Predictive Analytics and Futurism

December 2016

Issue 14

Abstractions & Working Effectively Alongside Artificial Intellects

By Dodzi Attimu and Bryon Robidoux

In the article "2036: An Actuarial Odyssey with AI" that appeared in the July 2016 issue of Predictive Analytics and Futurism, we explored the impact of artificial intelligence (AI) on actuarial work in particular and white collar work in general. Though there is the tendency to sensationalize the apocalyptic scenarios vis-à-vis, an "AI-calypse," there is a still a possibility (even if small) that net outcomes could be, well, apocalyptic (e.g., drastic reduction in employment in traditional jobs without others springing up, leading to social upheavals). Although the full implications cannot be forecast with certainty, we can say with certainty that the humans will increasingly continue to work alongside machines (artificial intellects[1]). Without a good framework to conceptualize and implement the partnership between humans and machines, very suboptimal utilization of technology can occur. This article specifically is about how abstraction, an important software development concept, can help in this regard. In addition, while there will be more emphasis on abstractions in the framework of software, the concept of abstraction is not limited to that domain.

ABSTRACTION

The concept of abstraction is ubiquitous. In everyday communication, it is summed up in the notion of communication based on one's audience. Another use of this notion is captured in the phrase "keeping information at a high-level." One definition of the word abstraction (See [3]) is: "The process of formulating generalized ideas or concepts by extracting common qualities from specific examples." Informally, abstraction can be said to be a way of specifying the "what" rather than the "how." In software development circles, abstractions are a means of managing complexity by thinking of software in terms of levels where each level has the right amount of information with more detailed information residing in the lower levels of the hierarchy. Abstractions can serve two related purposes:

1 Generalization—The purpose here is to focus attention on relevant components in a given layer. By abstracting away the lower level details, one can focus on the key components in a given layer. As an example, a high-level programming language (e.g., C++, Java, C#) is an abstraction of a lower level language (assembly language). The user at the higher level language layer doesn't have to worry about the low level assembly language layer. Domain specific languages (DSLs) are also higher levels of abstraction of lower level languages that process them. In the field of artificial intelligence, the highest level of abstraction would be natural language.

2 Flexible Implementation—A direct consequence of (i) above is the opportunity to carve out the specific implementation details and appropriately deal with them in a lower layer. In a solution that outputs data for example, one could have an abstract representation of the data and deal with the details of the various physical output formats like, Excel, JSON, HTML, etc. The flexibility stems from the fact that different implementations can be built for the same general purpose.

