Machine Learning for the Quantified Self

Chapter 8
Predictive Modeling with Notion of Time
Overview

• Previously: we looked at learning algorithms that did not model time explicitly

• Today: we will look at algorithms that consider time explicitly
  – Time series
  – Recurrent neural networks
  – Dynamical systems models
Time Series

• Time series focus on:
  – Understanding periodicity and trends
  – Forecasting
  – Control

• Time series can be decomposed in three components:
  – Periodic variations (daily, weekly, … seasonality)
  – Trend (how the mean evolves over time)
  – Irregular variations (left after we remove the periodic variations and trend)
Stationarity (1)

• Important concept: stationarity
  – Trends and periodic variations are removed
  – Variance of the remaining residuals is constant

• Prerequisite or intermediate step for many algorithms

• Additional criterion: the lagged autocorrelation should remain constant (note: \( x_t \) represents the value of one attribute):

\[
\rho_l = \frac{\sum_{t=1}^{N-l} (x_t - \bar{x})(x_{t+l} - \bar{x})}{\sum_{t=1}^{N} (x_t - \bar{x})^2}
\]
Stationarity (2)

- Autocorrelation represents in how far there is a correlation between a time series and a shifted version of itself (with $\lambda$ steps).

![Graphs showing autocorrelation for different time series]

- Figure 8.2: Lagged Autocorrelation for various time series:
  - (a) random
  - (b) moderate autocorrelation
  - (c) non-stationary

In 8.2(b) we introduce some memory over a couple of time steps which leads to autocorrelation. How we can calculate this will be discussed in the next sections.

Finally, 8.2(c) displays a time series that is generated by calculating the cumulative sum of the time series under (a). Because time series (c) exhibits a linear trend, you see immediately that it is non-stationary. In the next section we learn how we can model trends and periodicity of time series.

8.2.2 Filtering and smoothing

To analyze trends in time series, we start with a simple approach that is close to what we have already seen in Chapter 2. Remember, what we did in order to reduce the noise of the raw data: We defined a regular spaced time grid with a time window $D_t$ and aggregated the unevenly spaced (raw data) measurements accordingly ($X_T$).

As a result we were able to not only reduce the variability of data and to handle missing values but also to bring the data into a rectangular shape as it is useful for a variety of machine learning algorithms discussed in Chapter 7.
Filtering and smoothing (1)

• Let us assume our time series of values $x_t$ with a fixed step size $\Delta t$

• We can apply a filter to our data, taking $q$ points in the future and past into account:

$$z_t = \sum_{r=-q}^{q} a_r x_{t+r}$$

• This generates a new time series $z_t$
Filtering and smoothing (2)

• What could $a_r$ look like?
  – If we take $a_r = (2q + 1)^{-1}$ it is just the moving average
  – If measurements closer to $t$ are more important we can use a triangular shape:
    \[
    a_r = \begin{cases} 
    \frac{q-|r|}{q^2} & -q \leq r \leq q \\
    0 & \text{otherwise}
    \end{cases}
    \]
  – Or exponential smoothing (only past time points mostly):
    \[
    a_r = \frac{\alpha(1 - \alpha)^{|r|}}{2 - \alpha}
    \]
Filtering and smoothing (3)

- Example exponential smoothing

\[ a = 0.2 \]

\[ a = 0.05 \]

Figure 8.3: Exponential Smoothing of the time series from Figure 8.2. Black dotted (red dashed) line corresponds to \( a = 0.2 \) and \( a = 0.05 \). This corresponds to a new linear filter with weights \( b_r = a^r \) for all \( r \) except \( r = 0 \) for which \( b_r = 1 \).

We apply this by using our exponential smoothing filter (Figure 8.4).

