Supplementary material

A List of abbreviations

Abbreviation	Meaning
AI	Artificial Intelligence
CCA	Canonical Correlation Analysis
EPA	Equal predictive ability
FFNN	Feed Forward Neural Network
GLM	Generalized Linear Model
GNM	Generalized non-linear model
GRU	Gated Recurrent Unit
LC model	Lee-Carter model
LSTM	Long-Short Term Memory
MCS	Model Confidence Set
PCA	Principal Component Analysis
RF	Random Forest
RNN	Recurrent Neural Network
SVD	Singular Value Decomposition
SVM	Support Vector Machine
XGB	Extreme Gradient Boosting

B Multi level factor model

Choi et al. (2018) assumes the following about the global and country factors

(i) $\{G_t\}$, $\{F_{1t}\}$,..., $\{F_{Mt}\}$ are zero-mean, stationary-processes that satisfy the conditions for the law of large numbers and central limit theorem.

(ii) $\{G_t\}$, $\{F_{1t}\}$,..., $\{F_{Mt}\}$ are uncorrelated. That is, $E\left[F_{i,t}F'_{m,t}\right] = 0$ for all t and $i \neq m$ and $E\left[G_tF'_{i,t}\right] = 0$ for all t and i.

(iii) $\{G_t\}$, $\{F_{1t}\}$,..., $\{F_{Mt}\}$ satisfy conditions for the law of large numbers and the central limit theorem applied to their self- and cross-products.

(iv) $E\left[e_ie'_m\right] = 0$ for every i and m with $i \neq m$.

(v) $N_1, ..., N_M$ and $N = N_1 + ... + N_M$ are of the same order of magnitude.

Without assuming that the factors are uncorrelated, then identification of the global factors using CCA would not be possible. Assumption (iv) implies that the error terms of each population used in the model are uncorrelated. The last assumption means that no population can have a dominantly large or small number of individuals (in this case age groups). The model also requires that the number of populations M is finite.

Model estimation is as described earlier set-up in the following way (superscript is used to denote the iteration number).

1. Select two countries and obtain the global factor estimator $\hat{G}_t^{(1)}$ using CCA.

2. Find the respective factor loadings $\hat{\gamma}_{x,i}^{(1)} = \left(G^{(1)\prime}G^{(1)}\right)^{-1} G^{(1)\prime}Y_i$ for the global factor using data for each of the populations $i = 1, ..., M$. The resulting vector will be of dimensions $(N_i \times 1)$ where N_i is the number of age-groups for population i.

3. Estimate the country factors $\hat{F}_{i,t}^{(1)}$ and loadings $\lambda_{x,i}^{(1)}$ using SVD on the residuals $Y_{x,i,t} - \hat{\gamma}_{x,i}^{(1)\prime}\hat{G}_t^{(1)}$.

Choi et al. (2018) suggest using the two countries that yield the maximum sample mean of their eigenvalues in step 1. For all subsequent iterations $(j = 2, ...)$, step 1 will be replaced with

1. Estimate the global factor using SVD on the residual matrix $Y_{x,i,t} - \hat{\lambda}_{x,i}^{(j)\prime}\hat{F}_{i,t}^{(j)}$, where each populations residual vector is stacked.

Choi et al. (2018) shows that the factors are consistently estimated using this strategy. They also show through a simulation study that the performance of the model is better when the second iteration is included because the global factor will be estimated using more information.

The number of static factors must also be determined and there are several ways of doing this. The multi-level factor model assumes that the number of global factors is predetermined, and the number of country factors is found using information criteria after eliminating the information from the global factor. Choi et al. (2018) consider the following information criteria

- $IC_{p2} = \ln\left(V_i\left(k_i, F_{k_i}^{(1)}\right)\right) + \ln\left(\min\{N_i, T\}\right)\left(\frac{k_i(N_i+T)}{N_i T}\right)$
- $BIC = T \cdot tr\left(\ln\left(\frac{1}{T}Y_i^{G\prime}M_{\tilde{F}_{k_m}^{(1)}}Y_i^G\right)\right) + \ln\left(N_i T\right)\left[k_i\left(N_i + T\right) + N_i\right]$
- $HQ_c = T \cdot tr\left(\ln\left(\frac{1}{T}X_i^{G\prime}M_{\tilde{F}_{k_m}^{(1)}}\right)\right) + c\ln\left(N_i T\right)\left[k_i\left(N_i + T\right) + N_i\right]$

where $V_i\left(k_i, F_{k_i}^{(1)}\right)$ is the sum of squared residuals for country i in step 2 when k_i country factors are estimated by $\tilde{F}_{k_m}^{(1)}$ and $tr\left(\cdot\right)$ is the trace function. In HQ_c there is a constant c that needs to be chosen, which will determine the trade-off between fit and number of parameters. Choi et al. (2018) note that previous studies have found BIC and HQ_c to perform well in finite samples. It is also found that IC_{p2} overestimates the number of factors when there is strong serial correlation. BIC and HQ_c tend to underestimate when the number of cross-section observations N_i is small[1]. BIC will be applied in this paper, as it is found to perform the best among the information criteria in Choi et al. (2018). The multi-level factor model will be estimated separately for each gender and the total population, using information from all the included countries.

C Machine learning forecasting strategies

In general, there are two main ways of doing time-series forecasts with machine learning models: the recursive and the direct forecasting method. The recursive forecasting method will set-up a model that is estimated with the output variable being the one-period ahead value. Thus, when multi-step ahead forecasting is performed, input values will be forecasts. If the

[1] $N_i \leq 20$, which is not the case in this application.

one-step ahead model is defined as $\hat{f}$, which is only a function of previous output values, then forecasting are given by

$$\hat{y}_{T+h} = \begin{cases} \hat{f}\left(y_T, ..., y_0\right), & \text{if } h = 1 \\ \hat{f}\left(\hat{y}_{T+h-1}, ..., \hat{y}_{T+1}, y_T, ..., y_{h-1}\right) & \text{if } h \in (2, ..., T) \\ \hat{f}\left(\hat{y}_{T+h-1}, ..., \hat{y}_h\right) & \text{if } h > T \end{cases} \tag{1}$$

Therefore, at some point, all input values can be forecast values. The recursive strategy is sensitive to the accumulation of errors with the forecast horizon. Errors will propagate forward because the forecasts are used to determine subsequent forecasts (Taieb et al., 2012). Direct forecasting specifies a forecasting model for each step-ahead forecasting period. This means that a model will be estimated to do one-step ahead forecasting, a model will be estimated to do two-step ahead forecasting, and so forth. Therefore, h models will be needed for h-step ahead forecasting. The direct forecasting method is, for that reason, more computational as it requires a model for each step-ahead forecasting period, but will also be less biased. Using direct forecasting can lead to irregularities in the forecasting function, as the individual step-ahead models can be quite different (Taieb et al., 2012). The recursive forecasting method is used in this paper, such that forecasted values will be used for multi-step ahead forecasts.