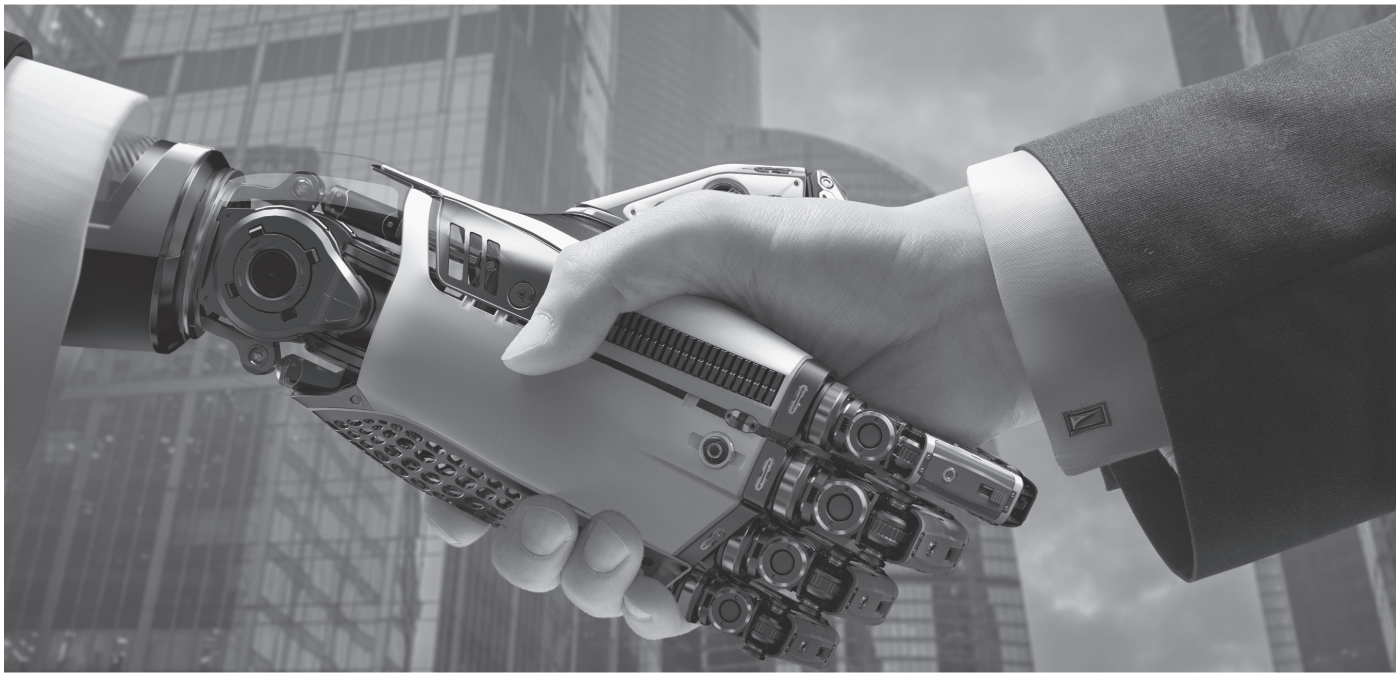
Different layers of abstraction could be identified for a given context/scenario. Typically, (i) would represent a higher level of abstraction and (ii) a lower level of abstraction. But it may be necessary, to further carve out a lower level of abstraction from (ii) and so on. In addition, in the programming language example noted in (i), an assembly language is a level of abstraction above the machine language (a language whose syntax consists of 0's and 1's).

In software development, one can identify three types of abstractions: data abstraction, procedural abstraction and configuration abstraction. Data abstraction is concerned with unifying different input sources and coming up with a simplified and generic representation. Procedure abstraction is concerned with defining different types of functionality in generic ways without specifying the details. For example, if the actuary wants to value a future, swap or option, they define a common way of valuing a derivative. The details of valuing each individual are ignored at this level (of abstraction). This allows the ability to design functionality without getting overwhelmed in details and allows for easier extension to other types of derivatives in the future. Configuration abstraction deals with changing the behavior of the software without requiring more code modifications. At its simplest level, this minimizes or in some cases avoids completely the coding of any details needed for the model to run.

FIXED AND VARIABLE PARTS OF SOFTWARE SOLUTION

One can classify AI into the categories of classical, machine learning and machine intelligence. At the highest level of abstraction, AI is software. In that regard, let's assume we have a system's logic as Sys^{Logic}, and the input (structure) that goes with the logic, Sys^{Input}[2]. That will constitute a logic-input pair, i.e., $(Sys^{Logic}, Sys^{Input})$. For a given software system, Sys, we can consider the input structure as a function of the logic, i.e.,

$$Sys^{Input} = \tau(Sys^{Logic})$$



where τ is the function that translates the logic of system to its input structure.

Given two software designs A and B to solve a problem, we would have their mathematical representation as

$$\left(\mathrm{Sys}^{\mathrm{Logic}^{A}}, \mathrm{Sys}^{\mathrm{Input}^{A}} = \tau(\mathrm{Sys}^{\mathrm{Logic}^{A}})\right)$$
$$\text{and}$$
$$\left(\mathrm{Sys}^{\mathrm{Logic}^{B}}, \mathrm{Sys}^{\mathrm{Input}^{B}} = \tau(\mathrm{Sys}^{\mathrm{Logic}^{B}})\right)$$

respectively. The importance of the representation is that for a given system, there is a correspondence between the software code and the input structure.