"Art" is a term we use in the title of this book - here we have another example: it is a kind of an art to say what the \( q \) and \( a_r \) should be to find trends, it depends on our understanding of the problem at hand. For example, if you want to use a smartphone to detect stepping patterns of person that has her phone in the pocket, there might be periodic pattern with a frequency of about 1Hz. At the same time the orientation and position of the smartphone in the pocket of that person might vary over longer time intervals. To extract the stepping pattern, we would like to remove changes that are due to the changing orientation of the smartphone, so we would use a time interval for smoothing that is significantly larger than 1 second. We will return to this in an example at the end of this section.

8.2.3 Autoregressive Integrated Moving Average Model - ARIMA

In this section we will take a more conceptual approach to modeling time series. We do so because we ultimately want to estimate a model that describes the empirical data well and that can be used to predict future data points.
Now how can we remove a trend?

Let us take a filter again, but a simple one:

\[ z_t = x_t - x_{t-1} = \nabla x_t \]

This takes the difference between the current and previous measurement

- A long-term trend has more or less the same influence on the previous and current time point

We can apply this operator \(d\) times (e.g. \(d=2\)):

\[ \nabla^2 x_t = \nabla x_t - \nabla x_{t-1} \]
Filtering and smoothing (5)

- But $x_{t-1}$ might not be a good estimation of the trend, we can therefore also use an exponential smoothing $z_t$ and take $x_t - z_t$
ARIMA (1)

• Of course we would like to forecast, let us turn to ARIMA
  – Assume a measurement at time point $t$ is generated by a probability distribution $P_t$
  – The expected mean is: $\mu(t) = E[P_t]$
  – The auto-covariance function is:
    $$\gamma(t_1, t_2) = E[(P_{t_1} - \mu(t_1))(P_{t_2} - \mu(t_2))]$$
  – A series is stationary when the mean is constant and when the auto-covariance only depends on the time difference $\lambda = t_2 - t_1$
ARIMA (2)

• We assume that the probability distribution at time point $t$ is regressed on its own lagged values ($p$ past measurements to be precise)

• $W_t$ is the noise we encounter. We can account for the noise by a moving average component (with $q$ past values)

• Overall the model becomes:

$$P_t = \phi_1 P_{t-1} + \ldots \phi_p P_{t-p} + W_t + \theta_1 W_{t-1} + \ldots \theta_q W_{t-q}$$
ARIMA (3)

- In addition, we consider differencing with an order \( d \)
- So how do we find the values for \( p, q, \) and \( d \)?
  - For \( p \) we can look at the correlation between \( x_t \) and \( x_{t-p} \) (Partial Autocorrelation Function)
  - We can do a grid search over the other parameters and determine the goodness of fit (e.g. using the Akaike Information Criterion)
- And how about the other parameters?
  - We use our data, e.g. for autoregressive component:
    \[
    S = \sum_{t=p+1}^{N} (x_t - \phi_1x_{t-1} - \cdots - \phi_p x_{t-p})
    \]
ARIMA - example (1)

• One step ahead prediction (p=3, q=2)
ARIMA - example (2)

- Multiple steps ahead prediction ($p=3$, $q=2$)
ARIMA - example (3)

- **Seasonality decomposition**

---

Fig. 8.10: The data decomposed in a seasonality and a residual part.

Fig. 8.11: The blue line indicates the original time series that was used to estimate the SARIMA model. The red line is the "long-term" prediction, the shaded area its uncertainty.
• Multiple steps ahead prediction with seasonality (p=3, q=2)

![Graph](image.png)
Neural networks with time

- Variant of neural networks that take time component into account explicitly
- Let us consider the simplest variant first:

![Recurrent Neural Network Diagram]

Figure 8.12: Simplified Recurrent Neural Network

Compared to the neural network examples we have previously considered recurrent neural networks can contain cycles that feed the values of a neuron at a previous time point (assuming discrete time steps) back into the network. Hence, we create a form of memory. In our figure, the output of the network ($Y_1(t)$) is fed back as input. Time is expressed by means of the brackets behind each attribute, so we predict the next value ($Y_1(t+1)$) based on the previous values of the input ($X_1(t),...,X_p(t)$) and the previous value of the output ($Y_1(t)$). Precisely what values are fed back into the network depends on design choices, in our figure it concerns the output, but any choice can be made here.