D Neural network details

D.1 Feed forward neural networks

D.1.1 Backpropagation

All outputs of a layer can be calculated given the information from the previous layers, and therefore the weights can also be optimized in this manner. The derivatives are calculated starting from the output and going back to the input layer, and then all weights are updated. Optimization in a neural network is done by minimizing a loss function $L\left(\hat{y}, y\right)$. The loss function in the regression problem is typically the mean squared error loss function. The backpropagation optimization algorithm use chain rule derivatives with respect to each parameter to find the gradient descent direction, which minimizes the loss. The loss surface is non-convex, and therefore it is not ensured that backpropagation will reach global minimum. The result, therefore, depends on the initialization of the parameters. First, the derivative with respect to the weights in the output layer is calculated and then so on for the previous layers. After calculating all the gradients, then the weights are updated in the direction that minimizes the loss. Updates usually happen in batches of data, such that a sample of the data is used to update the weights in each update iteration. The first step is to compute the derivative of the loss with respect to the output o, $\frac{\partial L}{\partial o}$. Subsequently, the derivatives are calculated in a backward direction through the network using chain rule derivatives. The parameter from the hidden layer h_{r-1} to h_r denoted as $w_{r,r-1}$ will have the chain rule derivative that is given as

$$\frac{\partial L}{\partial w_{r,r-1}} = \underbrace{\frac{\partial L}{\partial o} \cdot \left[\sum_{[h_r, h_{r+1}, ..., h_k, o] \in \mathcal{P}} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\Delta(h_r, o) = \frac{\partial L}{\partial h_r}} \frac{\partial h_r}{\partial w_{r,r-1}}. \tag{2}$$

The partial derivative is therefore computed by aggregating the product of the partial derivatives from all paths from h_r to the output o. The value h_r is the post-activation value $h_r = a_h\left(x_{h_r}\right)$, and x_{h_r} is the input that goes into the hidden layer h_r. The set of paths to the output is denoted as $\mathcal{P}$. The value of $\Delta\left(h_r, o\right) \frac{\partial L}{\partial h_r}$ is what is calculated in the backpropagation update, as the last part can be calculated beforehand which is the derivative of the weight with respect to the activation function. This

quantity can be calculated as

$$\Delta\left(h_r, o\right) = \frac{\partial L}{\partial h_r} = \sum_{h:h_r \Rightarrow h} \frac{\partial L}{\partial h} \frac{\partial h}{\partial h_r} = \frac{\partial h}{\partial h_r} \Delta\left(h, o\right), \tag{3}$$

where $h : h_r$ denotes all the subsequent layers from h_r. When computing derivatives for layer h_r then all the subsequent layers h will already be calculated when evaluating $\Delta\left(h_r, o\right)$. What is calculated here is therefore $\frac{\partial h}{\partial h_r}$. The layer $h = a_h\left(x_h\right)$ transform the inputs x_h, which is a linear combination of the previous units through the activation function a_h. This means that the amount $\frac{\partial h}{\partial h_r}$ is computed as,

$$\frac{\partial h}{\partial h_r} = \frac{\partial h}{\partial x_h} \frac{\partial a_h\left(x_h\right)}{\partial x_h} = \frac{\partial a_h\left(x_h\right)}{\partial x_h} w_{h_r,h} \tag{4}$$

The backward recursion is linear in the number of edges between units in the network, which is a useful feature that ensures that computation is feasible (Aggarwal, 2018). The negative gradient indicates the direction of the steepest descent in the loss-surface, where the error on average is low (LeCun et al., 2015; Lipton et al., 2015). Backpropagation can use dynamic programming to increase speed, as the derivatives for a layer can be used for all the preceding layers (Aggarwal, 2018). Backpropagation updating is done in batches of observations to make updates more feasible. After the whole dataset has been used in batches, this is called an epoch. The backpropagation is then run for an appropriate amount of epochs until the parameter values have converged (Zhang et al., 2020).

D.1.2 Gradient descent strategy

It is normal to use the steepest descent algorithm, where the update happens with respect to the steepest descent of the gradient. Data usually is, as in this paper, updated in batches using stochastic gradient descent, which updates parameters for a sample of observations at each update. Therefore, the model parameters are first initialized, typically at random, with values around zero. Then model parameters are updated iteratively using batches of the data for each layer starting backward at the output layer, updating the parameters in the negative gradient direction. The gradients are calculated with respect to all batch observations, to add up their local gradient and execute them for all of these observations at once (Aggarwal, 2018). Stochastic gradient descent is a greedy algorithm, as it only considers the steepest direction for the sample of observations that it is currently updating.

It is a good idea to use a decaying learning rate for the updating mechanism in the backpropagation algorithm. The learning rate measures how much of the new update is added to the current parameter values. Starting with a high learning rate will make the algorithm go faster towards optima, and the decay will ensure that it does not diverge at a later point. The decaying learning rate is used in this paper, where the learning rate decreases with a certain factor when the backpropagation algorithm does not improve the loss for a certain amount of epochs (Aggarwal, 2018). Using a constantly high learning rate leaves the derivatives oscillating around the optima that it has found. Instead, one can use parameter-specific learning rates that are specific to each parameter. It is done because parameters with large partial derivatives often are oscillating, and parameters with small partial derivatives tend to be more consistent in the movement towards optima (Aggarwal, 2018). The popular Adam gradient descent strategy is well suited for stochastic gradient descent and is used in this paper. A_i is the exponential weighted average value of the $i'th$ parameter

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2, \qquad \forall i. \tag{5}$$

Where ρ is a decaying parameter. Smoothing is performed with a different decaying parameter ρ_f in

$$F_i \Leftarrow \rho_f F_i + (1-\rho_f)\left(\frac{\partial L}{\partial w_i}\right), \qquad \forall i. \tag{6}$$

The Adam update is then given by

$$w_i \Leftarrow w_i - \frac{\alpha_t}{\sqrt{A_i} + \epsilon} F_i, \qquad \forall i, \tag{7}$$

where α_t is the learning rate at update iteration t (Aggarwal, 2018; Kingma and Ba, 2014). It is smoothed by using the previous values of itself. By dividing with the squared gradient A_i, then it is ensured that the weight update gets smaller and smaller, ensuring that an optimum is not missed. When it oscillates a lot, then A_i is large, and the updates are reduced.

Neural networks often face problems with overfitting, meaning that the algorithm is not able to generalize accurately, fitting too much to the estimation data. To try and overcome overfitting in a neural network, regularly dropout is used.

D.1.3 Dropout

Dropout tries to make the network spread the weight out on all the features, rather than focusing on a few, potentially spurious, relations in the data (Zhang et al., 2020). Dropout holds out a portion of the neurons during training. Dropout is generally specified for each layer during training, where the probability for a neuron to be dropped in the given layer is determined by a dropout probability p_j (Zhang et al., 2020). If a node is dropped during training, then all incoming and outgoing connections are dropped as well. The idea is much like the random forest algorithm, where only a subset of split-variables is considered at each split. For a neural network, it forces the algorithm to focus on all neurons rather than just a few. It is also possible to use batched normalization to stabilize optimization.

