll be able to call on. That metric class will have search, as well as a way to monitor the history of a given metric historically.

The `ModelBaseHandler` design pseudo-code:

```
class ModelBase(FileHandler):
   def __init__(self, *args, **kwargs):
      super().__init__(*args, **kwargs)
      self.entity = "entity_name"
      # Model_Type has to be one of the available types.
      self.model_type = "model_type"
```

```
        self.online = False
        # We use the requirements to save the settings
        self.requirements = {
            "...": "..."
        }
        # ModelMetrics Handler will be a mix of multiple metrics that this model will use
        # It should be an inherited version of DBHandler with settings of how we'll monitor it
        # The most recent informaton will be searchable
        # We'll also store the history of each metric
        self._model_metrics : Optional[ModelMetrics] = None
        # The versioning scheme we want to use
        self._versioning: Optional[VersionScheme] = None
        self._episode = uuid.uuid4().hex
        self._live = False
        self._current_model = "..."
        self._base_model = "..."

    """
        # Properties
        ---

        The properties we'll be using:

        * metrics - This will be how we'll monitor the model we have.
        * procedure - This will have a way we'll train the model. It should be different for e
        * available - The available model types we'll have
        * episode - String labelling the episode we have
        * optimizer - The optimizer we will have. Will be different based on the library we us
        * loss - The loss criteria we'll use for the given function
        * live - Boolean labelling if this model is deployed
        * is_deployed - Boolean stating if this model is deployed

    """

    @property
    def available(self):
        return [
            "sklearn", "torch", "keras", "creme", "tensorflow", "sonnet", "jax"
        ]

    @property
    def procedure(self):
        proc = self._get_general_procedure()
        proc.online = self.online
        return proc
```

```
@property
def preprocessing(self):
   preproc = self._get_preprocessing_procedure()
   preproc.online = self.online
   return preproc

@property
def metrics(self):
   self._model_metrics.episode = self.episode
   self._model_metrics.live = self.live
   self._model_metrics.info = self.info # StorageHandler().info is the new way to get al
   return self._model_metrics

@metrics.setter
def metrics(self, _metrics:ModelMetrics):
   self._model_metrics = _metrics

def live(self):
   return self._live

@live.setter
def live(self, _live:bool):
   self._live = _live

@property
def model(self):
   # reset should have been called by now
   return self._current_model

def verify(self):
   """ Verifies the information that that's inside of the constructor"""
   if not isinstance(self.model_type, str) or (self.model_type not in self.available):
      raise ValueError("Model type is not valid type")

   if not isinstance(self.online, bool):
      raise ValueError("Not able to determine if this model is online")
   # No idea what else should be here

def _get_general_procedure(self) -> ProcedureAbstract:
   """ We'll return a procedure"""
   return {
      "sklearn":SklearnProcedure(),
      "torch": TorchProcedure(),
      "keras":KerasProcedure(),
      "creme":CremeProcedure(),
      "tensorflow":FlowProcedure(),
```

```
        "sonnet":SonnetProcedure(),
        "jax": JaxProcedure()
    }

def _get_preprocessing_procedure(self) -> PreprocessingProcedureAbstract:
    """ We'll return a procedure"""
    return {
        "sklearn":SklearnProcedure,
        "torch": TorchProcedure,
        "keras":KerasProcedure,
        "creme":CremeProcedure,
        "tensorflow":FlowProcedure,
        "sonnet":SonnetProcedure,
        "jax": JaxProcedure
    }


def upgrade(self):
    """ Upgrades the model using the versioning scheme"""
    pass

def save_all(self):
    pass

def load_all(self):
    pass


"""
    # ML Specific Functions

"""


def fit(self, data):
    """ Fit the model"""
    try:
        model = self.model
        procedure = self.procedure.fit(model, data)
        self.model = self.model
        # What ever else we want to do here
    except Exception as e:
        logger.exception(e)

def partial_fit(self, data):
    """ Partial fit the model"""
```

```
        try:
            model = self.model
            procedure = self.procedure.partial_fit(model, data)
            self.model = self.model
            # What ever else we want to do here
        except Exception as e:
            logger.exception(e)

    def predict(self, data):
        with catch_exception():
            prediction = self.model.predict(data) # Will require some abstractions over the mo
            return prediction
        return []

    def predict_prob(self, data):
        """ """
        pass
    """
        # Gym Like Functions
        - Step:
            - gets the model and predicts accordingly
            - preprocesses data to be handled
            - if it's online, we partial fit the data before
        - Reset:
            - Loads or saves model depending on if we've saved a model already
            - Loads settings as well

    """




    def reset(self):
        """ # Reset
            ---
            Resets is a Gym like interface. Does the following:

            1. Creates a new model if it doesn't exist yet
                * Sets the version using the versioning scheme
            2. Load the latest model if it does exist
            3. Saves or loads all of the settings
        """
        pass

    def step(self, data:Any):
        """ """
```

```
      preprocessed = self.proprocessing.step(data)
      if self.online:
         self.partial_fit(preprocessed)

      pass

   """
      # Saving functions
   """

   def latest_model(self):
      """ Get the latest model by version"""
      model = self.last(av="version")
      if model is None:
         raise AttributeError("The latest model doesn't exist")


   """
      # Loading Functions
   """
```

## General

Blob information that exist within various groups simulations we'd end up running.