Let us consider our example shown in Figure 8.1 again. We might want to predict the mood of Bruce based on the activity level. We can build a recurrent neural network for predicting mood at the next point in time (i.e. $t+1$). As input for the model, we use the previous mood predictions ($Y_1(t)$) and the other measurements at the previous time point ($X_1(t),...,X_p(t)$, in this case only activity level).
Recurrent neural networks (1)

- Not very different from our previous neural networks, but cycles make training tricky
  - How do we handle this?
  - We unfold the network, and apply back propagation again
Recurrent neural networks (2)

- How does this work mathematically?
  - Update of regular connections remains the same:
    \[
    \Delta w_{ij} = \eta \delta_j(t)\hat{y}_t^{(i)}
    \]
    \[
    \delta_j(t) = \begin{cases} 
    \varphi'(v_j(t))(y_t^j - \hat{y}_t^{(j)}) & \text{if } j \text{ is an output node} \\
    \varphi'(v_j(t))\sum_1^k (\delta_k(t)w_{jk}) & \text{otherwise}
    \end{cases}
    \]
  - For recurrent connections we take:
    \[
    \Delta w_{ij} = \eta \delta_j(t)\hat{y}_t^{(i)}
    \]
    \[
    \delta_j(t - 1) = \varphi'(v_j(t - 1))\sum_1^k (\delta_k(t)w_{jk})
    \]
Echo state networks (1)

- Training of RNN’s is difficult
- Echo state networks try to tackle this problem
- Have a reservoir of randomly collected neurons

![Diagram of an Echo State Network](image)

The effect of a previous state $r_i$ and a previous input $x_i$ on a future state $x_{i+k}$ should vanish gradually as time passes (i.e. $k \to \infty$) and not persist or even get amplified.

In other words, the reservoir should never amplify or let values of states/neurons persist. Otherwise, it will be impossible to learn the temporal patterns we envision.

Unfortunately no formal procedures exist that are guaranteed to give us a reservoir with the echo state property. However, a heuristic exists that works well in practice.

If you are unfamiliar with the terminology that follows there is no need to worry,
Echo state networks (2)

- We have several matrices of weights:
  
  \( \mathbf{W}^{\text{in}} \) is an \( n \times p \) matrix for the weights from the input layer to the reservoir.
  
  \( \mathbf{W} \) is the \( n \times n \) matrix of the internal weights in the reservoir.
  
  \( \mathbf{W}^{\text{out}} \) is the \( l \times n \) matrix that specifies the weights to the connections between the reservoir and the output.

- \( \mathbf{W}^{\text{in}} \) and \( \mathbf{W} \) are randomly set and fixed, we only learn \( \mathbf{W}^{\text{out}} \)

- We compute the output as follows:

  \[
  r_{i+1} = \varphi(\mathbf{W}^{\text{in}}x_{i+1} + \mathbf{W}r_i)
  \]

  \[
  \hat{y}_{i+1} = \varphi_{\text{out}}(\mathbf{W}^{\text{out}}r_{i+1})
  \]
Echo state networks (3)

• And we simply learn a $W^{\text{out}}$ that minimizes the difference between the actual and predicted $y$

• How should we create the random reservoir?
  – It should satisfy the echo state property

**Definition 8.1.** Echo state property: The effect of a previous state $r_i$ and a previous input $x_i$ on a future state $r_{i+k}$ should vanish gradually as time passes (i.e. $k \to \infty$) and not persist or even get amplified.
Echo state networks (4)

• And how do we establish this?
  – Follow this procedure:

<table>
<thead>
<tr>
<th>Algorithm 19: Reservoir initialization procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Randomly initialize an internal weight matrix $W_0$. $W_0$ should be sparse and have a mean of 0. The size $n$ reflects the number of training examples $N$ (should not exceed $\frac{N}{10}$ to $\frac{N}{2}$ depending on the complexity)</td>
</tr>
<tr>
<td>2. Normalize $W_0$ to matrix $W_1$ with unit spectral radius by putting $W_1 = \frac{1}{\rho(W_0)} W_0$</td>
</tr>
<tr>
<td>3. Scale $W_1$ to $W = \alpha W_1$ where $\alpha &lt; 1$, whereby $\rho(W) = \alpha$</td>
</tr>
<tr>
<td>Then $W$ is a network with the echo state property (has always found to be)</td>
</tr>
</tbody>
</table>
Dynamical Systems Models (1)

- The final type of temporal model we will focus on are dynamical systems models.
- Express differential equations between attributes and targets.

![Graphical illustration of dynamical systems model](image-url)

**Algorithm 19:** Reservoir initialization procedure

1. Randomly initialize an internal weight matrix $W_0$. $W_0$ should be sparse and have a mean of 0. The size $n$ reflects the number of training examples $N$ (should not exceed $N^2$ to $N^3$ depending on the complexity).
2. Normalize $W_0$ to matrix $W_1$ with unit spectral radius by putting $W_1 = \frac{1}{\lambda(W_0)} W_0$.
3. Scale $W_1$ to $W = a W_1$ where $a < 1$, whereby $\lambda(W) = a$.

Then $W$ is a network with the echo state property (has always found to be)

The lower the value of the parameter $a$, the faster the dynamics of the reservoir are.

Well, there you go.

8.4 Dynamical Systems Models

While we have seen models with a lot of expressivity, we started without any knowledge on the precise form of the relationships between the different attributes and targets: these had to be learned from the data. What if we do have some knowledge about the form of the relationships, for instance obtained from literature in the specific domain? Obviously, we want to exploit this knowledge. An approach to build domain knowledge-based models that cover temporal relationships are so-called dynamical systems models. The models represent the temporal relationships between attributes and targets by means of differential equations (of arbitrary complexity) and assume only numerical states. An example graphical illustration is shown in Figure 8.15.
Dynamical Systems Models (2)

- Equations are based on domain knowledge (e.g. scientific papers)
- Models do contain parameters that can be tuned
  - that is where the machine learning will come into play
Let us consider an example model – model the relationship between activity and mood (think of the example of Bruce)

\[
\hat{y}_{\text{mood}}(t + \Delta t) = y_{\text{mood}}(t) +
\]
\[
x_{\text{outside}}(t) \cdot (\gamma_1 \cdot (1 - y_{\text{mood}}(t))) \cdot \text{pos}(y_{\text{activity level}}(t) - y_{\text{mood}}(t)) + \gamma_2 \cdot y_{\text{mood}}(t) \cdot \text{neg}(y_{\text{activity level}}(t) - y_{\text{mood}}(t))) \cdot \Delta t
\]
\[
\hat{y}_{\text{activity level}}(t + \Delta t) = y_{\text{activity level}}(t) +
\]
\[
(\gamma_3 \cdot (1 - y_{\text{activity level}}(t))) \cdot \text{pos}(\sin(\frac{t - \gamma_4 \pi}{\gamma_5})) + \gamma_4 \cdot y_{\text{activity level}}(t) \cdot \text{neg}(\sin(\frac{t - \gamma_4 \pi}{\gamma_5}))) \cdot \Delta t
\]

\[
pos(v) = \begin{cases} 
0 & v < 0 \\
\nu & \text{otherwise}
\end{cases}
\]

\[
neg(v) = \begin{cases} 
\nu & v < 0 \\
0 & \text{otherwise}
\end{cases}
\]
Parameter optimization (1)

• How important are the parameters?
Parameter optimization (2)

• How do we find appropriate parameter settings?
  – Manual tweaking requires a lot of labor
  – We have data available that allows us to find “the best” parameter values using machine learning
  – Assume $\lambda$ is a vector of parameter values, then the error we see is

\[
E(\lambda) = \sum_{t=1}^{N} (\hat{y}(t) - y(t))^2
\]

where $\hat{y}(t)$ is the prediction of the model given $\lambda$
Simulated Annealing (1)