D.1.4 Batch normalization

Batch normalization is a technique that normalizes the inputs in each batch. The neural network is updated using batches of data, and to make these inputs more stable, the inputs are normalized in each batch. It transforms the inputs of a batch to have zero mean and unit variance. The parameters change during training, as only a batch of the data is seen in each update. When the inputs are normalized using this technique, then the derivative is better behaved. It is less likely to be stuck in a local optimum and the gradient vanishing/exploding. Covariate shifts between batches are reduced, and the network converge faster (Ioffe and Szegedy, 2015).

D.1.5 Embeddings

Another way of transforming the input space is by using embeddings. Richman and Wüthrich (2018) found that using embedding connections improved performance for neural networks in the mortality forecasting setting. This paper finds similar results, where performance is improved by using embeddings to transform the feature-space of the categorical variables. Categorical variables are often modeled as dummy-variables when input into machine learning models. It transforms the input into a sparse vector of binary dummy variables. This can create problems in estimation if there are not representations for some of the values to learn the relationship between the output and the given category. One can instead transform the categorical variables using embeddings. Embeddings are continuous representations of discrete variables. Embedding vectors maps the discrete variable on a continuous scale, mapping the relationship between the categories. It can be the relationship between different ages if the embeddings are modeled on the age-dimension in the mortality forecasting framework. The relationships are learned during optimization using skip-gram connections. The skip-gram connections take a single input of w and output m context outputs. The goal is for the skip-gram connection to estimate $P(w_1, w_2, ..., w_m \mid w)$ (Aggarwal, 2018). Skip-grams use the context of inputs appearing to have closely related values to help reduce dimensionality. Using embeddings for each categorical feature allows a dense representation to be learned for each feature before being combined in the neural network (Richman, 2018). It gives a way of learning patterns for discrete features better, as they are learned in

groups rather than each category of the variable. Skip-gram embedding layers are used throughout the deep learning models as they are found to improve the models. The skip-gram connection method can be viewed as an application of exponential-family principal components analysis (EPCA) to an integer matrix of co-occurrence counts (Cotterell et al., 2017). Skip-gram is EPCA using the multinomial distribution and log link function. When using embeddings in a neural network, the values are updated during backpropagation to optimize with respect to the loss function. In the mortality rate forecasting problem, it means the embeddings are finding ages, countries, genders, and cohorts related to the same mortality rates.

D.2 Recurrent neural networks

D.2.1 Backpropagation through time

An assumption in the backpropagation algorithm is that weights in different layers are distinct from one another. The backpropagation through time starts by assuming that parameters in the temporal layers are independent of each other. After calculating the gradient for each of these parameters, the contribution from different time-steps is added to create a unified update for each weight parameter, thereby accounting for the temporal dependence. Therefore, the backpropagation through time algorithm starts as the normal backpropagation by computing the loss. Then, the gradient for each weight parameter is calculated, treating all parameters in each temporal layer as unique. As with backpropagation, this process starts backward from the last period. Finally, all shared weights are added to compute the parameter update as,

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial W_{xh}^{(t)}} \tag{8}$$

This procedure is very similar to the backpropagation algorithm but is adjusted to accommodate for the temporal dimension (Aggarwal, 2018). This can be a very computational process if the time-dependencies are very long, and therefore usually, a truncated backpropagation through time algorithm is used. This algorithm updates only with certain lookback through time, such that the whole sequence is not used, but only a certain amount of time-steps back.

A problem that is sometimes seen for regular RNN models is that the long-range dependencies are hard to optimize, as the gradient explodes or vanishes (Bengio et al., 1994; Chung et al., 2014; Lipton et al., 2015). It happens because the long-term dependencies are hidden by the effect of short-term dependencies. To overcome this problem, often gated recurrent neural networks are used instead. The gradient of the sigmoid function is never larger than 0.25, and thus long-range dependencies will quickly have the gradient vanish. The ReLu activation function is less prone to having the gradient vanish when the input values operate in the region where the gradient is equal to one for this activation function (Aggarwal, 2018). The gradient descent updating mechanism is made for infinitesimally small steps, and the gradient can vanish for long dependencies or explode depending on the size of the gradient. The LSTM and GRU are gated recurrent neural networks proposed to overcome the challenges of regular RNN models.

D.2.2 Long Short Term Memory (LSTM) neural network

LSTM hidden layers were introduced by Hochreiter and Schmidhuber (1997), which introduces a memory cell to replace each unit in the layer. The instability caused in RNN models is the direct result of successive multiplication of the weight matrix at various time-steps (Aggarwal, 2018).

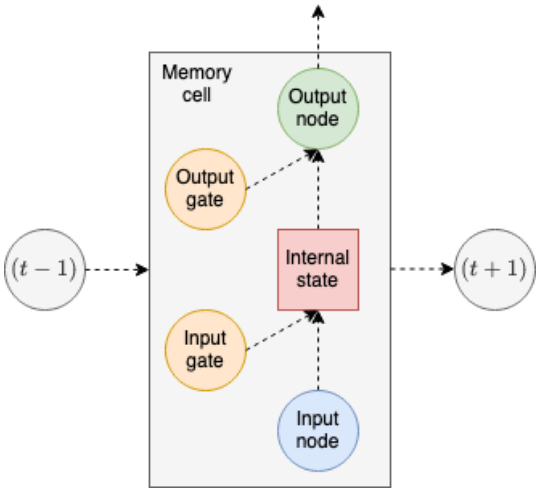

Figure 1: LSTM memory cell structure.

The memory cell was introduced to overcome the vanishing gradient problem. Each memory cell contains a self-connected time weight that is fixed to a value of one. This ensures that the gradient is constant, and therefore does not vanish nor explode. The name is derived from the fact that the network has a long memory by using information from previous time-steps and short memory from previous information in the network at the same time-step. The structure of the memory cell can be seen in fig. 1. In the memory cell structure from fig. 1, the memory cell retains long-term information by updating the state in increments from previous states. This ensures that the successive multiplication problem is alleviated. If the states in different time-steps share a higher level of similarity through this persistence, then it is also harder for the gradient to be drastically different (Aggarwal, 2018).

Input nodes take inputs $x^{(t)}$ and the values from the hidden layer at the previous time-step $h^{(t-1)}$, and transforms it using an activation function

$$g^{(t)} = a\left(W^{gx}x^{(t)} + W^{gh}h^{(t-1)} + b_g\right). \tag{9}$$

Input gates is a weight between zero and one for each of the input node inputs, using a sigmoid activation function. Given a value of zero for a particular input then that means that the information from that input will be cut-off. A value of one, on the other hand, means that all information from that input will be used.

$$i^{(t)} = \sigma\left(W^{gx}x^{(t)} + W^{gh}h^{(t-1)} + b_g\right). \tag{10}$$

Internal states has a self-connected recurrent edge with a weight of one. This means that information will flow between time-steps without the gradient exploding or vanishing. It collects the information from the input gate, the input node, and the previous internal state.

$$s^{(t)} = g^{(t)} \odot i^{(t)} + f^{(t)} \odot s^{(t-1)}, \tag{11}$$

where $\odot$ is point-wise matrix multiplication and $f^{(t)}$ is a forget-gate. Forget gates work much like the input-gate and weights each of the inputs coming from the previous internal state between zero and one.