Conversely, observe that given an input structure, there is a logical model specification that works with the structure to meet system requirements. In other words, given an input structure $\mathrm{Sys}^{\mathrm{Input}}$, one can obtain the corresponding logic, via the inverse transformation $\mathrm{Sys}^{\mathrm{Logic}}=\tau^{\leftarrow}(\mathrm{Sys}^{\mathrm{Input}})$. This begs the question whether there is a better starting point viz $\tau^{\leftarrow}$ and τ. Using a user experience (UX) paradigm (See [2]), the input design/structure should come first. In our context, it is the input structure that should be mapped first, i.e., $\tau^{\leftarrow}$, should be the first focus as it maps the input to the logic. In addition, the exact details of how the input is structured can be abstracted away as well into another layer where emphasis is placed first on what data is required before getting to how (where) it is stored (e.g., txt file, xml, etc). We will consider the coded logic as the fixed part of the system and the input as the variable part. Designing a system where different behavior can be achieved via changes to input enhances flexibility and transparency in the use of the system. In model building projects for example, there is potential to get unnecessarily held up over choice of methodology but with the appropriate abstraction, the system can be designed to support alternative approaches via the inputs. This effectively defers and delegates the decision on the choice of methodology to the end user.

CLASSICAL AI—ABSTRACTION IN MODEL DESIGN

Designing flexible models is an imperative in the fast-paced world of actuaries these days. This is an area where effective abstractions can be used to enhance flexibility of the system. Another important corollary of the pace of modeling requirements and ERM best practices is the uniformity of models across the enterprise. To achieve this, models should be designed leveraging abstractions that support flexibility[3] for different uses/purposes. We illustrate with a relatively simple example. Consider a model that at any point in time evaluates a call option on an index.[4] Mathematically, the formula for a European call on an underlying S with strike K in the generalized Black-Scholes[5] model is C(t,T;) given by:

$$C(t, T; \Sigma_{t,T}, K) = S(t)N(d_1) - Ke^{-(T-t)Y(t,T)}N(d_2), \qquad (1A)$$

$$\text{where } d_1 = \frac{\ln\left(\frac{S(t)}{K}\right) + \left(Y(t,T) + \frac{1}{2}\Sigma_{t,T}{}^2\right)(T-t)}{\Sigma_{t,T}\sqrt{(T-t)}}, \; d_2 = d_1 - \Sigma_{t,T}\sqrt{T-t} \qquad (1B)$$

In the formulae above, Y(t,T) is the yield from time t to T, and $\Sigma_{t,T}$ is the "(implied) volatility" of the forward price of the index. The forward price of an index (underlying) S, is defined as

$$F^S(t, T) := \frac{S(t)}{P(t, T)}$$

where P(t,T) is the price at time t of a zero-coupon bond paying a unit amount of currency at time T defined by $P(t,T)=e^{-(T-t)Y(t,T)}$. This general model is actually more ideal for modeling purposes as the yield curve at any projection time step, t,

$$(Y(t,t+T))_{T \text{ in } \{T\_i \mid i \text{ in set of points on yield curve}\}}$$

is typically either an input or internally generated in actuarial models.

Under the classical Black-Scholes model, the formula for a call on the a stock index S is given by

$$C(t,T;\sigma_{t,T}) = S(t)N(d_1) - e^{-(T-t)r}N(d_2) \tag{2A}$$

$$\text{where } d_1 = \frac{\ln\left(\frac{S(t)}{K}\right)+\left(r_t+\frac{1}{2}\sigma_{t,T}^2\right)(T-t)}{\sigma_{t,T}\sqrt{(T-t)}},\ d_2 = d_1 - \sigma_{t,T}\sqrt{T-t} \tag{2B}$$

Where the interest rate between t and T is assumed to be the constant short rate, r_t, and the (implied) volatility of the stock index S between t and T is $\sigma_{t,T.}$

Though both models are idealizations of reality, the modeling needn't be held-up over uncertainties/differences of opinion about what to implement. This is because one can find an abstraction that can handle both approaches making any debate essentially irrelevant to model development. To see this, it suffices to note the relationship between the two approaches (or formulae). By inspection of the formulae in (2) and (1), the following relationships can be inferred:

> In model building projects for example, there is potential to get unnecessarily held up over choice of methodology but with the appropriate abstraction, the system can be designed to support alternative approaches. ...

- $Y(t,T)=r_t$ —That is, the generalized model uses the yield between projection time step and maturity, whereas the classic model uses a constant short rate (could be proxied by the short end of the yield curve for example)
- $\Sigma_{t,T}= \sigma_{t,T}$ —That is, for the generalized model, $\Sigma_{t,T}$ is the "implied volatility" of the forward price of the underlying index, (i.e., the variable $\frac{S(t)}{P(t,T)}$ which incorporates the volatility in interest rates), whereas for the classical model, $\sigma_{t,T}$ is the "implied volatility" of the underlying index, S(t).