• Let us first focus on a very simple algorithm: simulated annealing
  – We randomly move in our parameter space
    • so we randomly select values for our parameters
  – Is that all?
    • Nope, we assume a \textit{temperature} of our process, and a maximum number of iterations ($k_{\text{max}}$)
    • We accept parameter vectors that are better
    • Depending on the temperature we can also accept parameter vectors that are worse (exploration)
    • The closer we get to the end (the lower the temperature), the lower the probability of accepting vectors with worse performance
Simulated Annealing (2)

Algorithm 20: Simulated Annealing

\[
\begin{align*}
\lambda_{\text{current}} &= \text{random} \\
E_{\text{prev}} &= \infty \\
\text{Temp} &= \text{Temp}_{\text{init}} \\
\text{for } k \text{ from } 1 \text{ to } k_{\text{max}} \text{ do} \\
&\quad \text{for } i \text{ in } \lambda \text{ do} \\
&\quad\quad \lambda_i' = \lambda_i + \text{random} \\
&\quad \text{end} \\
&\quad \text{if } E(\lambda') \leq E(\lambda_{\text{current}}) \text{ then} \\
&\quad\quad \lambda_{\text{current}} = \lambda' \\
&\quad \text{else if } e^{\frac{(E(\lambda_{\text{current}}) - E(\lambda'))}{k_B \text{Temp}}} \geq \text{random}(0, 1) \text{ then} \\
&\quad\quad\quad \lambda_{\text{current}} = \lambda' \\
&\quad\quad \text{Temp} = \alpha \cdot \text{Temp} \\
&\quad \text{end} \\
\text{return } \lambda_{\text{current}}
\end{align*}
\]

8.4.2.2 Genetic Algorithms

An alternative to the simulated annealing scheme presented above are genetic algorithms. They work based on the theory of evolution. While there have been many advancements in the field (see e.g. [41]), we will explain only the so-called simple GA to get a feeling for the type of approach. Here, we follow the explanation as provided in [41]. The basic starting point of the algorithm is a population of candidate solutions, in our case parameter vectors. In genetic algorithms these candidates are represented by means of binary strings (the so-called genotype). We can allocate a number of bits in the string per parameter value. The number of bits depends on the desired granularity. Assuming we use \( n \) bits to represent one parameter value, we end up with a genotype of length \( n \cdot |\lambda| \). An example of a bit string representing a parameter vector \( \lambda \) with two values, each represented by 4 bits, is shown in Figure 8.17.

![Figure 8.17: Parameter Vector genotype](image-url)
Genetic Algorithms (1)

- Second option is to optimize the parameters using genetic algorithms
- We consider the simple GA
- We encode our parameter values as bits (genotype), these are individuals

We create a whole population of these individuals

\[
\begin{array}{cccc}
\lambda_1 & 1 & 0 & 1 \\
\lambda_2 & 1 & 1 & 1 \\
\end{array}
\]
Genetic Algorithms (2)

- From the population, we select parents for a mating pool
- We perform crossover and mutation
- We select a new population for the next generation
- We continue this for a number of generations
Algorithm 21: Simple Generic Algorithm

population = random initialization of population with set population size ps

for $i$ from 1 to $\max{\text{generations}}$

- Select $ps$ parents according to equation 8.21
- Select pairs of parents from the individuals we have selected (without replacement)
- Apply crossover to the parents with probability $p_c$ or copy the original parents
- Apply bitwise mutation with probability $p_m$ on the result individuals
- The individuals we have just created become the new population

end

return fittest individual in the final population

8.4.2.3 Multi-Criteria Optimization

The niche of these types of models is however that they focus on the relationship between multiple target states. Often, trade-offs need to be made in the setting of the parameters: adjusting a parameter might improve the predictions on one target, while it worsens them for another target. If we assume all are equally important, we can just optimize the mean squared error between the target values using the methods we listed before. If we want to study the trade-off between the performance on each of the targets we have a multi-criteria optimization problem.