Output gates weights the output of the internal state, just like the input gate did with the input from the input node. Ultimately $h^{(t)}$ is the output produced by the memory cell in the output node. The output gate produces the weight that is multiplied onto the value of the internal state to produce $h^{(t)}$.

Output node produces the final value of the memory cell that is used much like other neurons in a normal neural network.

$$h^{(t)} = a\left(s^{(t)}\right) \odot o^{(t)}, \tag{12}$$

where $o^{(t)}$ is the vector of weights from the output gate (Lipton et al., 2015).

The partial derivative of the internal state from eq. (11) with respect to the previous value is going to be the forget gate value. They are often initially high, which means the gradient decay will be slow (Aggarwal, 2018). Like the LSTM neural network, gated recurrent unit (GRU) networks were also proposed to overcome the gradient challenges of the normal RNN.

D.2.3 Gated recurrent unit (GRU) neural network

GRU neural networks were proposed by Cho et al. (2014,?) to overcome the challenges of a normal RNN. The main difference from an RNN is the way that it incorporates gating into the hidden state variable that allows for time-dependencies. It thereby includes a mechanism that tells the network when to update the time-dependencies and when to reset them. Thus, it can keep important information stored and learn to ignore irrelevant temporary information. The GRU hidden layer will replace each node with a GRU cell just like the LSTM did with the memory cell. The GRU cell structure can be seen in fig. 2.

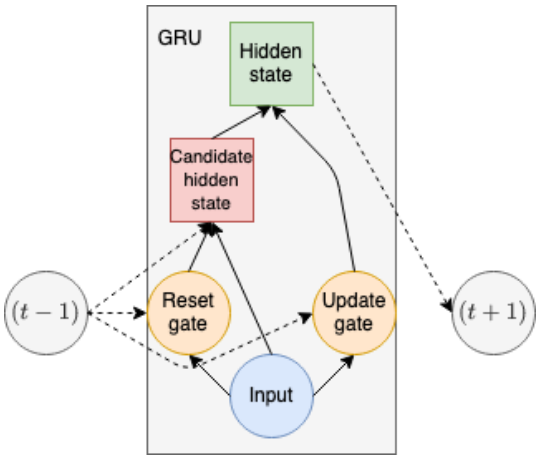

Figure 2: GRU cell structure.

The GRU cell takes in inputs and hidden state value from the previous period. These go into the reset gate that outputs a weight between zero and one determining how much previous information should be kept. Likewise, these inputs are also used in an update gate that determines how much of the new information should be used. The update gate determines how much information from previous and current states needs to be passed along to the future. On the other hand, the reset gate is used to determine how much of the previous information needs to be forgotten. Thus, even though these two gates work in the same way, they do two different tasks. The reset gate is calculated as

$$R^{(t)} = \sigma\left(X^{(t)}W^{rx} + H^{(t-1)}W^{rh} + b_r\right), \tag{13}$$

and the update gate as

$$Z^{(t)} = \sigma\left(X^{(t)}W^{zx} + H^{(t-1)}W^{zh} + b_r\right). \tag{14}$$

The candidate hidden state is calculated using a hyperbolic tangent activation function as,

$$\tilde{H}^{(t)} = \phi\left(X^{(t)}W^{hx} + \left(R^{(t)} \odot H^{(t-1)}\right)W^{hh} + b_h\right). \tag{15}$$

Afterward, the candidate hidden state and the update gate is used to compute the hidden state of the node at time t,

$$H^{(t)} = Z^{(t)} \odot H^{(t-1)} + \left(1 - Z^{(t)}\right) \odot \tilde{H}^{(t)} \tag{16}$$

Thus, as it can be seen from eq. (16), the update gate determines to which extend the current hidden state is from the previous hidden state and how much of the current information is used. When the update gate $Z^{(t)}$ is close to one, then this essentially means that the old state is retained and the current inputs $X^{(t)}$ is ignored.

GRU differs from LSTM in that it does not have an input gate that controls how much of the current state that the internal hidden state is exposed to. Similarly, the GRU network does not have an output gate either, which controls how much the rest of the network is exposed to the information from the recurrent unit cell (Chung et al., 2014). Here the GRU unit exposes the rest of the network to the full information. As with LSTM, GRU also stores a partial copy of the previous hidden states, which makes the gradient flow more stable during backpropagation (Aggarwal, 2018).

E Model confidence set

The EPA test statistic is calculated for an arbitrary loss function that satisfies weak stationarity conditions. This allows the procedure to be applied on different grounds, such as the predictive ability as done in Hansen and Lunde (2005) and in this paper, or it could be the goodness-of-fit as done in Hansen et al. (2011). Bollerslev et al. (1994); Hansen and Lunde (2005) points towards the natural choice of MSE when the loss is not defined with respect to variances. One could also use mean absolute error (MAE) that is more robust to outliers. The loss for the i'th model at time t can formally be defined as

$$l_{i,t} = L\left(Y_t, \hat{Y}_{i,t}\right) \tag{17}$$

The procedure starts with the initial set of models $\hat{M}^0$, and for a given confidence level $1-\alpha$, it will deliver a smaller set of models, that is the superior set of models $\hat{M}^*_{1-\alpha}$ with $m^* \leq m$ models. The difference in loss between two models can be defined as

$$d_{ij,t} = l_{i,t} - l_{j,t}, \qquad i,j = 1,...,m, \qquad t = 1,...,T, \tag{18}$$

and the simple difference of model i relative to the other models at time t is given as,

$$d_{i\cdot,t} = \frac{1}{m-1} \sum_{j \in M} d_{ij,t}, \qquad i = 1,...,m. \tag{19}$$

The EPA hypothesis for a given set of models M can be formulated as

$$\begin{aligned} H_{0,M} &: c_{ij} = 0, \qquad \text{for all } i,j = 1,...,m \\ H_{A,M} &: c_{ij} \neq 0, \qquad \text{for some } i,j = 1,...,m, \end{aligned} \tag{20}$$

or as

$$\begin{aligned} H_{0,M} &: c_{i\cdot} = 0, \qquad \text{for all } i = 1,...,m \\ H_{A,M} &: c_{i\cdot} \neq 0, \qquad \text{for some } i = 1,...,m, \end{aligned} \tag{21}$$

where $c_{ij} = E\left[d_{ij}\right]$ and $c_{i\cdot} = E\left[d_{i\cdot}\right]$ (Bernardi and Catania, 2015).

Then the following tests are constructed to test the hypothesis in eq. (20) and eq. (21) respectively,

$$t_{ij} = \frac{\bar{d}_{ij}}{\sqrt{\hat{var}\left(\bar{d}_{ij}\right)}}, \qquad t_{i\cdot} = \frac{\bar{d}_{i\cdot}}{\sqrt{\hat{var}\left(\bar{d}_{i\cdot}\right)}}. \tag{22}$$

Here $\bar{d}_{i\cdot} = \frac{1}{m-1}\sum_{j\in M}\bar{d}_{ij}$ is the loss of the i'th model relative to the average losses across the models in the remaining model set M. The number $\bar{d}_{ij} = \frac{1}{m}\sum_{t=1}^{T} d_{ij,t}$ measures the relative loss between the i'th and j'th models. The variances $\hat{var}\left(\bar{d}_{i\cdot}\right)$ and $\hat{var}\left(\bar{d}_{ij}\right)$ are the bootstrapped estimates of the respective variances. Hansen et al. (2011) suggests performing a block-bootstrap with 5000 resamples. The block length p is the maximum number of significant parameters obtained by fitting an $AR\left(p\right)$ process on all the d_{ij} terms.

