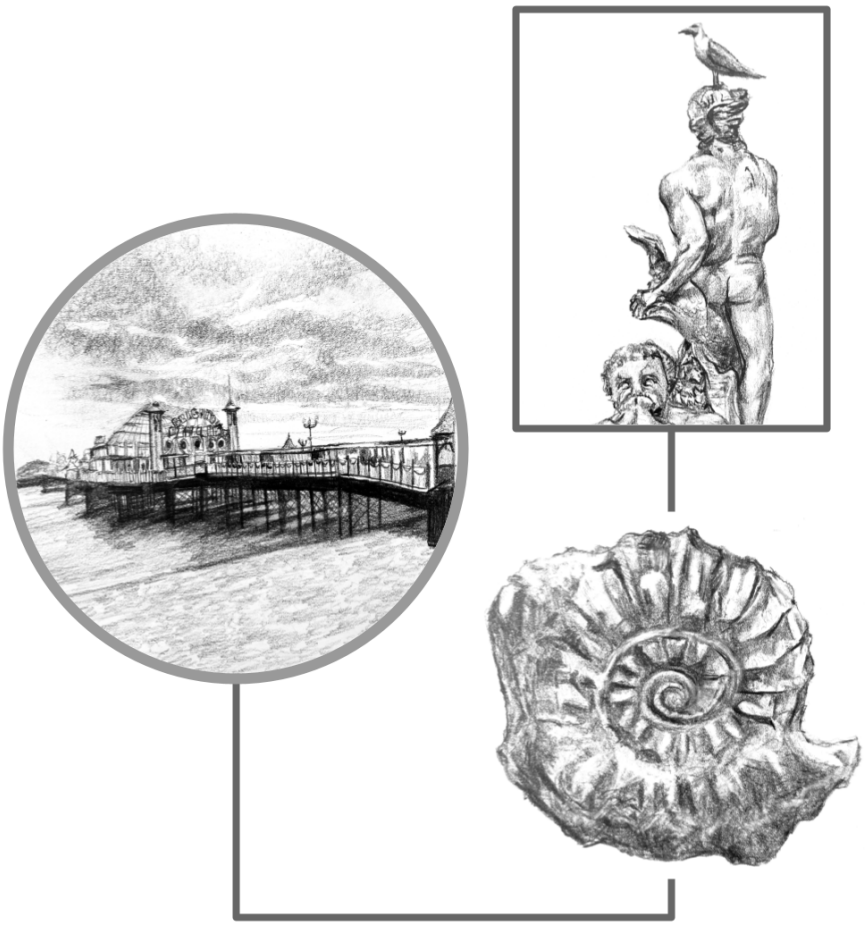

4

Empirical probabilistic filtering

Concept. To extend the formalism that we developed in previous chapters to enable the empirical emulation of real-world data via a probabilistic filter. This technique should enable a researcher to model complex dynamical trends in the data very well; at the cost of making the abstract interpretation of the model less immediately comprehensible than the statistical inference models in some proceeding chapters. As our generalised framework applies to a wide variety stochastic phenomena, our filtering algorithm will be applicable to a great breadth of data modeling problems as well. We will also explore some examples which illustrate how the filter should be applied in practice and then follow this up with how the code is designed and implemented as part of a new software package called the 'learnadex'. For the mathematically-inclined, this chapter will take a detailed look at how our formalism can be extended to focus on probabilistic filters and their optimisation using real-world data. For the programmers, the software described in this chapter lives in the public Git repository: https://github.com/umbralcalc/learnadex.

4.1 Probabilistic formalism

The key distinction between the methods that we will develop in this chapter and the ones in the proceeding chapters is in their utility when faced with the problem of attempting to model real-world data. In the proceeding chapter, we shall describe some powerful techniques that can be used most effectively when the researcher is aware of the family of models that generated the data. In the present chapter, we will go into the details of how a more 'empirical' approach can be derived for dynamical process modeling in a probabilistic framework which locally adapts the model to the data through time.

While we think that it's worth going into some mathematical detail to give a better sense of where our formalism comes from; we want to emphasise that the framework we discuss here is not new to the technical literature at all. Our overall framework draws on influences from Empirical Dynamical Modeling (EDM) [1], some classic nonparametric local regression techniques — such as LOWESS/Savitzky-Golay filtering [2] — and also Gaussian processes [3] as well. The novelties here, instead, lie more in the specifics of how we combine some of these ideas together when referencing

the stochadex formalism, and how this manifests in designing more generally-applicable software for the user.

Before we are able to develop this empirical filtering algorithm, we need to return to the stochadex formalism that we introduced in the first chapter of this book. As we discussed at that point; this formalism is appropriate for sampling from nearly every stochastic phenomenon that one can think of. However, when trying robustly assess how far a model is from accurately describing a set of real-world data, trying to use only generated samples of the model process can be diffcult. Instead, in this section, we are going to extend this formalism to look at how probability theory can help with this data comparison problem in a systematic way.

So, how do we begin? In the first chapter, we defined the general stochastic process with the formula $X^i_{\mathtt{t}+1} = F^i_{\mathtt{t}+1}(X', z, \mathtt{t})$. This equation also has an implicit *master equation* associated to it that fully describes the time evolution of the *probability density function* $P_{\mathtt{t}+1}(x)$ of the next matrix row being $x = X_{\mathtt{t}+1}$. This can be written as

$$P_{\mathtt{t}+1}(x) = \frac{1}{\mathtt{t}} \sum_{\mathtt{t}'=0}^{\mathtt{t}} \int_{\omega_{\mathtt{t}'}} \mathrm{d}x' P_{\mathtt{t}'}(x') P_{(\mathtt{t}+1)\mathtt{t}'}(x|x') \,, \tag{4.1}$$

where at the moment we are assuming the state space is continuous in each dimension and $P_{(\mathtt{t}+1)\mathtt{t}'}(x|x')$ is the conditional probability that the matrix row at time $(\mathtt{t}+1)$ will be $x = X_{\mathtt{t}+1}$ given that the row at time $\mathtt{t}'$ was $x' = X_{\mathtt{t}'}$. This is a very general equation which should almost always apply to any continuous stochastic phenomenon we want to study in due course. To try and understand what this equation is saying we find it's helpful to think of an iterative relationship between probabilities; each of which is connected by their relative conditional probabilities. This kind of thinking is also illustrated in Fig. 4.1.

Let's say we also wanted to program what this equation is saying as a function in Go. Using a Monte Carlo approximation for the integral domain, the code might look something like this.

```
type StateVector  []float64

// returns a random draw of the possible state vectors at this timestep
func RandomPossibleStateVectors(timeStepNumber int) []StateVector {
    // return a slice of randomly-drawn possible state vectors
    // corresponding to the integral domain at this timestep
}

// returns the conditional probability of the state vector at this timestep
// given the value that the state vector had on a previous timestep
func StateVectorConditionalProbability(
    stateVector StateVector,
    timeStepNumber int,
    previousStateVector StateVector,
    previousTimeStepNumber int,
) float64 {
    // return the conditional probability value
}

// returns the probability of the state vector at this timestep
func StateVectorProbability(
    stateVector StateVector,
    timeStepNumber int,
) float64 {
    prob := 0.0
```

```
    // loop over all the possible previous timesteps
    for t := 0; t < timeStepNumber; t++ {
        // loop over the randomly-drawn possible state vectors
        // for this previous timestep
        possibleStateVectors := RandomPossibleStateVectors(t)
        for _, possibleStateVector := range possibleStateVectors {
            // note the recursion
            prob += StateVectorProbability(possibleStateVector, t)*
                StateVectorConditionalProbability(
                    stateVector,
                    timeStepNumber,
                    possibleStateVector,
                    t,
                )
        }
        // normalisation for the Monte Carlo integration
        prob /= float64(len(possibleStateVectors))
    }
    // timestep normalisation
    prob /= float64(timeStepNumber)
    return prob
}
```

The factor of $1/\mathsf{t}$ in Eq. (4.1) is a normalisation factor — this just normalises the sum of all probabilities to 1 given that there is a sum over t'. Note that, if the process is defined over continuous time, we would need to replace

$$\frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \to \frac{1}{t(\mathsf{t})}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \delta t(\mathsf{t}')\,. \tag{4.2}$$

But what is ω_{t}? You can think of this as just the domain of possible x' inputs into the integral which will depend on the specific stochastic process we are looking at.

If we wanted to compute the mean of the distribution $M_{\mathsf{t}+1}$ in Eq. (4.1), it would be straightforward to just multiply both sides of the expression by x and integrate over $\mathrm{d}x$ in the ω_{t} domain. However, there is another similar expression for the mean that we can derive under certain conditions which will be valuable to us when developing the empirical filter. If the probability distribution is *stationary* — meaning that $P_{\mathsf{t}'}(x) = P_{\mathsf{t}''}(x)$ for all t' and t'' — it's possible to derive[1]

$$M_{\mathsf{t}+1} = \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, x\, P_{\mathsf{t}+1}(x) = \frac{1}{\mathsf{t}}\sum_{\mathsf{t}'=0}^{\mathsf{t}} \int_{\omega_{\mathsf{t}'}} \mathrm{d}x' \int_{\omega_{\mathsf{t}+1}} \mathrm{d}x\, x'\, P_{\mathsf{t}'}(x') P_{(\mathsf{t}+1)\mathsf{t}'}(x|x')\,. \tag{4.3}$$

[1] To see that this is true, first note that the joint distribution $P_{(\mathsf{t}+1)\mathsf{t}'}(x,x') = P_{(\mathsf{t}+1)\mathsf{t}'}(x|x')P_{\mathsf{t}'}(x')$. Secondly, note that joint distributions always allow variable swaps trivially like this $P_{(\mathsf{t}+1)\mathsf{t}'}(x,x') = P_{\mathsf{t}'(\mathsf{t}+1)}(x',x)$. Then, lastly, note that stationarity of $P_{\mathsf{t}+1}(x) = P_{\mathsf{t}'}(x)$ means

$$\int \mathrm{d}x \int \mathrm{d}x'\, x\, P_{(\mathsf{t}+1)\mathsf{t}'}(x,x') = \int \mathrm{d}x \int \mathrm{d}x'\, x\, P_{\mathsf{t}'(\mathsf{t}+1)}(x,x') = \int \mathrm{d}x \int \mathrm{d}x'\, x\, P_{(\mathsf{t}+1)\mathsf{t}'}(x',x)\,,$$

where we've used the trivial variable swap to get to the last equality, and the domain references ω in the integrals are implicitly defined.

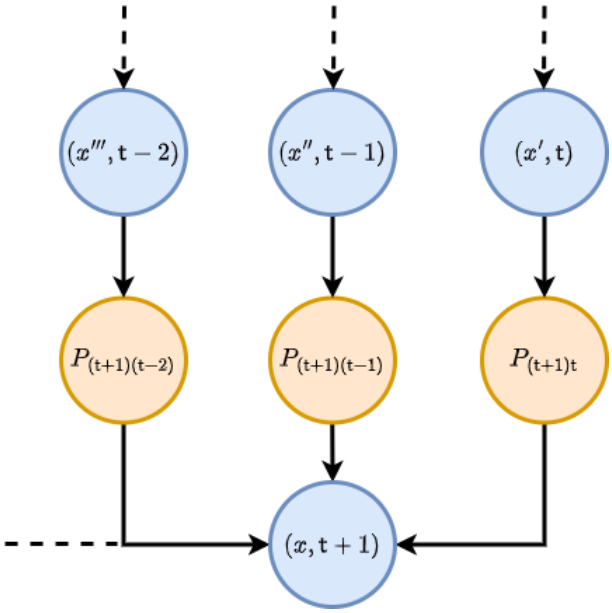

Figure 4.1: Graph representation of Eq. (4.1).

The standard covariance matrix elements can also be computed in a similar fashion

$$\begin{aligned} C^{ij}_{\mathrm{t}+1} &= \int_{\omega_{\mathrm{t}+1}} \mathrm{d}x\,(x - M_{\mathrm{t}+1})^i (x - M_{\mathrm{t}+1})^j P_{\mathrm{t}+1}(x) \\ &= \frac{1}{\mathrm{t}} \sum_{\mathrm{t}'=0}^{\mathrm{t}} \int_{\omega_{\mathrm{t}'}} \mathrm{d}x' \int_{\omega_{\mathrm{t}+1}} \mathrm{d}x\,(x' - M_{\mathrm{t}+1})^i (x' - M_{\mathrm{t}+1})^j P_{\mathrm{t}'}(x') P_{(\mathrm{t}+1)\mathrm{t}'}(x|x')\,. \end{aligned} \tag{4.4}$$

While they look quite abstract, Eqs. (4.3) and (4.4) express the core idea behind how our probabilistic filter will function. By assuming a stationary distribution, we gain the ability to directly estimate the statistics of the probability distribution $P_{\mathrm{t}+1}(x)$ from past samples it may have in empirical data; which are represented here by $P_{\mathrm{t}'}(x')$.

In order to study higher-order out-of-time-order correlations, we can also consider using the statistical moments computed from joint distributions like these

$$P_{(\mathrm{t}+1)\mathrm{t}'}(x, x') = P_{(\mathrm{t}+1)\mathrm{t}'}(x|x') P_{\mathrm{t}'}(x') \tag{4.5}$$

$$P_{(\mathrm{t}+1)\mathrm{t}'\mathrm{t}''}(x, x', x'') = P_{(\mathrm{t}+1)\mathrm{t}'\mathrm{t}''}(x|x', x'') P_{\mathrm{t}'\mathrm{t}''}(x'|x'') P_{\mathrm{t}''}(x'')\,. \tag{4.6}$$

For example, Eq. (4.5) would apply if we wanted to retrieve the out-of-time-order pairwise correlations between x at timestep $\mathrm{t}+1$ and x' at timestep t'. Using this pairwise relationship we can also derive an equation for the out-of-time-order covariance matrix elements

$$C^{ij}_{(\mathrm{t}+1)\mathrm{t}'} = \frac{1}{\mathrm{t}} \sum_{\mathrm{t}'=0}^{\mathrm{t}} \int_{\omega_{\mathrm{t}'}} \mathrm{d}x' \int_{\omega_{\mathrm{t}+1}} \mathrm{d}x\,(x - M_{\mathrm{t}+1})^i (x' - M_{\mathrm{t}'})^j P_{(\mathrm{t}+1)\mathrm{t}'}(x, x')\,. \tag{4.7}$$

What other processes can be described by Eq. (4.1)? For Markovian phenomena, the equation no longer depends on timesteps older than the immediately previous one, hence the expression reduces to just

$$P_{\mathsf{t}+1}(x) = \int_{\omega_{\mathsf{t}}} \mathrm{d}x' P_{\mathsf{t}}(x') P_{(\mathsf{t}+1)\mathsf{t}}(x|x') \,. \tag{4.8}$$

An analog of Eq. (4.1) exists for discrete state spaces as well. We just need to replace the integral with a sum and the schematic would look something like this

$$P_{\mathsf{t}+1}(x) = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{\omega_{\mathsf{t}'}} P_{\mathsf{t}'}(x') P_{(\mathsf{t}+1)\mathsf{t}'}(x|x') \,, \tag{4.9}$$

where we note that the P's in the expression above all now refer to *probability mass functions*. In the even-simpler case where x is just a vector of binary 'on' or 'off' states, Eq. (4.9) reduces to

$$P^i_{\mathsf{t}+1} = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{j=1}^{d} P^j_{\mathsf{t}'} P^{ij}_{(\mathsf{t}+1)\mathsf{t}'} = \frac{1}{\mathsf{t}} \sum_{\mathsf{t}'=0}^{\mathsf{t}} \sum_{j=1}^{d} \left[P^j_{\mathsf{t}'} A^{ij}_{(\mathsf{t}+1)\mathsf{t}'} + (1 - P^j_{\mathsf{t}'}) B^{ij}_{(\mathsf{t}+1)\mathsf{t}'} \right] , \tag{4.10}$$

where $P^i_{\mathsf{t}'}$ now represents the probability that element $x^i = 1$ (is 'on') at time t'. The matrices A and B are defined as conditional probabilities where the previous state in time $P^j_{\mathsf{t}'}$ was either 'on' or 'off', respectively.

In this section, we looked into how the mathematical formalism used in the stochadex could be extended with probability theory. Now that we have more of a sense of how this formalism works, we are ready to move on to designing the algorithm for our filter. So let's go!

4.2 Generalised filtering algorithm

The empirical probabilistic filtering algorithm that we're going to describe in this section will depend on the stationarity of $P_{\mathsf{t}+1}(x) = P_{\mathsf{t}'}(x)$ such that, e.g., Eq. (4.3) is applicable. But before we think about the various kinds of conditional probability we could use in the filter, we need to connect values of x in our filter to the data from which it will be learning.

If the mean is a sufficient statistic for the distribution which describes the data, a choice of, e.g., Exponential, Poisson or Binomial distribution could be used where the mean is estimated directly from the time series using Eq. (4.3), given a conditional probability $P_{(\mathsf{t}+1)\mathsf{t}'}(x|x')$. Extending this idea further to include distributions which also require a variance to be known, e.g., the Normal, Gamma or Negative Binomial distributions could be used where the variance (and/or covariance) could be estimated using Eq. (4.4). These are just a few simple examples of distributions that can link the estimated statistics from Eqs. (4.3) and (4.4) to a time series dataset. However, the algorithmic framework is very general to whatever choice of 'data linking' distribution that a researcher might need.

We should probably make what we've just said a little more mathematically concrete. Let's define a data vector y at time $\mathsf{t}+1$, which shares the same length as x, and represents a specific observation of x at this point in a real dataset. At this point in time, we can define a distribution $P_{\mathsf{t}+1}(y; M_{\mathsf{t}+1}, C_{\mathsf{t}+1}, \dots)$ which represents the likelihood of y given the estimated statistics from Eqs. (4.3) and (4.4) (and maybe higher-orders). Note that in order to do this, we need to identify

the x' and t' values that are used to estimate, e.g., $M_{\mathsf{t}+1}$ with the past data values which are observed in the dataset time series itself. Now that we have this likelihood, we can immediately define an overall objective function (a cumulative log-likelihood) that we might seek to optimise over for a given dataset

$$\ln \mathcal{L}_{\mathsf{t}+1} = \ln P_{\mathsf{t}+1}(y; M_{\mathsf{t}+1}, C_{\mathsf{t}+1}, \dots) + \ln P_{\mathsf{t}}(y'; M_{\mathsf{t}}, C_{\mathsf{t}}, \dots) + \dots \,, \tag{4.11}$$

where the summation continues until all of the past measurements have been taken into account.

In order to specify what $P_{(\mathsf{t}+1)\mathsf{t}'}(x|x')$ is, it's quite commmon to define a number of new hyperparameters. For example; one clear, and generally applicable, option for the conditional probability is that of a Gaussian process

$$P_{(\mathsf{t}+1)\mathsf{t}'}(x|x') = \frac{\exp\left\{-\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d}[x - f(\mathsf{t}+1)]^i \left[K^{-1}(\theta, \mathsf{t}+1, \mathsf{t}')\right]^{ij} [x' - f(\mathsf{t}')]^j\right\}}{|2\pi K(\theta, \mathsf{t}+1, \mathsf{t}')|^{\frac{1}{2}}} \,, \tag{4.12}$$

where K is a covariance matrix; f is a function in time and θ represents a set of hyperparameters which are both typically learned by optimisation of the corresponding likelihood function of the data; and $|A|$ denotes computing the determinant of matrix A.

By optimising Eq. (4.11) with respect to all of these hyperparameters which are defined with $P_{(\mathsf{t}+1)\mathsf{t}'}(x|x')$, we are effectively adopting what is known as an 'empirical Bayes' approach [3, 4]. We might think of the overall generalised filtering algorithm as the following Go code.

```go
type MeanVector   []float64
type DataVector   []float64
type Hyperparams  []float64
type DataHistory  []DataVector

// return the conditional probability of the 'dataVector'
// given the 'givenDataVector'
func ConditionalProbability(
    dataVector DataVector,
    givenDataVector DataVector,
    hyperparams Hyperparams,
) float64 {
    // return the conditional probability
}

// returns the current mean given a conditional probability weighting
// over past data vectors in the input history
func ComputeFilteredMean(
    dataHistory DataHistory,
    hyperparams Hyperparams,
) MeanVector {
    cumulativeWeightsSum := 0.0
    meanVector := make([]float64, len(dataVector[0]))
    for i, dataVector := range dataHistory {
        // ignore first (most recent) value in the history as this
        // is the one we want to compare to in the log-likelihood
        if i == 0 {
            continue
        }
        weight := ConditionalProbability(
            dataHistory[0],
            dataVector,
```

```
            hyperparams,
        )
        cumulativeWeightsSum += weight
        for j := range meanVector {
            meanVector[j] += weight * dataVector[j]
        }
    }
    // normalise the weights derived from the conditional probability
    for j := range meanVector {
        meanVector[j] /= cumulativeWeightsSum
    }
    return meanVector
}

// generates a log-probability for the current vector datapoint
// given the statistics produced by the probabilistic filter
// where inputs could be extended to include, e.g., the filtered
// covariance matrix
func DataLinkingLogProbability(
    dataVector DataVector,
    filteredMean MeanVector,
    hyperparams Hyperparams,
) float64 {
    // return a log-probability from, e.g., a Poisson distribution
}

// compute the overall log-likelihood for the most recent point found
// in the input the data history
func LogLikelihood(
    dataHistory DataHistory,
    hyperparams Hyperparams,
    timeStepNumber int,
) float64 {
    // get the most recent point in the data history
    dataVector := dataHistory[0]
    mean := ComputeFilteredMean(dataHistory, hyperparams)
    logLikelihood := DataLinkingLogProbability(
        dataVector,
        mean,
        hyperparams,
    )
    return logLikelihood
}
```

In the case of Eq. (4.12) above, the hyperparameters that would be optimised will not just include θ; but also a set of values for f at different timesteps. This would essentially equate our optimised filter to be the equivalent to the maximum a posteriori (MAP) of the Gaussian process in this instance. The only difference here is that by passing in sufficient statistics of this process to some data-linking distribution, we enable many different kinds of data to be described by the same underlying conditional probability-based filter. A Gaussian process also only represents one choice in the unlimited number of possible filters we are able to try in the very simple framework we're describing.

As another form of flexibility, one might also wish to try adapting the data-linking distributions to include an intercept term and linear coefficients for the statistics which are passed to it. These could hence be treated as additional hyperparameters and optimised jointly with the others if there

is sufficient constraining power in the data.

The optimisation approach that one chooses to use for obtaining the best hyperparameters in the conditional probability of Eq. (4.11) will depend on a few factors. For example, if the number of hyperparameters is relatively low, but their gradients are difficult to calculate exactly; then a gradient-free optimiser (such as the Nelder-Mead [5] or particle swarm [6, 7] method) would likely be the most effective choice. On the other hand, when the number of hyperparameters ends up being relatively large, it's usually quite desriable to utilise the gradients in algorithms like vanilla Stochastic Gradient Descent [8] (SGD) or Adam [9].

4.3 Software design

Let's now take a step back from the specifics of the probabilistc filtering algorithm to introduce our new sofware package for this part of the book: the 'learnadex'. At its core, the learnadex algorithm adapts the stochadex iteration engine to iterate through streams of data in order to accumulate a global objective function value with respect to that data. The user may then choose which optimisation algorithm (or write their own) to use in order to leverage this objective for learning a better representation of the data. Readers with some familiarity of machine learning concepts will note that this design should enable either 'offline' or 'online' learning algorithms to work.

Using the iteration engine of the stochadex makes neat use of software which has already been designed and tested in earlier chapters of this book. However, in order to fully achieve this, a few minor extensions to the typing structures and code abstractions are necessary; as we show in Fig. 4.2. To start with, we separate out 'learning' from the kind of optimiser in the overall config so as to enable multiple optimisation algorithms to be used for the same learning problem. The hyperparameters that define that optimisation problem domain are also mapped to a format that any algorithm can use via a 'translator' abstraction that is specified by the user as well.

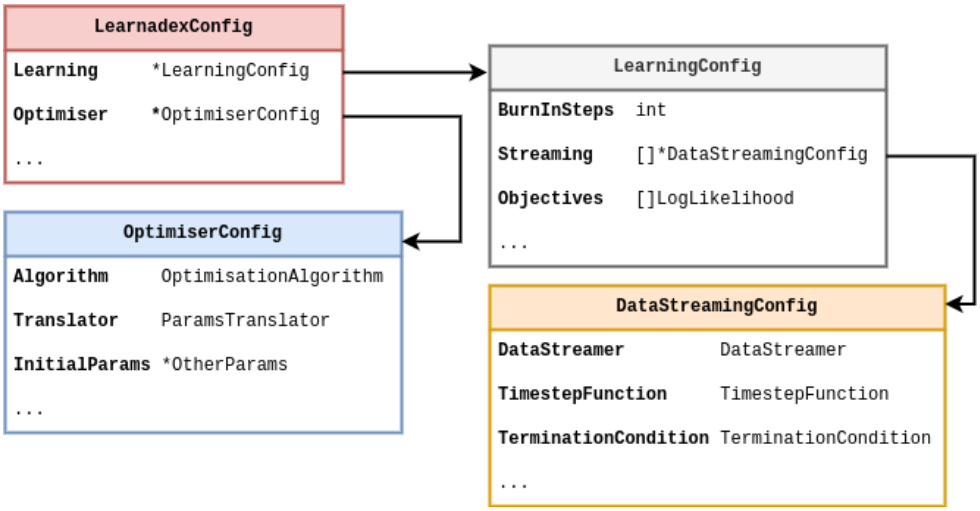

Figure 4.2: A relational summary of the core data types in the learnadex.

On the learning side; in order to define a specific objective for each data iterator to compute

while the data streams through it, we have abstracted a 'log-likelihood' type. Similarly, each iterator also gets a data streamer configuration which defines where the data is streaming from — e.g., from a file on disk, from a local database instance or maybe via a network socket — and also some inherited abstractions from the stochadex which define the time stepping function and when the data stream ends. In Fig. 4.3 below, we provide a schematic of the method calls of (and within) each data iterator.

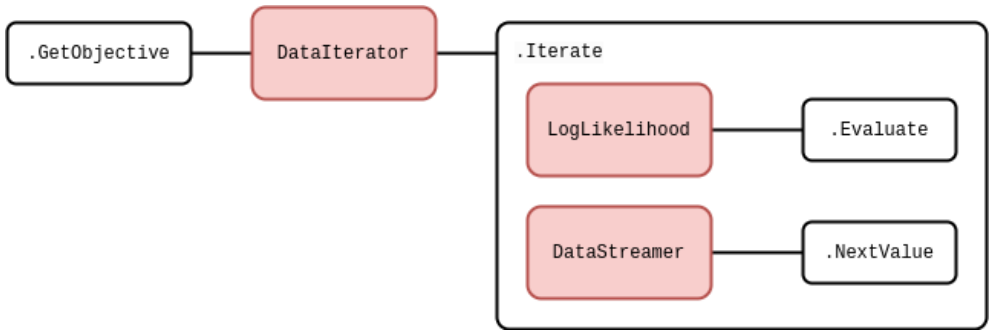

Figure 4.3: A schematic of the basic data iterator struct in the learnadex.

Bibliography

[1] G. Sugihara and R. M. May, *Nonlinear forecasting as a way of distinguishing chaos from measurement error in time series*, *Nature* **344** (1990) 734–741.

[2] A. Savitzky and M. J. Golay, *Smoothing and differentiation of data by simplified least squares procedures.*, *Analytical chemistry* **36** (1964) 1627–1639.

[3] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[4] D. J. MacKay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[5] J. A. Nelder and R. Mead, *A simplex method for function minimization*, *The computer journal* **7** (1965) 308–313.

[6] J. Kennedy and R. Eberhart, *Particle swarm optimization*, in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4, pp. 1942–1948, IEEE, 1995.

[7] Y. Shi and R. Eberhart, *A modified particle swarm optimizer*, in *1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360)*, pp. 69–73, IEEE, 1998.

[8] H. Robbins and S. Monro, *A stochastic approximation method*, *The annals of mathematical statistics* (1951) 400–407.

[9] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, *arXiv preprint arXiv:1412.6980* (2014) .