Fig. 8.19: Error score model instances with Pareto Front

If we return to our previous example, we could end up with a series of parameters settings for $g_1$ to $g_5$, where each setting results in an error on the mood target and an error on the activity level target. We call a model with these parameter settings a model instance. Hence, we have separate error functions per target:
Genetic Algorithms (4)

- Selection of individuals is done by a roulette wheel:

\[ P_i = \frac{(1 - E(\lambda_i))}{\sum_{j=1}^{\text{pop.size}} E(\lambda_j)} \]

- Crossover is straightforward (if we apply it to the parents), select a crossover point:

<table>
<thead>
<tr>
<th></th>
<th>crossover point</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent 1</td>
<td>1 0 1 1 0 1 1 1</td>
</tr>
<tr>
<td>parent 2</td>
<td>1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>child 1</td>
<td>1 0 1 1 0 1 0 1</td>
</tr>
<tr>
<td>child 2</td>
<td>1 1 0 1 0 1 1 1</td>
</tr>
</tbody>
</table>

- Mutation is done per bit, with a probability \( p_m \)
Multi-Criteria Optimization

• What if we have multiple targets?
• We could just take the average error over all targets (in fact, we have done so)
• We might however be interested in the trade-off between different targets
• This is then a multi-criteria optimization problem
• We call a dynamical systems model with certain parameter values $\lambda$ a *model instance*
Algorithm 21: Simple Generic Algorithm

\begin{align*}
\text{population} &= \text{random initialization of population with set population size } ps \\
\text{for } i \text{ from 1 to max generations } &\text{ do} \\
\quad \text{Select ps parents according to equation } 8.21 \\
\quad \text{Select pairs of parents from the individuals we have selected (without replacement) } \\
\quad \text{Apply crossover to the parents with probability } p_c \\
\quad &\text{or copy the original parents } \\
\quad \text{Apply bitwise mutation with probability } p_m \text{ on the result individuals } \\
\quad \text{The individuals we have just created become the new population} \\
\text{end} \\
\text{return fittest individual in the final population}
\end{align*}

8.4.2.3 Multi-Criteria Optimization

The niche of these types of models is however that they focus on the relationship between multiple target states. Often, trade-offs need to be made in the setting of the parameters: adjusting a parameter might improve the predictions on one target, while it worsens them for another target. If we assume all are equally important, we can just optimize the mean squared error between the target values using the methods we listed before. If we want to study the trade-off between the performance on each of the targets we have a multi-criteria optimization problem.

Fig. 8.19: Error score model instances with Pareto Front

If we return to our previous example, we could end up with a series of parameters settings for \( g_1 \) to \( g_5 \), where each setting results in an error on the mood target and an error on the activity level target. We call a model with these parameter settings a model instance. Hence, we have separate error functions per target:
Pareto efficiency (2)

• We assume an error function per target

\[ E_{\text{target}}(\lambda) = \sum_{t=1}^{N} (\hat{y}_{\text{target}}(t) - y_{\text{target}}(t))^2 \]

• We want to look for non-dominated instances

**Definition 8.2.** A model instance \( \lambda_m \) is dominated by another model instance \( \lambda_n \) when the mean squared error obtained by model instance \( \lambda_n \) is lower for at least one target and not higher for any of the other targets.

• Formally:

\[
\text{dominated}(\lambda_m, \lambda_n) = \begin{cases} 
1 & \exists i \in 1, \ldots, q : E_i(\lambda_m) > E_i(\lambda_n) \land \\
\forall j \in 1, \ldots, q : E_j(\lambda_m) \geq E_j(\lambda_m) & 0 \quad \text{otherwise}
\end{cases}
\]
The NSGA-II algorithm can find these non-dominated model instances

NSGA = Non-Dominated Sorting Genetic Algorithm

It has a population of solutions (like the simple GA)