The two test statistics in eq. (22) can be mapped to

$$T_{R,M} = \max_{i,j\in M}|t_{ij}|, \qquad T_{\max,M} = \max_{i\in M} t_{i\cdot}, \tag{23}$$

which can be used to test the EPA hypothesis. The test-distributions under the null hypothesis are found using bootstrap as their distributions are non-standard (Bernardi and Catania, 2015). The elimination rule is given as

$$e_{R,M} = \arg\max_{i}\left\{\sup_{j\in M}\frac{\bar{d}_{ij}}{\sqrt{\hat{var}\left(\bar{d}_{ij}\right)}}\right\}, \qquad e_{\max,M} = \arg\max_{i\in M}\frac{\bar{d}_{i\cdot}}{\sqrt{\hat{var}\left(\bar{d}_{i\cdot}\right)}}. \tag{24}$$

F Machine learning methods

Machine learning is a field within AI, where the algorithms are allowed to accumulate knowledge learned from estimation data (Goodfellow, 2016; Richman, 2018). Machine learning is broadly classified into three groups: supervised, unsupervised, and reinforcement learning. Supervised learning is the most popular group of models, where a model is trained to predict an output y (dependent variable), using a set of features X. I.e. the goal is to learn $f\left(X\right) = y$. These models span over well-known models such as generalized linear models (GLM), linear regressions, to deep neural networks. Unsupervised learning is the application of machine learning algorithms to find patterns and structures only within the X set of features. A well-known example here is the principal component analysis (PCA). Reinforcement learning is the part of machine learning, where an agent learns from rewards signals in real-time to improve decision making (Sutton, 1998). The mortality forecasting problem is a supervised learning problem when posed inside the machine learning domain. Next periods log-transformed mortality rates are the targets, and the features are year, age, gender, and current periods log mortality rate. Additionally, cohorts can pose as a feature. Mullainathan and Spiess (2017) summarises supervised machine learning as trying to optimize the following equation

$$\min L\left(f\left(X\right), y\right), \qquad \text{for all } f \in F, \qquad s.t. R\left(f\right) < c. \tag{25}$$

In eq. (25) it states that the goal is to minimize a loss function $L\left(\hat{f}\left(X\right), y\right)$ with respect to a certain model complexity level c. Therefore, machine learning focuses on the problem of prediction, i.e., to produce predictions of y from X. This is a fundamentally different problem than the one solved by many econometric models where the focus is on the parameter estimation, i.e., to produce parameters β that can explain the underlying relationship between y and X. There is a trade-off in parameter interpretability when one uses the more complex models rather than the simple linear regression models. Therefore, if one is willing to trade-off some model interpretability for more complexity and a larger focus on the predictions, there is a potential gain in using flexible machine learning models, such as the ones presented in this section and **??**. The parameters produced in the machine learning models are rarely consistent, and they often introduce bias into the models to lower the variance of the predictions (Mullainathan and Spiess, 2017).

A machine learning model must have very high representational capabilities to succeed in the mortality forecasting problem. The stochastic factor models are good at estimating the year and age mortality-pattern factors. The machine learning models presented in this thesis estimate the hidden patterns using non-linear transformations of the data. Especially the deep

learning algorithms of machine learning are known to have very strong representational abilities (Bengio et al., 2013). The machine learning algorithms presented in this section are more dependent on the input variables, where only age, year, country, and sub-population is available, and they might have limited success. To have high performance, the model must have a way of uncovering the more hidden correlations. As explained in **??**, it can be successfully done using neural network embeddings.

F.1 Random Forest (RF)

The random forest model is a tree-based algorithm derived from the bagging method. It uses an ensemble of decision trees made with bootstrap samples. The components of the random forest algorithm are explained below.

F.1.1 Decision trees

Decision trees can be applied to both regression and classification settings. The basic idea is that data is split based on the input features to get homogeneous subgroups. I.e. the predictor space is divided into J distinct and non-overlapping regions $(R_1, ..., R_J)$. Every observation that falls into the same region has the same output prediction. Typically in regression problems, such as the mortality forecasting problem, one would find the split regions using the residual sum of squares (RSS) as a loss function, i.e., the goal is to minimize

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} \left(y_i - \hat{y}_{R_j}\right)^2. \tag{26}$$

More specifically the split variable j and split point s that minimizes

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right], \tag{27}$$

where c_1 and c_2 is the prediction in region 1 and region 2, respectively. The prediction in a regression tree is most often given as the mean outcome of the observations in the region. As it is computationally infeasible to study all possible data-splits, a top-down greedy approach is used. For each sequential split, the algorithm only considers the current step and not all potential following steps (James, 2013). Each step chooses the single split of a feature that gives the largest reduction in the RSS. The splitting process is continued until some stopping criterion is met, which usually is that no region contains more than five observations. One can prune the trees using a cost-complexity loss function, which works much like the BIC information criteria, as it is a trade-off between fit and size of the decision-tree.

Comparing decision trees to a classic linear regression, it can be seen as an automatic way of looking at which interaction variables to include. In many problems, it is infeasible to study all possible interactions. Many interactions are often irrelevant to the problem, not providing any additional information. Decision trees are finding the relevant interactions automatically when minimizing the loss function (Mullainathan and Spiess, 2017).

F.1.2 Bootstrap aggregation (bagging)

Because regression trees are very variable and sensitive to the data, they are often used through the bagging technique. Bagging uses the average prediction of many decision trees to reduce the variance of the final prediction. Thus, the essential idea behind bagging is to average many noisy but approximately unbiased models to reduce the variance (Hastie, 2009). Averaging over B identically distributed (i.d.) models with pairwise correlation of ρ and individual variance σ^2 will give a variance of

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2. \tag{28}$$

As can be seen from eq. (28), increasing the number of bootstrap samples B makes the second term smaller. It means that the total variance improvement over using decision trees is limited by the pairwise correlation.

Each of the B bootstrap training sets is based on a decision tree, and the final bagging prediction is the average outcome of the individual bootstrap models. Outcomes are calculated for each bootstrap training set $\hat{f}^{*1}(x)$, $\hat{f}^{*2}(x)$, ..., $\hat{f}^{B}(x)$ using B bootstrap samples, and the outcome is given as

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x). \tag{29}$$

Each decision tree is not pruned in the bagging method, which causes each tree to have a lower bias. The decision trees are often correlated, which makes the effect of lowering the variance smaller. Averaging many highly correlated quantities does not lead to as substantial a variance reduction. The random forest extension to the bagging method is commonly preferred because it lowers the correlation between the decision trees (James, 2013). As mentioned above, and seen in eq. (28), the variance improvement was limited by the correlation between the decision trees, and thus being able to lower this correlation can give a larger variance reduction (Hastie, 2009). Therefore, the goal of the random forest is to lower the pairwise correlation by only allowing a subset of the features at each split, and thereby lowering the variance of the final model.

F.1.3 Random forest algorithm

As in bagging, the random forest algorithm also builds B decision trees using bootstrapped training samples. When building the decision trees, each time a split is considered, only a random sample of the p features is considered as splitting candidates. As a result, this causes the individual decision trees to be more different from each other, and thereby less correlated. A common choice is to choose the number of splitting candidates as $m = \sqrt{p}$ such that only a fraction of the available features is considered at each split (James, 2013). Comparing the random forest algorithm to the ridge regression, which is a regularized version of a linear regression, it can be noted that they are trying to do the same job. A ridge regression will add L2-penalization to linear regressions to lower the variance, at the cost of added bias. Random forest will also try to lower the variance to minimize the bias-variance trade-off, but the variance lowering technique differs from the ridge regression. Ridge regression penalizes high parameter values to lower the variance, whereas the random forest will use model averaging and decorrelation of the ensemble to lower the variance. Decision trees are much like linear regressions a relatively unbiased method but suffer from having high variance. Ridge regression and random forest seek to overcome this disadvantage of these methods. Opposite from the ridge regression, the random forest is a non-parametric method that works with the splitting rules (Mullainathan and Spiess, 2017). Regularization for random forest algorithms can also be provided through the aforementioned size of the trees. Smaller trees will be more robust at the cost of higher bias.

F.2 Boosting

Boosting is another way to improve the prediction results of decision trees. Boosting does not use bootstrap samples. Boosting grows the decision trees sequentially, such that each tree is using information from the previously grown trees. Instead of growing a large decision tree that overfits the training data, boosting will instead use weak learners to fit the data slowly (James, 2013). Thus, the idea is to combine outputs of many weak learners to form a powerful committee. A weak learner is a predictor whose error rate is only slightly better than random guessing, and the common choice is a decision tree with only a few splits. Boosting is a way of fitting an additive model, where each predictor is a regression tree (Hastie, 2009). Each tree is fit to the pseudo residuals of the current model calculated via the gradient. By fitting small trees to the residual, the model slowly improves in areas where it is not performing well. The loss function is minimized using the calculated first-order

gradients in the gradient boosting algorithm. Gradient boosting uses gradient descent to optimize directional movement. The gradient for the m'th iteration is given as,

$$-\hat{g}_m(x_i) = -\left[\frac{\partial \hat{R}(f(x_i))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)} = -\left[\frac{\partial L(y_i, f(x))}{\partial f(x_i)}\right]_{f(x)=\hat{f}^{(m-1)}(x)}. \tag{30}$$

The current solution is then updated as,

$$f_m = f_{m-1} - \rho_m \hat{g}_m(x_i), \tag{31}$$

where ρ_m is the learning rate at the m'th iteration. The process is repeated for each iteration. It is a greedy algorithm because the gradient descent is the local direction where the loss is decreasing.

Chen and Guestrin (2016) extended the gradient boosting algorithm to be more scalable, which they called extreme gradient boosting (XGB). XGB allows trees to be of different sizes, and it uses Newton boosting instead of gradient boosting. Tree sizes are determined using standard penalization. It will likely learn better tree structures from doing this (Nielsen, 2016). Using second-order derivatives with the Newton method provides more information about the direction of the gradient, which is often helpful for minimizing the loss. The XGB algorithm also uses a split finding algorithm that uses the feature quantiles instead of searching through all values (Chen and Guestrin, 2016).

F.3 Support Vector Machines (SVM)

The SVM algorithm is also a non-linear predictor that is an extension of the soft margin algorithm. The soft margin algorithm uses the feature space hyperplane that separates the observations the most while allowing some observations to be on the wrong side of the margin. The algorithm uses a cost function, that defines how much slack is allowed. SVM makes use of non-linear kernel functions to transform the feature space, making the prediction space non-linear. The SVM algorithm for regression problems makes use of a different loss-function where only residuals larger in absolute value than some constant will contribute to the loss (James, 2013). The optimization problem is a convex optimization problem with constraints. The problem can be re-expressed as,

$$\min_{\beta,\beta_0} \|\beta\|^2 + C\sum_{i=1}^{N} \xi_i, \text{ s.t. } \xi_i \geq 0, y_i(x_i'\beta + \beta_0) > 1 - \xi_i, \tag{32}$$

where C is a parameter that can be tuned, controlling how much slack is allowed. Slack means that some observations are allowed to be on the other side of the separating hyperplane. The procedure is more flexible when the feature space is enlarged using kernels or other basis expansions such as polynomials. This allows a more flexible division of the feature space. A popular choice of kernel function is the radial kernel,

$$K(x, x') = \exp\left(-\gamma\|x - x'\|^2\right) \tag{33}$$

In the regression problem

$$H(\beta, \beta_0) = \sum_{i=1}^{N} V(y_i - f(x_i)) + \frac{\lambda}{2}\|\beta\|^2, \tag{34}$$

is minimized, where λ controls how much to penalize large parameter values in order to lower the variance of the model. Here

$$V_\epsilon(r) = \begin{cases} 0, & \text{if } |r| < \epsilon \\ |r| - \epsilon, & \text{otherwise,} \end{cases} \tag{35}$$

which means error sized less than ϵ are ignored. Using a kernel then the estimation function $f(x)$ is

$$\hat{f}(x) = \sum_{i=1}^{N} \hat{a}_i K(x, x_i). \tag{36}$$

With SVMs, it is the data observations that are wrongly classified that are in the support vector. Only the observations in the support vector are given a positive weight $\hat{a}_i > 0$. Therefore, the estimation mechanism differs from linear regression with regularization, such as the ridge regression or the LASSO regression. In a LASSO regression, it is all observations that are used to make the parameter estimates. In contrast for the support vector regression, it will only be the observations that are in the support vector. Therefore, the difference from a regularized regression is which observations are used for parameter estimation and the ability to transform the input-space into a highly non-linear space using kernel functions (Awad and Khanna, 2015).

G Age specific results

References

Aggarwal, C. (2018). Neural networks and deep learning : a textbook. Cham, Switzerland: Springer.

Awad, M. and R. Khanna (2015). Support vector regression. In Efficient Learning Machines, pp. 67–80. Springer.

Bengio, Y., A. Courville, and P. Vincent (2013). Representation learning: A review and new perspectives. IEEE transactions on pattern analysis and machine intelligence 35(8), 1798–1828.

Bengio, Y., P. Simard, and P. Frasconi (1994). Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks 5(2), 157–166.

Bernardi, M. and L. Catania (2015). The model confidence set package for r.

Bollerslev, T., R. F. Engle, and D. B. Nelson (1994). Arch models. Handbook of econometrics 4, 2959–3038.

Chen, T. and C. Guestrin (2016). Xgboost: A scalable tree boosting system. In Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining, pp. 785–794.

Cho, K., B. Van Merriënboer, D. Bahdanau, and Y. Bengio (2014). On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259.

Cho, K., B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

Choi, I., D. Kim, Y. J. Kim, and N.-S. Kwark (2018). A multilevel factor model: Identification, asymptotic theory and applications. Journal of Applied Econometrics 33(3), 355–377.

Chung, J., C. Gulcehre, K. Cho, and Y. Bengio (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

Cotterell, R., A. Poliak, B. Van Durme, and J. Eisner (2017). Explaining and generalizing skip-gram through exponential family principal component analysis. In Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers, pp. 175–181.

	Age									
Deep learning models	50	51	52	53	54	55	56	57	58	59
FFNN (2000 epochs)	52	49	49	54	56	55	51	59	60	61
FFNN (500 epochs)	**54**	53	49	57	54	**56**	**64**	**65**	**65**	**66**
FFNN (with cohort)	14	14	16	13	17	15	18	17	27	37
FFNN (with factors)	52	**55**	**62**	**59**	**61**	55	59	64	63	64
FFNN (with factor loadings)	29	24	27	21	24	23	26	23	29	39
FFNN (200 epochs)	26	17	23	24	25	25	27	23	30	43
GRU (with cohort, no dropout)	5	7	6	7	7	9	10	9	14	17
GRU (with cohort)	5	6	6	7	6	7	9	8	13	18
LSTM (with cohort, no dropout)	6	7	7	7	6	7	10	10	15	16
LSTM (with cohort)	6	7	6	6	7	6	7	7	14	13
RNN (with cohort, no dropout)	6	5	6	5	6	7	8	9	11	13
RNN (with cohort)	6	5	5	5	4	5	6	7	12	23
Single population machine learning models										
Boosting (sp)	3	3	2	2	4	6	8	14	22	33
RF (sp)	1	0	0	0	0	2	3	3	5	4
SVM (sp)	0	0	0	0	0	0	1	1	2	3
XGB (sp)	10	15	23	23	23	20	29	29	33	40
Multi population factor model										
Multi level factor	48	53	58	52	50	52	54	50	52	53
Multi population machine learning models										
Boosting (mp)	6	7	10	11	13	17	16	21	28	36
RF (mp)	0	0	0	0	0	0	0	0	1	0
SVM (mp)	4	5	6	8	9	11	10	12	15	21
XGB (mp)	26	24	23	26	17	17	18	18	29	37
Single population factor models										
APC	36	36	36	35	38	34	39	37	43	47
CBD	31	36	35	36	31	37	37	40	46	49
LC (GLM)	39	40	45	43	41	35	38	34	45	48
LC (SVD)	42	41	44	42	38	36	39	36	44	51
M6	36	31	28	27	22	27	27	24	27	31
M7	24	19	21	20	15	19	19	15	17	16
M8	4	4	5	7	8	12	14	19	27	35
Plat	45	35	35	35	33	34	32	27	33	36
RH	10	13	16	15	13	20	22	26	36	37

Table 1: Individual age MCS results for ages 50-59.

	Age									
Deep learning models	60	61	62	63	64	65	66	67	68	69
FFNN (2000 epochs)	63	65	66	67	66	**69**	68	55	48	49
FFNN (500 epochs)	**69**	**69**	**69**	**69**	**69**	**69**	**69**	50	47	45
FFNN (with cohort)	53	67	68	**69**	**69**	68	67	35	34	32
FFNN (with factors)	65	68	**69**	**69**	**69**	**69**	**69**	45	42	43
FFNN (with factor loadings)	44	64	66	65	63	50	17	5	5	5
FFNN (200 epochs)	48	62	66	67	66	58	50	15	16	18
GRU (with cohort, no dropout)	32	50	**69**	67	**69**	68	**69**	**67**	66	65
GRU (with cohort)	36	54	67	**69**	**69**	67	66	63	**68**	**66**
LSTM (with cohort, no dropout)	25	46	68	**69**	**69**	67	67	63	65	62
LSTM (with cohort)	30	51	67	**69**	**69**	**69**	**69**	65	63	62
RNN (with cohort, no dropout)	22	45	68	**69**	**69**	68	**69**	65	66	64
RNN (with cohort)	39	64	66	67	66	66	66	58	58	59
Single population machine learning models										
Boosting (sp)	45	55	60	58	59	54	46	31	27	25
RF (sp)	11	18	29	25	25	20	5	3	1	3
SVM (sp)	6	5	5	1	2	1	1	1	0	1
XGB (sp)	39	57	62	62	58	51	33	13	10	11
Multi population factor model										
Multi level factor	56	64	65	65	65	65	59	27	24	23
Multi population machine learning models										
Boosting (mp)	46	54	58	57	54	48	38	21	13	16
RF (mp)	0	0	0	0	0	0	0	0	0	0
SVM (mp)	27	34	39	37	35	31	26	19	18	21
XGB (mp)	42	63	64	64	62	59	31	11	5	5
Single population factor models										
APC	52	60	62	62	60	59	54	21	16	18
CBD	55	65	66	66	65	63	58	20	15	10
LC (GLM)	49	62	63	63	62	57	47	17	15	17
LC (SVD)	49	63	65	63	63	60	50	16	12	14
M6	41	52	48	47	45	36	29	12	7	7
M7	21	26	24	22	23	20	20	8	6	6
M8	46	61	64	65	62	57	44	4	1	2
Plat	42	59	59	59	60	58	50	10	5	6
RH	47	52	52	52	52	49	46	35	41	38

Table 2: Individual age MCS results for ages 60-69.

	Age									
Deep learning models	70	71	72	73	74	75	76	77	78	79
FFNN (2000 epochs)	46	42	45	46	44	40	47	50	45	49
FFNN (500 epochs)	47	40	36	44	40	36	48	51	45	48
FFNN (with cohort)	31	22	23	33	32	28	34	30	34	37
FFNN (with factors)	42	35	36	37	35	36	38	39	39	38
FFNN (with factor loadings)	6	2	5	4	4	3	1	2	1	4
FFNN (200 epochs)	22	17	18	19	12	10	16	14	15	13
GRU (with cohort, no dropout)	55	56	53	57	54	**56**	58	**62**	**60**	53
GRU (with cohort)	39	29	18	26	19	20	20	22	17	25
LSTM (with cohort, no dropout)	58	51	57	56	54	49	56	49	50	49
LSTM (with cohort)	**61**	52	45	43	41	36	36	30	33	37
RNN (with cohort, no dropout)	**61**	**60**	58	64	58	**56**	**61**	58	59	**60**
RNN (with cohort)	60	**60**	**62**	**65**	**60**	**56**	57	54	58	55
Single population machine learning models										
Boosting (sp)	28	27	28	25	21	24	15	20	24	22
RF (sp)	4	1	4	2	1	2	1	1	1	5
SVM (sp)	1	0	1	1	1	1	0	0	1	1
XGB (sp)	12	5	9	3	5	6	6	4	5	7
Multi population factor model										
Multi level factor	25	19	16	23	18	17	16	15	16	17
Multi population machine learning models										
Boosting (mp)	12	16	13	17	8	15	12	14	17	13
RF (mp)	0	0	1	5	3	3	0	1	5	7
SVM (mp)	18	12	16	16	9	7	7	4	4	2
XGB (mp)	7	2	5	5	1	4	3	4	3	1
Single population factor models										
APC	13	17	12	13	11	8	10	14	11	15
CBD	6	3	3	3	1	2	0	0	2	6
LC (GLM)	20	17	21	19	18	14	16	12	19	19
LC (SVD)	18	13	19	21	18	15	16	14	20	16
M6	10	7	10	11	10	12	11	11	14	15
M7	5	3	5	6	3	3	1	2	2	1
M8	3	0	3	2	1	2	0	0	1	0
Plat	4	5	3	3	1	2	1	1	1	3
RH	37	34	32	36	32	26	33	32	29	30

Table 3: Individual age MCS results for ages 70-79.

	Age									
Deep learning models	80	81	82	83	84	85	86	87	88	89
FFNN (2000 epochs)	47	48	49	52	50	51	56	57	60	59
FFNN (500 epochs)	48	41	49	47	48	49	52	45	52	55
FFNN (with cohort)	34	29	37	44	42	50	49	53	62	58
FFNN (with factors)	42	36	42	49	46	48	52	53	54	55
FFNN (with factor loadings)	3	4	5	1	3	2	2	3	3	4
FFNN (200 epochs)	18	19	19	28	23	23	30	34	36	40
GRU (with cohort, no dropout)	57	54	61	63	63	56	53	56	44	46
GRU (with cohort)	20	19	30	51	55	58	65	65	**66**	**63**
LSTM (with cohort, no dropout)	56	48	58	63	62	59	64	61	61	61
LSTM (with cohort)	47	38	53	59	48	54	54	57	55	54
RNN (with cohort, no dropout)	57	**62**	**66**	**64**	**66**	**62**	**66**	**66**	64	59
RNN (with cohort)	**65**	60	64	63	65	59	60	58	63	**63**
Single population machine learning models										
Boosting (sp)	20	15	22	17	14	8	10	8	9	6
RF (sp)	5	5	3	3	1	1	0	1	4	7
SVM (sp)	1	0	1	1	0	0	1	4	3	6
XGB (sp)	11	4	4	5	3	2	4	4	11	7
Multi population factor model										
Multi level factor	13	13	14	15	21	21	25	29	33	38
Multi population machine learning models										
Boosting (mp)	18	17	20	24	15	10	4	5	6	6
RF (mp)	9	8	20	33	5	0	1	7	8	15
SVM (mp)	3	1	1	1	1	3	3	4	5	9
XGB (mp)	6	3	4	3	0	3	0	4	5	8
Single population factor models										
APC	20	26	30	44	46	49	49	49	51	49
CBD	7	8	8	16	21	29	38	41	46	48
LC (GLM)	19	15	19	26	32	34	39	39	40	39
LC (SVD)	16	14	17	20	26	28	39	36	41	43
M6	22	19	23	28	26	30	36	37	43	40
M7	2	4	6	14	18	26	35	41	42	36
M8	1	0	0	0	0	0	2	2	7	7
Plat	2	6	7	8	13	21	28	42	50	51
RH	29	25	25	29	26	28	27	31	33	34

Table 4: Individual age MCS results for ages 80-89.

	Age					
Deep learning models	90	91	92	93	94	95
FFNN (2000 epochs)	60	62	63	62	60	65
FFNN (500 epochs)	55	56	66	64	63	63
FFNN (with cohort)	63	**68**	**67**	65	**66**	**68**
FFNN (with factors)	56	57	59	61	62	61
FFNN (with factor loadings)	8	15	14	23	27	30
FFNN (200 epochs)	50	51	53	54	54	53
GRU (with cohort, no dropout)	46	46	50	62	50	66
GRU (with cohort)	58	55	49	55	49	52
LSTM (with cohort, no dropout)	60	62	64	**66**	65	65
LSTM (with cohort)	61	57	59	58	58	57
RNN (with cohort, no dropout)	**65**	60	59	64	59	65
RNN (with cohort)	61	54	56	59	59	52
Single population machine learning models						
Boosting (sp)	8	16	11	20	26	35
RF (sp)	12	13	14	30	31	53
SVM (sp)	9	19	22	34	55	69
XGB (sp)	12	15	16	18	23	35
Multi population factor model						
Multi level factor	48	48	54	58	56	53
Multi population machine learning models						
Boosting (mp)	7	11	10	11	20	26
RF (mp)	29	38	46	59	55	48
SVM (mp)	13	25	34	46	48	52
XGB (mp)	10	14	12	29	24	39
Single population factor models						
APC	41	37	36	28	26	22
CBD	57	54	52	51	48	47
LC (GLM)	53	57	57	61	60	58
LC (SVD)	54	52	55	56	51	48
M6	50	49	41	40	40	33
M7	35	32	29	28	24	26
M8	13	15	14	16	17	16
Plat	56	63	60	59	56	57
RH	42	37	41	42	41	40

Table 5: Individual age MCS results for ages 90-95.

Goodfellow, I. (2016). Deep learning. Cambridge, Massachusetts: The MIT Press.

Hansen, P. R. and A. Lunde (2005). A forecast comparison of volatility models: does anything beat a garch (1, 1)? Journal of applied econometrics 20(7), 873–889.

Hansen, P. R., A. Lunde, and J. M. Nason (2011). The model confidence set. Econometrica 79(2), 453–497.

Hastie, T. (2009). The elements of statistical learning : data mining, inference, and prediction. New York: Springer.

Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. Neural computation 9(8), 1735–1780.

Ioffe, S. and C. Szegedy (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

James, G. (2013). An introduction to statistical learning : with applications in R. New York, NY: Springer.

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

LeCun, Y., Y. Bengio, and G. Hinton (2015). Deep learning. nature 521(7553), 436–444.

Lipton, Z. C., J. Berkowitz, and C. Elkan (2015). A critical review of recurrent neural networks for sequence learning. arXiv preprint arXiv:1506.00019.

Mullainathan, S. and J. Spiess (2017). Machine learning: an applied econometric approach. Journal of Economic Perspectives 31(2), 87–106.

Nielsen, D. (2016). Tree boosting with xgboost-why does xgboost win" every" machine learning competition? Master's thesis, NTNU.

Richman, R. (2018). Ai in actuarial science. Available at SSRN 3218082.

Richman, R. and M. V. Wüthrich (2018). A neural network extension of the lee–carter model to multiple populations. Annals of Actuarial Science, 1–21.

Sutton, R. (1998). Reinforcement learning : an introduction. Cambridge, Mass: MIT Press.

Taieb, S. B., G. Bontempi, A. F. Atiya, and A. Sorjamaa (2012). A review and comparison of strategies for multi-step ahead time series forecasting based on the nn5 forecasting competition. Expert systems with applications 39(8), 7067–7083.

Taieb, S. B., R. J. Hyndman, et al. (2012). Recursive and direct multi-step forecasting: the best of both worlds, Volume 19. Citeseer.

Zhang, A., Z. C. Lipton, M. Li, and A. J. Smola (2020). Dive into Deep Learning. https://d2l.ai.