Consequently, building model components that utilize the following input:

- Projection time, t;
- Time of call expiry,T;
- The value of stock index at time projection time step, S(t);
- The strike price, K;
- Implied volatility parameter—this would be $\Sigma_{t,T}$ for the generalized pricing formula and $\sigma_{t,T}$ for the classical pricing formula; and
- Interest rate parameter—this would be Y(t,T) for the general case and r_t for the classical case,

should be able to accommodate either approach based on the input structure (focus will be on the raw input structure here):

- Have a volatility surface which is a two dimensional matrix structure, $\Sigma\left(\frac{K}{S},\tau\right)$ where $\frac{K}{S}$ is the money-ness parameter and τ is the time to maturity; and
- At any projection time step have a parameter setting procedure that determines how to source the values:
  - In the case of classical Black-Scholes approach, choose the point on yield curve that will be used as short rate (default to the three-month rate, for example), otherwise, for the generalized case, choose the yield with tenor equal to maturity. If interpolation of the yield curve is required, utilize an interpolation function to do so.
  - For volatility, we would expect the user to enter the correct projected "implied volatility surface" corresponding to the approach desired, i.e., to use the classical paradigm, the input would be the projected surface for the stock index, whereas in the generalized case, it would be that of the forward price of the index.
    - This structure naturally handles instances where the surface is flat along one or both dimensions of (money-ness) and τ (term structure).

From a model configuration perspective, we will expose a configuration/input to the user, e.g., whether to use classical or generalized formula and in the case of the classical, which point on the yield curve to use as proxy for the "constant replicating short rate." The above approach unifies both methodologies giving the user the flexibility to ultimately develop their assumptions and corresponding inputs (hence methodology).[6] A very unproductive approach in our opinion is to create a tailored solution for one case only to later have to "change" it to another case. Consequently, by pushing the decision to the user (via inputs), time as well as energy is saved.

Finally, this example also illustrates how a modeling functionality in particular and models in general can support sundry modeling uses, e.g., pricing, valuation, risk management, etc. In particular, more sophisticated volatility assumptions may be utilized for pricing purposes compared to for valuation purposes and the abstraction handles each approach in the same generic way.

MACHINE LEARNING SYSTEMS

What is machine learning? At the highest level of abstraction, this is a mechanism of creating systems that performs a task through processing of data without being explicitly programmed. The concept is aptly summarized in Mitchell T (1997): "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

The experience relates to a data processing step which starts with a data abstraction using a generic vector $x=(x^1,\ldots,x^{N_{feat}})$ where N_{feat} represents the number of features for each data point. For example, to approximate the rate of inflation as a function of the 90-day and the one-year treasury rates, we have $x=(x^1,x^2)$, where x^1,x^2 represents the 90-day and one-year rates respectively.

Next, a helpful abstraction is to consider the different observations of x, with the processing of each observation constituting the experience E. Using subscripts to denote the observation number so that

$$\mathbf{x}_k=(x_k^1, \ldots, x_k^{N_{feat}})$$

represents the kth input observation, an abstract representation of input data to a machine learning system is a matrix (or a table),

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_{N} \end{pmatrix} = \begin{pmatrix} x_1^1 & \cdots & x_1^{N_{feat}} \\ \vdots & \ddots & \vdots \\ x^1 & & x^{N_{feat}} \end{pmatrix}.$$

In a supervised machine learning context, another input is the actual values corresponding to each of the input data observations. In our inflation prediction problem, these would correspond to the inflation corresponding to each 90-day and one-year treasury rate observation. We can represent the output a matrix, Y, with N_{obs} rows, each row corresponding to a data observation, i.e.,

$$\mathbf{Y} = \begin{pmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_{N_{obs}} \end{pmatrix} = \begin{pmatrix} y_1^1 & \cdots & y_1^{N_{out}} \\ \vdots & \ddots & \vdots \\ y_{N_{obs}}^1 & \cdots & y_{N_{obs}}^{N_{out}} \end{pmatrix},$$

where N_{out} is the number of components of the output. In our inflation prediction case, $N_{out}=1$ and Y is a column vector.

Though machine learning encompasses more than artificial neural networks (ANNs), we will focus on ANNs as it is the approach to AI outside the classical methodology that is inspired (albeit in a very simplified way) on the working of the brain. In that regard, there are different artificial neural network architectures, with a common architecture being the feed-forward architecture. The machine learning problem reduces to finding an approximating function that performs the task T.[7] There are many libraries that provide implementations for the actual training step which is the iterative estimation of parameters (weights of the neurons in network) of the approximation function. In these settings, the user needs to specify the number of layers in the neural network as well as the number of neurons per layer.[8]

This suggests a data structure of a vector

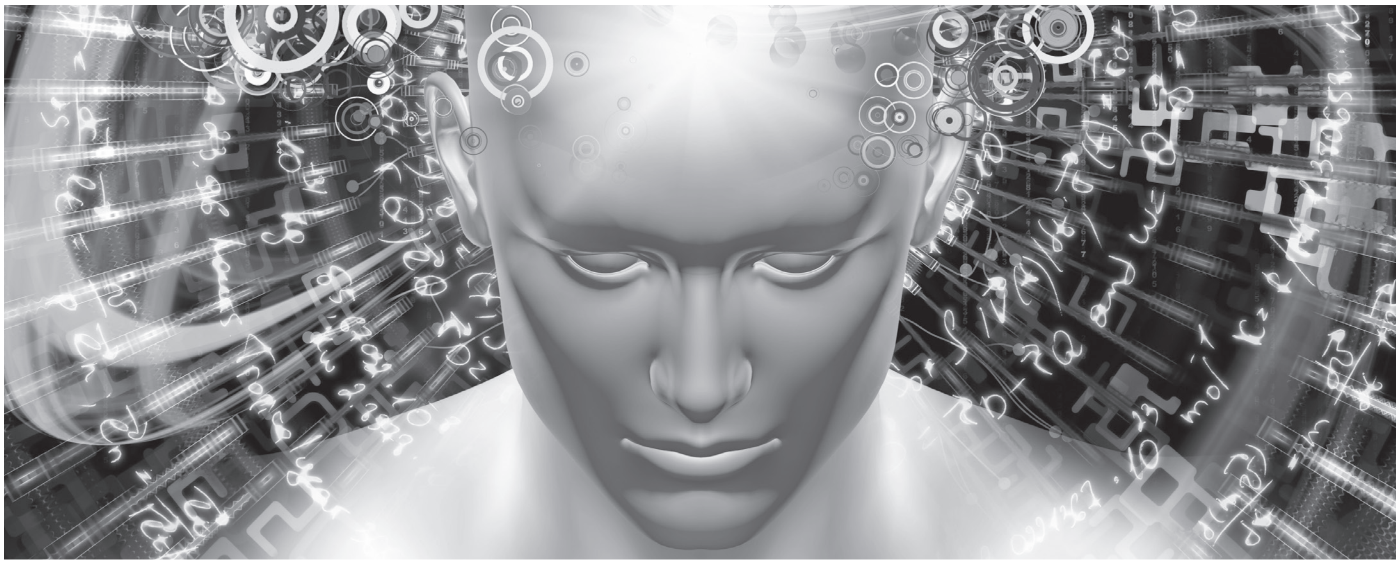
$$\mathbf{n}=(n^1, \ldots, n^{N_{layer}}),$$

where n^k represents the number of neurons in the k-th layer and N_{layer} represents the number of layers in the network. With this abstraction we have a blueprint for engineering a neural network system whose configuration is driven by inputs including X, Y and n. In so doing, we have abstracted away the low level details of the heavy lifting that would be carried out by a machine learning engine (e.g., an R package like neuralnet, Python package like scikit-learn, or first principle implementation) and all a user needs to utilize the system would be the data input data structure.

BIOLOGICAL NEURAL NETWORKS/ MACHINE INTELLIGENCE

One observation from the previous section is that abstractions play a key role in the world of traditional ANNs from the generic input structure to the generic processing of each input/ observation. In this section we point out the fact that the brain itself is a big abstraction leveraging mechanism.

As described in the article "2036: An Actuarial Odyssey with AI" by Attimu and Robidoux (*Predictive Analytics and Futurism*, July 2016), Machine Intelligence systems attempt to model how the brain works with the Hierarchical Temporal Memory (HTM) framework developed by Jeff Hawkins of numenta.org. As noted in Hawkins, et. al. (2016), Classic AI and ANNs are designed to solve specific problems, e.g., the model component and ANN structure illustrated earlier. The biology of the neocortex, which occupies about 75 percent[10] of the brain's volume, is the basis of (HTM). Though one could be tempted to think that the neo cortex of the brain has very different algorithms for hearing, vision, touch, and other senses, this is not the case. The brain utilizes common algorithms for vision, hearing, touch, language and behavior.[11] Within the context of



this article, we can infer that though the former approaches (classical AI and ANNs) do admit abstractions, these abstractions are not powerful enough to generalize the cognitive processes of the brain. In fact, the brain's function is probably the best example of the use of abstraction to create a generalized computing framework.

Knowledge representation (representing facts and relationships in the world) is difficult using traditional computing approaches. The brain's approach to knowledge representation utilizes a data structure called Sparse Distributed Representations (SDRs). The SDR is the perfect example of the brain using data abstraction to abstract different sensory inputs into a common data structure. Just like a computer word, an SDR is made up of 0 and 1 bits. Unlike computer information, which could be represented using 8, 32, or 64 bits and for which semantic meaning of the information is captured in the bit representation as a whole, an SDR is made up of thousands of bits and they are sparsely activated (i.e., a small fraction of the bits are 1s) and each bit contributes to the semantic meaning of the representation. The SDR representation has some very powerful and useful properties including being robust to noise.

To illustrate the difference between sparse and dense representations, consider the ASCII code for the letter 'x' which is 01111000. When we flip the 4th digit, we obtain the representation 01101000 which corresponds to the letter 'h.' This illustrates the fact that there is no semantic meaning inherent in the individual bits, but in the collection of all the bits. This representation is not robust to noise. On the other hand, consider an SDR representation scheme which consists of 1000 bits of which only 1 percent are 1s. The bits of SDRs carry semantic information. For example, the positions in the SDR could represent different characteristics of class of data represented. To illustrate, consider sound data where bits would capture pitch, amplitude, etc. Furthermore, two SDR's that are semantically similar will have overlaps in their "on" ("1") bits. Consider the information encapsulated by two SDRs shown below:

$$\mathrm{SDR}_x = \underbrace{011101011100010001\ldots000001}_{1000\text{ bits}}$$
$$\mathrm{SDR}_y = \underbrace{011101011100010000\ldots100010}_{1000\text{ bits}}$$

There is an overlap in position of 80 percent of the "1" bits. Since the individual bits in an SDR have meaning, the x and y are closer semantically than x and another data point, z, for example, whose "1" bit positions overlap with 50 percent of those of x. In fact, SDRs have very important mathematical properties that traditional data structures lack and which make them a particularly powerful abstraction of information for modeling cognitive processes. One important property is their robustness to noise. Indeed, a subset of the on ("1') bit positions can be used to identify an SDR with high accuracy.[12] For details we refer the reader to Hawkins, et. al. (2016).

We revisit our earlier point about the cognitive (computational) processes in the neocortex being homogenous. The key to learning via the neocortex of the brain is that every sense is responsible for putting its information into a sparse distributed representation (SDR). The SDR must capture the important characteristics for the task. At this point, the brain doesn't have to worry about what created the data (SDR). It only has to concern itself with recognizing patterns. In effect, the details of the specific types of information input are abstracted away via SDRs. Consequently, a general algorithm can work on different types of cognitive processes e.g., smell, sight, touch, etc. A great example that illustrates this idea is the Brainport which is a sensor that sits upon the top of the tongue and allows blind individuals to "see" using the tongue.[13]

The pattern recognition and learning is done within Hierarchical Temporal Memory (HTM). This is a perfect example of function abstraction in the brain. Each level of the hierarchy works with the SDR data structure and **performs the same learning algorithm.** The difference is the level of the information. Imagine learning a language, you start by learning letters. You then learn words and then finally sentences. Each one of the learning tasks would be at each level in the hierarchy. Abstracting the learning in this way creates a generalized algorithm which reduces the training time and decreases the memory usage compared to using traditional methods of data processing.

The pattern recognition is done by first learning spatial patterns, which constitutes learning bits that often appear together. The temporal patterns learn how the spatial patterns appear throughout time. After these patterns have been learned it can start using them to make inference on new data. These inferences can be used to predict what is likely to occur next. The nice part of this design is that you don't have to separate learning from inference. They feed off of each other with this design. In the Numenta Platform for Intelligent Computing (NUPIC) library, encoders are used to change your data into an SDR. You feed the SDR to the Spatial Pooler and Temporal Pooler to start the learning process.

CONCLUSION

Abstractions are not only a means to create flexible and robust systems; they also help our understanding of concepts and how they relate to each other. Designing software solutions requires the use of appropriate abstractions to make systems both manageable and easy to use. From classical software to more modern AI oriented software, thinking in terms of appropriate abstractions helps engineer more effective solutions to improve the human-machine collaboration we will increasingly see in white-collar work in general and in actuarial work in particular. ■

Dodzi Attimu, FSA, CERA, CFA, MAAA, Ph.D., is director and actuary at Prudential Financial Inc. He can be contacted at *dodzi.attimu@prudential.com.*

Bryon Robidoux, FSA, is director and actuary, at AIG in Chesterfield, Mo. He can be reached at *Bryon.Robidoux@aig.com.*

REFERENCES

1 Kerievsky J, Refactoring to Patterns, Addison Wesley, 2005

2 Esposito D and Saltarello A, Microsoft.NET: architecting Applications for the Enterprise, 2nd Ed , Microsoft Press, 2014.

3 *http://dictionary.reference.com/browse/abstraction*

4 *https://en.wikipedia.org/wiki/Abstraction_(computer_science)*

5 Bjork T, Arbitrage Theory in Continuous Time, 3rd Ed, Oxford Finance, 2009

6 Jeff Hawkins , On Intelligence, Times Books, 2005

7 Kaplan J, 2015. Humans Need Not Apply: A Guide to Wealth and Work in the Age of Artificial Intelligence, New Haven: Yale University Press

8 Awad M and Khana R, Efficient Learning Machines, Apress Open, 2016

9 *https://en.wikipedia.org/wiki/Brainport*

10 *http://numenta.com/assets/pdf/whitepapers/hierarchical-temporal-memory-cortical-learning-algorithm-0.2.1-en.pdf*

11 Mitchell, T. Machine Learning, McGraw Hill,1997

12 Attimu D and Robidoux B, PAF Newsletter, Issue 13, July 2016

13 Hawkins, J. et al. 2016. Biological and Machine Intelligence. Release 0.4. Accessed at *http://numenta.com/biological-and-machine-intelligence*

14 K. Hornik et al, Multilayer feedforward networks are universal approximators. Neural Networks, 2(5): 359-366, 1989.

15 Hagan T et al, Neural Network Design, 2nd Ed, 2016.

ENDNOTES

1 The term is borrowed from Kaplan(2015)

2 In this article, we consider configurations as inputs

3 This is related to the transformation τ^{c} or τ discussed earlier

4 This could be part of an Equity-Indexed Annuity projection engine

5 Unlike the classical Black-Scholes model that assumes constant (deterministic) interest rates, the generalized model dispenses of that restriction, being valid under stochastic interest rate assumption. See for example, pages 406-409 in [5]

6 It is well known that both approaches are simplifications and may not be appropriate for some modeling situations

7 See Hornik et al (1989)

8 For an introduction to neural networks, see Hagan et al (2016).

9 Other things that might be exposed to input include the performance measure P and other lower level implementation choices that are part of the core machine learning engine API employed

10 See Hawkings et al (2016)

11 This was first proposed by Vernon Mountcastle in 1979 (See [13])

12 In fact the human brain seems to identify entities with just a subset of information e.g. One may be able to identity another's voice even if the voice is in a noisy background.

13 See [9]. Note that this example does not explicitly rely on the abstractions of HTM per say, it and is evidence of generality of cognitive algorithms utilized by the brain.