We create fronts from the population
After we have passed all model instances in our population we have our first front, although we obviously do no know how optimal it is. We then continue with the remaining model instances and look for a next collection of them that together form a new one. Great, so we have a set of Pareto Fronts $F_1, \ldots, F_f$. Another aspect is how well solutions are spread across our Pareto Front. We would like to have a collection of solutions that are nicely spread across the front otherwise we are not quite sure on the shape of the front, and the expert will not be able to select a model that represents specific trade-offs well. To establish this, we first need to determine the distance between points in our fronts. This is performed using calculation expressed in Algorithm 23. In the algorithm, we see that points in the Pareto Front are initialized to a zero distance. Then, for each objective they are sorted based on their error for that objective. Points that score highest or lowest on the objective are set to an infinite distance as they are the most extreme points on the front that we want to keep for sure. This is because we want have the maximum width of our front and well will see that bigger distance means a higher chance of being selected. Distance between the other points is the difference between the error for the two adjacent points.

**Algorithm 22: Finding Pareto Fronts**

```plaintext
find_pareto_fronts(P):
    used = []
    i = 0
    F = []
    while $|P| > 0$ do
        $P' = P[1]$ // The first model instance in the population
        for $p \in P \land p \notin P'$ do
            $P' = P' \cup \{p\}$
            for $q \in P' \land p \neq q$ do
                if dominated($q, p$) then
                    $P' = P' \setminus \{q\}$
                else if dominated($p, q$) then
                    $P' = P' \setminus \{p\}$
            end
        end
        Add $P'$ to $F$
        $P = P \setminus P'$
        $i = i + 1$
    end
    return $F$
```

We are finally ready to look at the overall algorithm, which is expressed in Algorithm 24. This shows how we proceed from one generation to the next. The initialization can be done in a random way. In the algorithm, a combined population of the parents and children is created. From these we identify the Pareto fronts following the function we have previously seen. We then create a new population of parents which we are going to fill. We start with the first (and most dominant) Pareto Front we have identified, and add the complete set of points that span up the front.
• We would like to get a nice spread of solutions across the Pareto Front
• We compute the distance of points to other points in the front and sort the points based in this distance (highest distance first)
• Points on the boundary are set to infinite distance
Algorithm 24: NSGA-II main loop

\[ \begin{align*}
R_t &= P_t \cup C_t \quad // \text{Take the parent and child population} \\
F_1, \ldots, F_f &= \text{find_pareto_fronts}(R_t) \\
P_i &= \emptyset \\
i &= 1 \\
\textbf{while } |P_{t+1}| < |P_t| \quad \textbf{do} \\
\quad \textbf{if } |P_{t+1}| - |P_t| \geq |F_i| \quad \textbf{then} \\
\quad\quad P_{t+1} &= P_{t+1} \cup F_i \\
\quad\quad i &= i + 1 \\
\quad \textbf{else} \\
\quad\quad d &= \text{distance_assignment}(F_i) \\
\quad\quad F_{\text{sorted}} &= \text{sort}(F_i, d) \\
\quad\quad P_{t+1} &= P_{t+1} \cup \{F_{\text{sorted}}[1]\} \cup \cdots \cup \{F_{\text{sorted}}[|P_{t+1}| - |P_t|]\} \\
\textbf{end} \\
\text{Create } C_{t+1} \text{ using crossover and mutation} \\
t &= t + 1
\end{align*} \]

This ends our part on dynamical systems models. Note that more complex functions to define the best possible model have been defined as well, see e.g. [24].
Case study

• Let us move to the CrowdSignals again
• Recall from last time: how did we tune the parameters?
  – Cross validation
  – Does that make sense now?
  – Nope it does not…..

7.10.2 Regression: predicting the heart rate

The next task we are going to consider is to predict the heart rate of the user based on all other measurements, including the label. This is a regression task since we are aiming to predict a continuous value. We will address this as a temporal task: we will learn on the first part of the data (in terms of time) and predict for the remaining part of the data. Of course the approaches we use from this chapter will not exploit the ordering of the data, but we still have the temporal features we have identified in Chapter 4. We will go through each of the steps in somewhat less detail compared to our previous classification problem since the approach is very similar. Instead of the accuracy for our previous problem we will use the mean squared error as a performance metric (i.e. the lower the better).

7.10.2.1 Preparing the dataset for learning

As said, we are going to address this problem as a temporal learning problem. We take an interval from the starting point of our measurements to about halfway (totalling to 1422 data points for training). We have selected this point because nearly all activities (that are likely to be highly influential on the heart rate) are reflected in this training period. As a test set we select the interval following our training interval. We do however not continue all the way to the end point of our dataset, but stop when we encounter an activity we have not seen before (and are unable to train on), namely running. We understand this might seem a bit artificial but we need to make sure that we have a representative test set as well. If we would have a lot more data there would be no need to be picky. The test set contains 736 instances. Figure 7.17 illustrates the selected training and test set. In our regression case we do not remove any additional time points since the heart rate is known everywhere or has been interpolated at least.
Algorithms used

• What techniques do we use?
  – Note: normalize values to [0,1] and [-0.9,0.9]/[0.1, 0.9]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Variant description</th>
<th>Parameters varied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Echo State Network (ESN)</td>
<td>Randomly connected reservoir of neurons with tanh activation function with the output being fed back into the reservoir</td>
<td>Number of neurons in reservoir: {500, 1000, 5000}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \alpha: {0.4, 0.6} )</td>
</tr>
<tr>
<td>Recurrent Neural Network (RNN)</td>
<td>Recurrent neural network with one layer of hidden neurons with a sigmoid activation function and sigmoid output nodes</td>
<td>Number of hidden neurons: {50, 100}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>maximum iterations over the entire dataset: {250, 500}</td>
</tr>
<tr>
<td>Time series</td>
<td>ARIMAX algorithm using Bayesian inference</td>
<td>p: {0, 1, 3, 5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>q: {0, 1, 3, 5}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>d: {0, 1}</td>
</tr>
</tbody>
</table>
Time series

• Autocorrelations we observe:

![Autocorrelation plot](chart)

• Stationary time series (Dickey-Fuller test)
Results

• Results:

![Graph showing performance of different algorithms](image-url)
Results (2)

• Time series:

![Image of time series graph showing training and test sets](image-url)
Results (3)

- Time series feature importance:

<table>
<thead>
<tr>
<th>Approach</th>
<th>$\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MA(1)$</td>
<td>1.8606</td>
</tr>
<tr>
<td>$MA(2)$</td>
<td>2.2943</td>
</tr>
<tr>
<td>$MA(3)$</td>
<td>2.0588</td>
</tr>
<tr>
<td>$MA(4)$</td>
<td>1.2991</td>
</tr>
<tr>
<td>$MA(5)$</td>
<td>0.4741</td>
</tr>
<tr>
<td>$acc_phone_x$</td>
<td>-0.2286</td>
</tr>
<tr>
<td>$acc_phone_y$</td>
<td>-0.3136</td>
</tr>
<tr>
<td>$acc_phone_z$</td>
<td>0.1977</td>
</tr>
<tr>
<td>$acc_watch_x$</td>
<td>0.1259</td>
</tr>
<tr>
<td>$gyr_phone_y$</td>
<td>-0.1105</td>
</tr>
<tr>
<td>$gyr_phone_z$</td>
<td>-0.1282</td>
</tr>
<tr>
<td>$label_OnTable$</td>
<td>0.2781</td>
</tr>
<tr>
<td>$label_Sitting$</td>
<td>-0.1099</td>
</tr>
<tr>
<td>$mag_phone_y$</td>
<td>-0.1317</td>
</tr>
<tr>
<td>$press_phone_pressure$</td>
<td>0.1173</td>
</tr>
</tbody>
</table>
Results (4)

• Recurrent Neural Network:

![Graph showing performance of different algorithms and actual versus predicted values for RNN](image-url)
Results (5)

- Echo State Network: