MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

EXPLORATION OF SURROGATE MODELS FOR REMAINING USEFUL LIFE ESTIMATION

Autor: José Ignacio Ríos Goytre

Director: Prof. Dr. Markus Zimmermann

Co-Director: M.Sc. Simon Pfingstl

Madrid

Agosto de 2019

AUTHORIZATION FOR DIGITALIZATION, STORAGE AND DISSEMINATION IN THE NETWORK OF END-OF-DEGREE PROJECTS, MASTER PROJECTS, DISSERTATIONS OR BACHILLERATO REPORTS

1. Declaration of authorship and accreditation thereof.

The author Mr. /Ms. José Ignacio Ríos Goytre

HEREBY DECLARES that he/she owns the intellectual property rights regarding the piece of work: "Exploration of Surrogate Models for Remaining Useful Life Estimation"

that this is an original piece of work, and that he/she holds the status of author, in the sense granted by the Intellectual Property Law.

2. Subject matter and purpose of this assignment.

With the aim of disseminating the aforementioned piece of work as widely as possible using the University's Institutional Repository the author hereby **GRANTS** Comillas Pontifical University, on a royalty-free and non-exclusive basis, for the maximum legal term and with universal scope, the digitization, archiving, reproduction, distribution and public communication rights, including the right to make it electronically available, as described in the Intellectual Property Law. Transformation rights are assigned solely for the purposes described in a) of the following section.

3. Transfer and access terms

Without prejudice to the ownership of the work, which remains with its author, the transfer of rights covered by this license enables:

a) Transform it in order to adapt it to any technology suitable for sharing it online, as well as including metadata to register the piece of work and include "watermarks" or any other security or protection system.

b) Reproduce it in any digital medium in order to be included on an electronic database, including the right to reproduce and store the work on servers for the purposes of guaranteeing its security, maintaining it and preserving its format.

c) Communicate it, by default, by means of an institutional open archive, which has open and cost-free online access.

d) Any other way of access (restricted, embargoed, closed) shall be explicitly requested and requires that good cause be demonstrated.

e) Assign these pieces of work a Creative Commons license by default.

f) Assign these pieces of work a HANDLE (*persistent* URL). by default.

4. Copyright.

The author, as the owner of a piece of work, has the right to:

a) Have his/her name clearly identified by the University as the author

b) Communicate and publish the work in the version assigned and in other subsequent versions using any medium.

c) Request that the work be withdrawn from the repository for just cause.

d) Receive reliable communication of any claims third parties may make in relation to the work and, in particular, any claims relating to its intellectual property rights.

5. Duties of the author.

The author agrees to:

a) Guarantee that the commitment undertaken by means of this official document does not infringe any third party rights, regardless of whether they relate to industrial or intellectual property or any other type.

b) Guarantee that the content of the work does not infringe any third party honor, privacy or image rights.
c) Take responsibility for all claims and liability, including compensation for any damages, which may be brought against the University by third parties who believe that their rights and interests have been infringed by the assignment.
d) Take responsibility in the event that the institutions are found guilty of a rights infringement regarding the work subject to assignment.

6. Institutional Repository purposes and functioning.

The work shall be made available to the users so that they may use it in a fair and respectful way with regards to the copyright, according to the allowances given in the relevant legislation, and for study or research purposes, or any other legal use. With this aim in mind, the University undertakes the following duties and reserves the following powers:

a) The University shall inform the archive users of the permitted uses; however, it shall not guarantee or take any responsibility for any other subsequent ways the work may be used by users, which are non-compliant with the legislation in force. Any subsequent use, beyond private copying, shall require the source to be cited and authorship to be recognized, as well as the guarantee not to use it to gain commercial profit or carry out any derivative works.
b) The University shall not review the content of the works, which shall at all times fall under the exclusive responsibility of the author and it shall not be obligated to take part in lawsuits on behalf of the author in the event of any infringement of intellectual property rights deriving from storing and archiving the works. The author hereby waives any claim against the University due to any way the users may use the works that is not in keeping with the legislation in force.
c) The University shall adopt the necessary measures to safeguard the work in the future.
d) The University reserves the right to withdraw the work, after notifying the author, in sufficiently justified cases, or in the event of third party claims.

Madrid, on 21…….. of August 2019……..., …….

HEREBY ACCEPTS

Signed……José Ignacio Ríos Goytre……

Reasons for requesting the restricted, closed or embargoed access to the work in the Institution's Repository

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título "Exploration of Surrogate Models for Remaining Useful Life Estimation" en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2018-19 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: José Ignacio Ríos Goytre Fecha: 21/08/2019

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Lehrstuhl für
PRODUKTENTWICKLUNG UND LEICHTBAU
Dr.- Ing. Markus Mörtl
Akademischer Oberrat

Fdo.: Prof. Dr. Markus Zimmerman Fecha: 23/08/ 2019

Technische Universität München
Lehrstuhl für Produktentwicklung und Leichtbau
Prof. Dr. Markus Zimmermann
Boltzmannstr. 15 - 85748 Garching

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

EXPLORATION OF SURROGATE MODELS FOR REMAINING USEFUL LIFE ESTIMATION

Autor: José Ignacio Ríos Goytre

Director: Prof. Dr. Markus Zimmermann

Co-Director: M.Sc. Simon Pfingstl

Madrid

Agosto de 2019

EXPLORATION OF SURROGATE MODELS FOR REMAINING USEFUL LIFE ESTIMATION

Autor: Ríos Goytre, José Ignacio.
Director: Zimmerman, Prof. Dr. Markus
Entidad colaboradora: Technische Universität München

EXTRACTO
Los componentes de las estructuras de aeronaves soportan cargas cíclicas durante su operación, lo cual conlleva efectos de fatiga como el crecimiento de las grietas. Las estimaciones de la Vida Útil Restante (RUL) son por lo tanto esenciales para programar reparaciones. En este trabajo se busca estimar la RUL a través de mediciones de deformación en la pieza. Los modelos propuestos (SVR y RNN) se entrenan en datos simulados y puestos a prueba en los datos reales de la base de datos de Virkler (base de datos ampliamente utilizada para el análisis estadístico del crecimiento de grieta).

1 INTRODUCCIÓN

La integridad estructural de los aviones debe ser asegurada en todo momento durante su operación. Los componentes del mismo trabajan bajo condiciones de carga por fatiga, lo cual requiere los correspondiente test de fatiga de los materiales empleados. Con estos test se puede formar una base estadística sólida y comprender mejor el comportamiento del material. La investigación está enfocada en el estudio de la aleación de aluminio 2024-T3 y el crecimiento de grieta en él.

El crecimiento de grieta por fatiga se analiza a partir del trabajo realizado por Virkler *et al.* [VIRK78], el cual ha servido históricamente de plataforma para comparar distintos modelos estadísticos para analizar el crecimiento de grieta por fatiga. El espécimen analizado en su trabajo disponía una grieta en el centro, siendo sometido a una carga cíclica homogénea.

Partiendo de su análisis y resultados, el objetivo de la investigación fue predecir la Vida Útil Restante (RUL) en cada punto de crecimiento de grieta, ofreciendo un intervalo de confianza para estas predicciones, habiendo entrenado los modelos exclusivamente con datos generados a través de Métodos Extendidos de Elementos Finitos (XFEM). Para hacer el estudio más similar a aplicaciones reales, no se utiliza la longitud de grieta, sino el deformación en un punto del espécimen estratégicamente elegido. Esta deformación se calcularía también a través de simulaciones de grieta en XFEM.

Los modelos que hasta ahora se han empleado para tratar de predecir la RUL son muchos y variados, pero se pueden clasificar de acuerdo a dos criterios: el tipo de datos disponible [BARU18] y el peso de el conocimiento previo sobre el fenómeno físico [KHEL16]. Los posibles tipos de datos disponibles son datos del tiempo de vida (sólo los tiempos de fallo se conocen), datos frontera (hay un límite definido, a partir del cual el espécimen se considera que ha fallado) y datos hasta el fallo (se conocen las medidas durante la vida completa del espécimen). En los test realizados por Virkler, se establece una longitud de grieta límite (derivada del factor de intensidad de estrés crítico), por lo que el segundo caso es el que aplica a la investigación. El peso del conocimiento previo sobre el fenómeno físico también juega un rol importante a la hora de diseñar un modelo adecuado para predecir la RUL. Si el fenómeno físico y el comportamiento del espécimen son ampliamente conocidos, se pueden incluir más suposiciones a la hora de diseñar el modelo. Si el fenómeno físico no se comprende profundamente, los datos medidos sobre el comportamiento pueden servir para entrenar a un modelo que desarrolle una intuición sobre las leyes subyacentes que lo rigen. En un tercer planteamiento, se puede usar una mezcla de ambos. Si se tiene profundo conocimiento sobre el fenómeno físico y se conocen las leyes

que lo rigen, los parámetros de estas leyes se pueden hallar de forma experimental. En la investigación se desarrollan dos modelos que emplean los datos disponibles para hacer predicciones. Estos modelos son Redes Neuronales Recurrentes (RNN) y Regresión con Vectores de Soporte (SVR).

Con los modelos seleccionados se tiene en cuenta que el crecimiento de grieta por fatiga es un problema de series temporales. Con el modelo de SVR, las series temporales se analizan a través de ventana deslizante, Con el modelo de RNN, todos los puntos previos al estudiado son considerados. SVR se basa en los principios de Minimización del Riesgo Estructural, lo cual asegura un potencial de generalización del modelo. Las RNN con células LSTM tienen la habilidad de capturar dependencias temporales en las secuencias.

2 METODOLOGÍA

El objetivo final del trabajo fue predecir la RUL sobre los datos reales obtenidos por Virkler *et al.* (ofreciendo un intervalo de confianza en la predicción de cada punto) con los dos modelos (SVR y RNN) entrenados con datos de simulaciones en XFEM. Para lograr este objetivo final, la generación de datos y el desarrollo de los modelos de Machine Learning ya mencionados se organizó el trabajo de la siguiente manera:

1. Literatura y desarrollo del conocimiento en mecánica de fractura y fatiga
2. Generación de datos analíticos a través de la ley de Paris
3. Desarrollo del modelo de RNN con los datos analíticos
4. Desarrollo del modelo de SVR con los datos analíticos
5. Generación de los datos de XFEM
6. Desarrollo del modelo de RNN con los datos de XFEM
7. Desarrollo del modelo de SVR con los datos de XFEM
8. Estimación de los datos reales (del trabajo de Virkler *et al.*) con el modelo de RNN entrenado en los datos de XFEM
9. Estimación de los datos reales (del trabajo de Virkler *et al.*) con el modelo de SVR entrenado en los datos de XFEM
10. Comparar el rendimiento de ambos modelos, su potencial de generalización y encontrar limitaciones

Los datos analíticos se generaron a través de la ley de Paris, dando como resultado secuencias de longitudes de grieta (aún no secuencias de deformación, este paso se daría al generar los datos con XFEM). Los parámetros de la ley de Paris se obtuvieron a través de el análisis estadístico del trabajo el mismo por Wang *et al.* [WANG17]- Los valores óptimos de estos parámetros fueron: $m = 2.9$ y $C_{med} = 8.586 * 10^{-11}$ y $C_{std} = 0.619 * 10^{-11}$ (distribución normal). A partir de estos datos, se desarrollaron los modelos SVR y RNN.

Una vez desarrollados los modelos y comprobada su viabilidad, se generaron las secuencias de datos a través de XFEM. Esta vez no se generarían secuencias de longitud de grieta, sino la evolución de la deformación. Para obtener estos resultados, se simuló la pieza empleada por Virkler, con la grieta en el medio. La evolución dinámica de la grieta realizada con XFEM provee la secuencia longitud de grieta-ciclo, pero requiere varias horas por simulación en los equipos disponibles. Esto imposibilitaría la posibilidad de realizar una cantidad significativa de simulaciones que utilizar para entrenar los modelos. La aproximación analítica ofrecía resultados suficientemente similares a la aproximación por XFEM como para sacar conclusiones aplicables. Así, a partir de las secuencias generadas analíticamente se calculó el valor de la deformación para cada longitud de grieta. La deformación se calculó en un punto estratégicamente colocado que pudiese capturar las mayores variaciones en deformación y de la forma más linear posible. El punto escogido presenta la siguiente relación entre longitud de grieta y deformación.

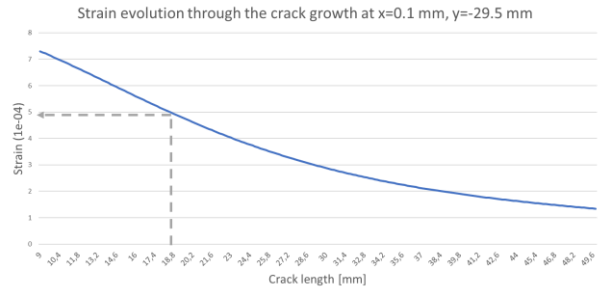


Figura 1: Relación longitud de grieta – deformación en el punto escogido

De esta forma, los valores de la longitud de grieta en las secuencias generadas (así como en las obtenidas por Virkler) fueron sustituidas por los correspondientes valores de deformación, obteniendo la evolución de la deformación. Con el fin de emular el comportamiento de las galgas extensométricas que medirían este valor en test reales, se añadió un error a los valores de deformación en cada punto. A cada valor de deformación se le añadió un valor aleatorio extraído de una distribución normal con media cero. EL resultado fueron 1000 ejemplos de evolución de la deformación generados, que se representan en la Figura 2.

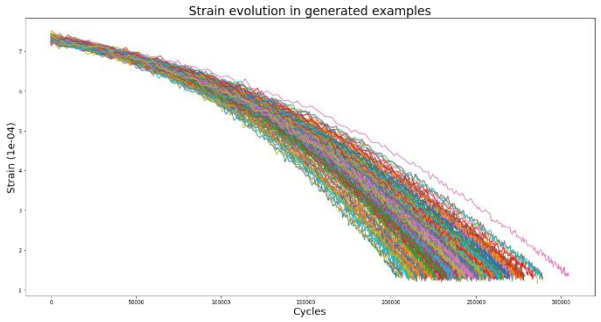


Figura 2: Secuencias generadas de evolución de la deformación

Los dos algoritmos desarrollados reciben estos datos, pero de dos formas distintas.

El modelo SVR opera con un número fijo de entradas, por lo que las secuencias fueros separadas en ventanas de longitud fija (24553 ciclos, alrededor de 10 ventanas por secuencia), y de cada ventana se extraían la media y la inclinación, de tal forma que estos dos valores se usen como entrada para el modelo SVR. La media se calculó como el valor medio de todos los puntos incluidos en la ventana y la inclinación a través de un regresión lineal dentro de la ventana. La salida de este modelo era un único valor de estimación de la RUL, por lo que para realizar estimaciones más fiables se desarrolló un predictor de la distribución. Para obtener esta distribución a estimar, se asumió una distribución normal y se empleó el modelo de K-vecinos más próximos (KNN) para predecir el error cometido por el modelo SVR. A través de el error cometido en estimaciones previas por el modelo SVR, el modelo KNN fue entrenado para recibir como entrada la media y la inclinación en cada ventana (igual que el SVR) y predecir el error cometido por el modelo SVR. Esta predicción del error se utilizaría como varianza de la distribución predicha, y la predicción del SVR como media.

El modelo RNN permite recibir como entrada vectores de longitud variable, por lo que se evita tener que usar ventanas en esta ocasión. Se introducen los valores de deformación (y los ciclos) directamente al modelo. Para lograr que reciba entradas similares al anterior modelo, por cada ventana de puntos que recibía el modelo SVR, este recibe esos mismos puntos y los previos correspondientes a esa secuencia. Con estos datos, se diseñó el modelo RNN para recibir una cantidad de valores variable y salir dos valores. La primera salida correspondería a la media de la distribución predicha, y la segunda la desviación típica. La función de coste empleada fue la log-verosimilitud negativa, la cual refleja dónde se sitúa el valor real de la RUL en la distribución predicha. En cuanto a la estructura del modelo RNN, consistía en una única capa oculta compuesta por celdas LSTM con un tamaño de vector del estado oculto de 4. Los LSTM solucionan un problema recurrente con las RNN: el desvanecimiento del gradiente, el cual resulta en la imposibilidad de recoger las dependencias a largo plazo.

3 RESULTADOS

Con el problema y las estructuras de los modelos definidos, se ajustaron los hiperparámetros de los modelos y se evaluaron los resultados. Las predicciones de los modelos fueron evaluadas con distintas criterios: R^2, MAE (ambos con respecto a la media predicha) and MPDF (con respecto a toda la distribución predicha). Estas tres métricas son explicadas en el trabajo completo, por lo que aquí solo se mencionan sus respectivas formulas:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \hat{Y}_i|$$

$$MPDF = \frac{1}{n}\sum_{i=1}^{n} PDF = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{\sqrt{2\pi\hat{\sigma}_i}} \cdot e^{-\frac{(Y_i - \hat{\mu}_i)^2}{2\hat{\sigma}_i^2}}$$

Los resultados también se analizaron visualmente para comprender mejor la calidad de las predicciones.

El ajuste de hiperparámetros del modelo SVR se realizó en base a R^2, y el resultado fue la elección de una función kernel RBF con $\gamma = 0.5$, $C = 10^3$ y $\varepsilon = 0.01$. En cuanto al modelo de varianza KNN, el número elegido de vecinos fue 500. Los resultados del modelo en los datos da prueba (‘test set’), compuesto por las 68 secuencias de la base de datos de Virkler, se presentan en .

Tabla 1: Resultados del modelo SVR en los distintos datos

Metric	Training	Validation	Test
R^2	0.981	0.989	0.966
MAE [cycles]	6347.3	3441.9	5628.5
MPDF	18.6	33.4	17.1

Los resultados en los datos de validación (‘Validation’) son mejores que los de entrenamiento (‘Training’) debido a la forma en que fueron generados. Los datos de validación se generan con una Ventana deslizante por cada punto, empezando por el final de cada secuencia, y las secuencias tienen mayor densidad de puntos cuanto más Avanza el proceso. Esto conlleva más ventanas al final de la secuencia en los datos de validación. Las predicciones al final de las secuencias son mejores, e implican mejores resultados en las tres métricas. En cuanto a los datos de prueba (‘test set’), el coeficiente de determinación es $R^2 = 0.966$, lo cual sugiere que el modelo SVR está aproximando bien la RUL de las observaciones. En cuanto al MAE, los resultados son peores que en los datos de validación, pero el valor de 5628.5 ciclos de media puede ser aún aceptable. Por último, la MPDF ofrece unos resultados peores que en validación, lo cual indica que, probabilísticamente, el modelo no está prediciendo la RUL tan bien como en los datos de validación. Evaluando visualmente, las predicciones (se representa una de las secuencias completa) son representadas en la Figura 3. Ambas gráficas comparten el eje de abscisas (ciclos transcurridos dentro de esa secuencia). La primera representa en naranja el último ciclo de la secuencia y en azul el intervalo de confianza del 99.7% predicho. La segunda gráfica muestra la evolución del PDF (probabilidad) de los verdaderos valores de RUL en la distribución normal predicha.

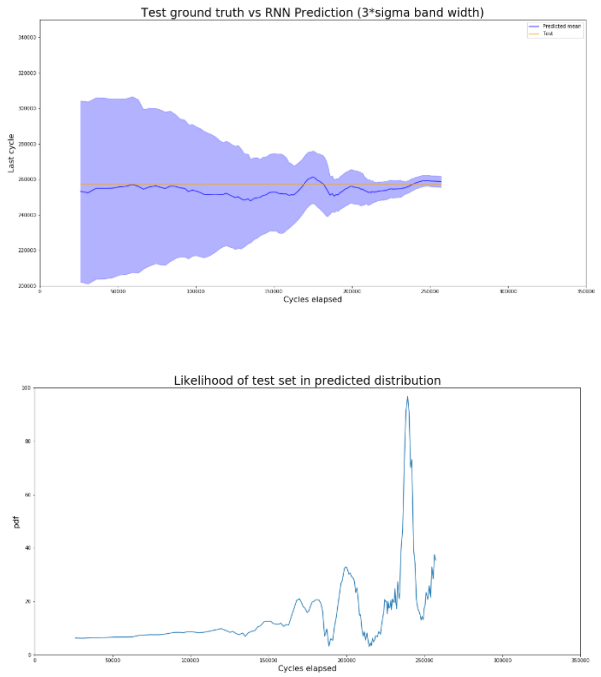


Figura 3: Graficas de evaluación de la mejor muestra predicha con el modelo SVR

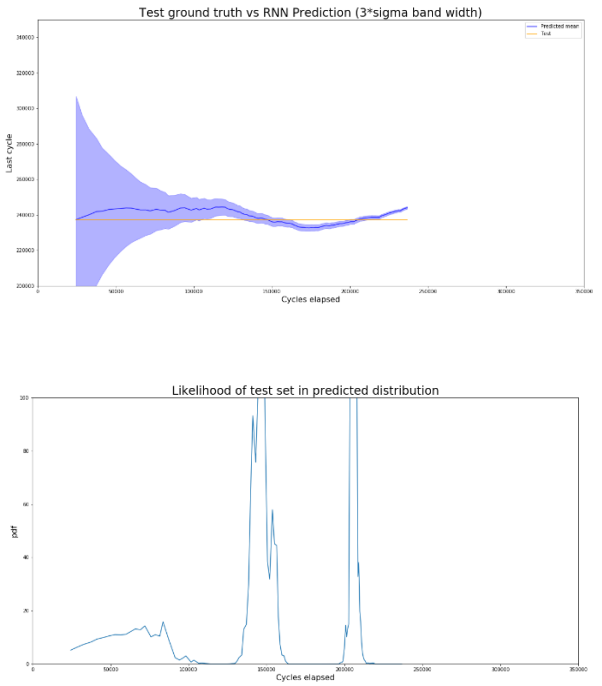


Figura 4: Graficas de evaluación de la mejor muestra predicha con el modelo RNN

En cuanto al modelo RNN, se escogió in tamaño de lote de 256. El resto de hiperparámetros, relacionados con el optimizador (Adam), se determinó que fueran $\alpha = 0.005$, $\beta_1 = 0.9$ y $\beta_2 = 0.999$. Los resultados del modelo en los datos da prueba ('test set'), compuesto por las 68 secuencias de la base de datos de Virkler, se presentan en la Tabla 2.

Tabla 2: Resultados del modelo RNN en los distintos datos

Metric	Training	Validation	Test
R^2	0.989	0.995	0.945
MAE [cycles]	1498.7	1285.3	8718.0
MPDF	197.1	191.6	15.4

La cuestión de los resultados en los datos de validación y entrenamiento expuesta para el modelo SVR también aplica aquí. En cuanto a los datos de prueba ('Test'), los resultados muestran un peor rendimiento que en el modelo SVR. Al evaluar las predicciones visualmente, se comprueba que el modelo RNN no traslada lo aprendido en los datos de entrenamiento a las secuencias reales de la base de datos de Virkler.

En un intento por ayudar al modelo a generalizar mejor, se aplicó la regularización por 'dropout' de las entradas, lo cual no ofreció resultados satisfactorios, por lo que esta alternativa fue descartada. Finalmente, para comprobar si este modelo podría ofrecer resultados positivos al entrenarlo con datos reales de la base de datos de Virkler, en vez de simulaciones, se comprobaron los resultados con exactamente el mismo modelo. El modelo ofrece un buen rendimiento en este caso, lo que provee una visión optimista en cuanto a este potencial problema alternativo. Estos resultados se muestran en la Tabla 3 y Figura 5.

Tabla 3: Resultados del modelo RNN, entrenado con los datos de Virkler, en los distintos datos

Metric	Training	Test
R^2	0.982	0.983
MAE [cycles]	5056	5246
MPDF	19.2	19.9

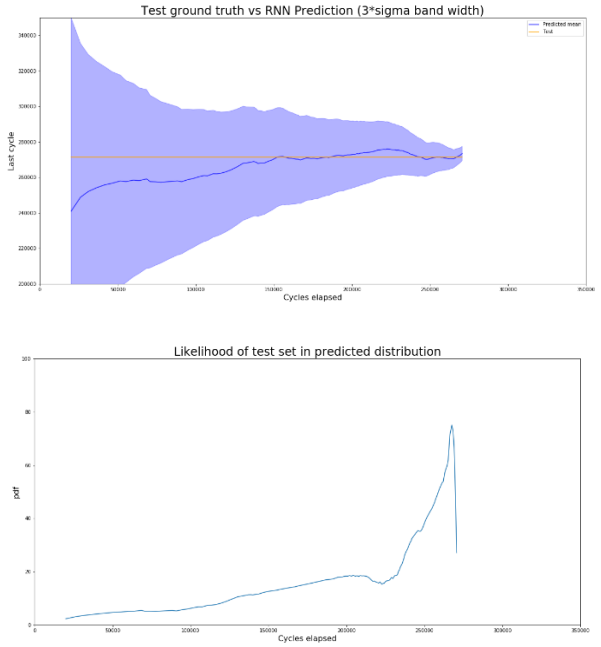


Figura 5: Graficas de evaluación de una muestra predicha con el modelo RNN, entrenado con los datos de Virkler

4 CONCLUSIONES

La principal cuestión a responder en la investigación era la posibilidad de estimar degradación real de forma probabilística mediante el uso de secuencias simuladas. La investigación se vio claramente favorecida por la información previa relativa a los datos de Virkler, pero de todas formas se comprobó su factibilidad. Si fuera posible, con un estudio sobre la variabilidad de los parámetros de la ley de Paris dentro de cada secuencia se podrían lograr aproximaciones más cercanas a la realidad y ofrecer mejores resultados. Adicionalmente, las mediciones "reales" de la deformación fueron creadas convirtiendo las longitudes de grieta en valores de deformación a través de XFEM. Con mediciones reales de deformación disponibles, los resultados a obtener serían mucho más fiables.

En cuanto a los propios modelos, es relevante mencionar que la normalización de la salida (RUL) supuso una importante mejora en el aprendizaje de ambos modelos. La evaluación visual también resulto ser vital, dado que las métricas no reflejan toda la verdad sobre el rendimiento de cada modelo, donde el modelo SVR resultó mucho más adecuado que el RNN para este caso. El modelo SVR fue originalmente seleccionado por su capacidad de generalización (se basa en los principios de Minimización del Riesgo Estructural, SRM), lo cual se tradujo en mejores resultados en datos reales. No obstante, los resultados obtenidos en los datos de entrenamiento y validación sugieren el uso de simulaciones más realistas favorecería al modelo RNN.

También hay un aspecto a resaltar sobre la base de datos de Vikler. Las secuencias de esta base de datos muestran un aceleración en la degradación (velocidad de crecimiento de grieta) al final de las secuencias que hace que los modelos predigan peor en los últimos instantes de cada secuencia. Esta aceleración repentina se podría explicar por el paso del proceso de crecimiento de grieta a la región III, en la que la propagación de la grieta ocurre de forma mucho más rápida. De ser este el caso, se recomienda establecer un límite de fallo más conservador.

El trabajo presentado pertenece a una investigación sobre la estimación de la RUL con datos simulados, la cual continúa con la exploración de un tercer modelo y centrándose en la longitud de grieta en vez de en la deformación. El entrenamiento de los modelos en datos reales podría ser un trabajo futuro que aportaría valor a la materia. Tal y como se demuestra con la RNN, este planteamiento puede ofrecer resultados satisfactorios y es una buena opción si los datos son disponibles. Asimismo, la mayoría de las propuestas en el campo de la propagación de grieta por fatiga se centran en el proceso completo de propagación de la grieta y no sólo en la estimación de la RUL. Algunos de estos modelos, basados en el análisis de series temporales, que han sido probados son ARIMA [SOLO91], Procesos Gaussianos [MOHA09] o ANN [MOHN09]. Podría ser interesante comparar los resultados de estos modelos con los empleados en este trabajo.

5 REFERENCIAS

[VIRK78] Virkler, Dennis Andrew, Brnm Hillberry, and P. K. Goel. "The statistical nature of fatigue crack propagation." Journal of Engineering Materials and Technology 101.2 (1979): 148-153.

[BARU18] Baru, Aditya. “Three Ways to Estimate Remaining Useful Life for Predictive Maintenance.” MATLAB & Simulink, MathWorks, 2018, https://www.mathworks.com/company/newsletters/articles/three-ways-to-estimate-remaining-useful-life-for-predictive-maintenance.html.

[KHEL16] Khelif, Racha, et al. "Direct remaining useful life estimation based on support vector regression." IEEE Transactions on industrial electronics 64.3 (2016): 2276-2285.

[WANG17] Wang, Wenyi, Weiping Hu, and Nicholas Armstrong. "Fatigue crack prognosis using Bayesian probabilistic modelling." Mechanical Engineering Journal 4.5 (2017): 16-00702.

[SOLO91] Solomos, G. P., and V. C. Moussas. "A time series approach to fatigue crack propagation." Structural Safety 9.3 (1991): 211-226.

[MOHA09] Mohanty, Subhasish, et al. "Gaussian process time series model for life prognosis of metallic structures." Journal of Intelligent Material Systems and Structures 20.8 (2009): 887-896.

[MOHN09] Mohanty, J. R., et al. "Application of artificial neural network for predicting fatigue crack propagation life of aluminum alloys." (2009).

ABSTRACT

The components of aircraft structure have to bear cyclic loading, which leads to fatigue effects, such as the growth of cracks. The estimations about the Remaining Useful Life (RUL) of the components becomes essential to program repairs. This work aims at estimating the RUL based on strain measurements. The models proposed (SVR and RNN) were trained on simulated data and tested on Virkler's dataset (widely used dataset for statistical analysis of crack growth).

1 INTRODUCTION

Structural integrity must be ensured during the operation of aircrafts. The components operate under conditions of fatigue loading, which requires appropriate fatigue testing of the materials employed. With these tests, a solid statistical base is created for the understanding of the material's behavior. The focus of the research is on 2024-T3 aluminum alloy and the fatigue crack growth on it.

The fatigue crack growth was analyzed mimicking Virkler's work [VIRK78], a common platform for comparing statistical approaches to fatigue crack growth. The specimen used in this work was a plate with a crack in the middle, under homogeneous cyclic loading.

Using the analysis and results from Virkler's work, the goal was to predict the Remaining Useful Life (RUL) at each point of the crack growth, with a confidence interval for the predictions, having trained the models only on generated data through Extended Finite Elements Method. In order to make the study more useful in real applications, the strain at a strategically placed point of the specimen was used as input for the models. This strain was calculated by XFEM crack simulation.

The models employed to estimate Remaining Useful Life are various, but they can be classified according to two criterion: the type of data available [BARU18] and the weight of prior physical knowledge [KHEL16]. The possible available types of data are lifetime data (only the times of failure are known), threshold data (threshold defined, specimen deemed to have failed if the threshold is surpassed) and run-to-failure data (the measurements of the whole lifetime of the specimens are known). In the studied case, the available data from Virkler's dataset has a threshold crack length (derived from the critical stress intensity factor), so the second case is the one that applies to the research. The weight of prior physical knowledge also plays a role in the designing of an adequate model to predict RUL. If the phenomenon and the physical behavior are deeply understood, more assumptions will be introduced when designing the model. If the phenomenon is not deeply understood, the best approach is to use data to train a model that will develop an intuition of the underlying behavior. A third case a mixture of both, where physical knowledge about the phenomenon is available, but the parameters of the model are tuned through real data. In the research, data-driven models are employed, such as Recurrent Neural Networks (RNN) and Support Vector Regression (SVR).

The models were selected and train bearing in mind that the degradation through crack growth in fatigue is a time series problem. With the SVR model, the time series are analyzed by sliding window. With the RNN model, all the previous points to the current one are considered. SVR is based Structural Risk Minimization principles, which grant a good generalization potential. RNNs with LSTM cells can capture the time dependencies.

2 METHODOLOGY

The final goal was estimating the RUL of the real data obtained by Virkler *et al.* (by providing a confidence interval at each point) with the two models (RNN and SVR) trained on simulated data from XFEM. In order to reach this final goal, the data generation and the machine learning algorithms development was organized as follows:

11. Literature and developing an understanding of fracture mechanics and fatigue
12. Generate analytical data (Paris' law)

13. Develop RNN model with analytical data
14. Develop SVR model with analytical data
15. Generate XFEM data
16. Develop RNN model with XFEM data
17. Develop SVR model with XFEM data
18. Estimate real data (from Virkler's tests) with RNN model trained on XFEM data
19. Estimate real data (from Virkler's tests) with SVR model trained on XFEM data
20. Compare performance of both models, generalization potential and find limitations

The analytical data was generated using Paris' law, resulting in crack growth sequences (not strain sequences yet, this step will be taken when generating the XFEM data). The parameters from Paris' law were obtained through the statistical analysis of Virkler's dataset by Wang *et al.* [WANG17]. The optimal values for these parameters were: $m = 2.9$ and $C_{med} = 8.586 * 10^{-11}$ and $C_{std} = 0.619 * 10^{-11}$ (normal distribution). Then, the SVR model and the RNN model were developed to make predictions on this data.

Once proven feasible, the XFEM data was generated. For this step, the data generated wasn't the evolution of the crack length anymore, but the strain evolution. To obtain those results, Virkler's specimen with a crack in the middle was simulated. The dynamic crack growth of XFEM provided the sequence of crack length growth, but each simulation takes several hours to compute on the university's computers, which wouldn't allow building a big enough training set. The analytical approximation to this case had provided similar results to the XFEM simulations, so for this specimen geometry and loading conditions, the analytical approximation was taken as similar enough to XFEM's approximation to draw applicable conclusions.

Then, with the crack length sequences the values of the strain for each crack size were computed. The strain was obtained at a strategically chosen point, which would capture the biggest differences in the degradation process and was as linear as possible. The chosen point provided the following relationship between crack length and strain.

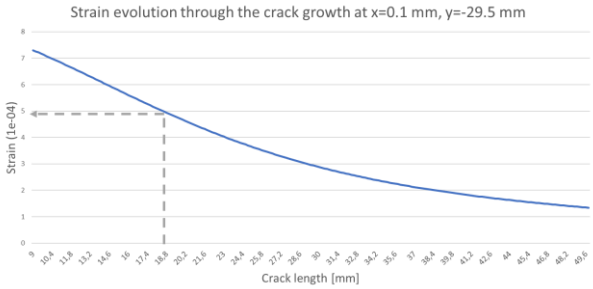


Figure 2: Crack length – strain relationship at chosen point

This way, the values of crack length in the generated sequences (as well as in Virkler's dataset) were swapped with their corresponding strain values, obtaining the strain evolution. To replicate strain gauge measurements of the strain, an error was added to the strain values at each point. Each strain value was added a random value extracted from a normal distribution with mean zero. The resulting 1000 generated examples of strain evolution are displayed in Figure 1.

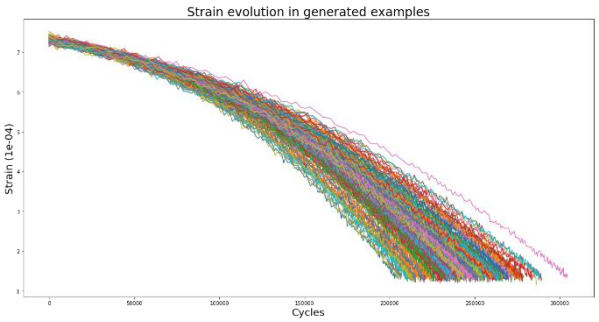


Figure 1: Generated strain sequences

The two algorithms were fed this data in different ways, according to their capabilities.

The SVR model operates with a fixed number of inputs, so the sequences were split in windows on a certain length (24553 cycles, around 10 windows per sequence) and the mean and slope were extracted from each window, being used these two values as inputs for the SVR model. The mean was calculated as the average of the values included in the window, and the slope was approximated by a linear regression within the window. This model outputs a single value for the RUL

estimation, but in order to make safer estimations, a distribution was also predicted. To build this predicted distribution, a normal distribution was assumed and a K-Nearest Neighbors (KNN) model was used to estimate the errors of the SVR predictions. Through the errors of previous estimations by SVR, the KNN model was fed the same mean and variance, and predicted the error that the SVR model would make. This error prediction was used as variance of the prediction distribution, and the SVR prediction as the mean.

The RNN model can be fed variable lengths of inputs, avoiding having to employ windows on the sequences this time. The strain values (and the cycles) were fed directly to the RNN. In order to make the input as similar as possible as the ones provided to the SVR model, per each window of points that the SVR was fed, the RNN was fed the same points in that window, as well as the previous points of the sequence. With this data, the RNN model was then designed to have multiple inputs and two outputs. The first output would be responsible of providing the mean of the distribution, and the second output would provide the standard deviation. The cost function employed was the negative log-likelihood, which reflects where the true value of RUL falls in the predicted distribution. As for the structure of the RNN model, it consisted on one hidden layer composed of LSTM cells with a hidden state size of 4. The LSTM solve a recurrent problem of the RNNs, which are the vanishing gradients, which result on not being able to capture long term dependencies. Luckily, with LSTM cells this problem is tackled.

3 RESULTS

With the problem and the structures of the models to employ completely defined, the hyperparameters of the models were then tuned, and the results evaluated. The outcome of the models were evaluated through three different metrics: R^2, MAE (both with respect to the mean prediction) and MPDF (with respect to the whole distribution prediction). The three metrics are explained in the dissertation, so only their equations are presented here:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|Y_i - \hat{Y}_i\right|$$

$$MPDF = \frac{1}{n}\sum_{i=1}^{n} PDF = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\sqrt{2\pi\hat{\sigma}_i}}\cdot e^{-\frac{(Y_i-\hat{\mu}_i)^2}{2\hat{\sigma}_i{}^2}}$$

The results were also assessed visually to gain a better understanding of the quality of predictions.

The tuning of the hyperparameters of the SVR model for the mean was performed following the scores on R^2, and resulted in the selection of a RBF kernel with $\gamma = 0.5$, $C = 10^3$ and $\varepsilon = 0.01$. As for the variance KNN model, the selected number of neighbors was $k = 500$. The scores of the model on the test set composed of the 68 Virkler's dataset sequences are presented in Table 1.

Table 1: Scores of the SVR model on the different sets

Metric	Training	Validation	Test
R^2	0.981	0.989	0.966
MAE [cycles]	6347.3	3441.9	5628.5
MPDF	18.6	33.4	17.1

The validation set's results are better than the training set due to the nature of each data. The validation sets were obtained with a sliding window for each point starting at the end of the sequences, and the sequences have more density of points towards the end. This results in more windows towards the end in the validation sets. The predictions at the end are better than at the beginning, and therefore the scores for the three metrics are better overall. As for the test set, the first metric delivers a score of $R^2 = 0.966$, which suggests that the SVR mean model is approximating the observations of RUL well. As for the MAE, the model performs worse than the validation set,

but the value of 5628.5 cycles in offset on average can still be acceptable. Lastly, the MPDF shows a lower result than the validation sets, which indicates that the model isn't capturing the true RUL as good as in the validation sets. When assessing visually, the predictions (one full sequence represented) looked like **¡Error! No se encuentra el origen de la referencia.**Figure 3. Both graphs share the x-axis (cycles elapsed within that complete sequence). The first one represents in orange the last cycle of the sequence and the confidence interval of 99.7% in blue. The second graph shows the evolution of the PDF value (likelihood) of the true points in the normal distribution prediction.

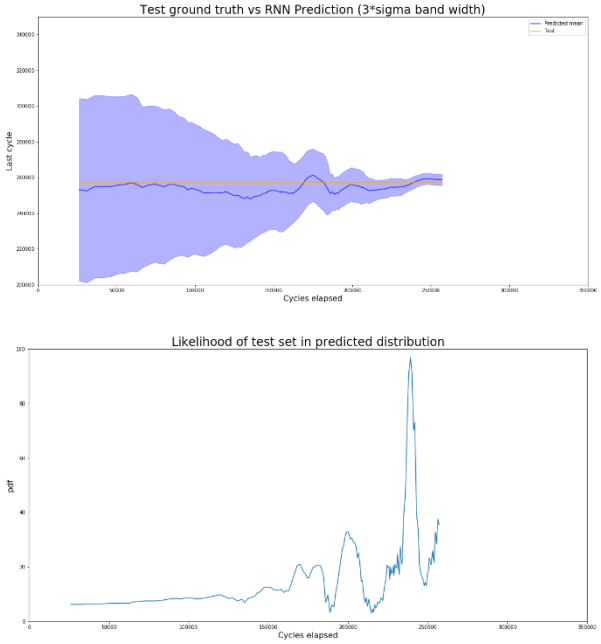


Figure 3: Evaluation graphs on best SVR prediction sample of test set

As for the RNN model, the chosen batch size was 256. As for the hyperparameters regarding the optimization (Adam) the chosen hyperparameters were $\alpha = 0.005$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The scores of the model on the test set composed of the 68 Virkler's dataset sequences are presented in Table 2.

Table 2: Scores of the RNN model on the different sets

Metric	Training	Validation	Test
R^2	0.989	0.995	0.945
MAE [cycles]	1498.7	1285.3	8718.0
MPDF	197.1	191.6	15.4

The same issue as mentioned on SVR occurs with regards to the scores in validation and train set. As for the test set, the metrics showed a worse performance than the SVR. When assessing visually (Figure 4) it becomes obvious that the RNN didn't do a good job at translating what was learned from the simulated sequences to the real sequences on Virkler's dataset.

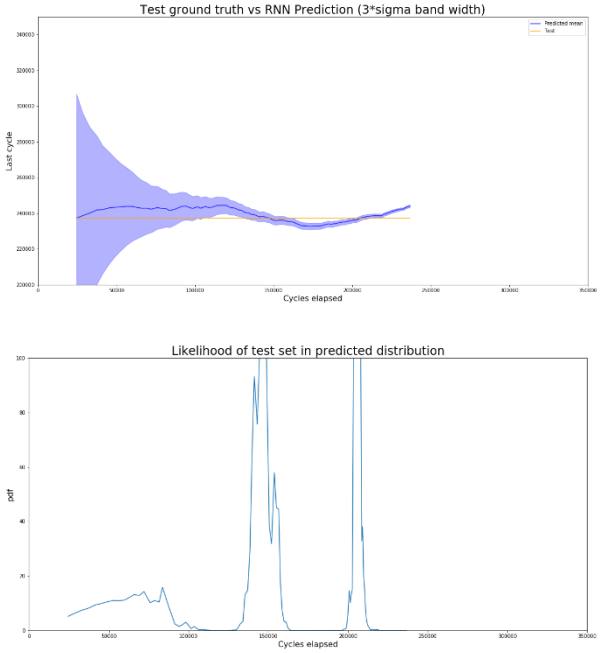


Figure 4: Evaluation graphs on best RNN prediction sample of test set

In an attempt to help the model generalize better, dropout regularization was performed on the inputs. This didn't improve the results, so this alternative was discarded. Finally, to check if the RNN model could perform well if trained on Virkler's dataset instead of simulations, this approach was taken using the same model. The model performs well in this case, offering an optimistic view for this potential approach. The results are shown in Table 3 and Figure 5.

Table 3: Scores of the RNN model trained on Virkler's data on the different sets

Metric	Training	Test
R^2	0.982	0.983
MAE [cycles]	5056	5246
MPDF	19.2	19.9

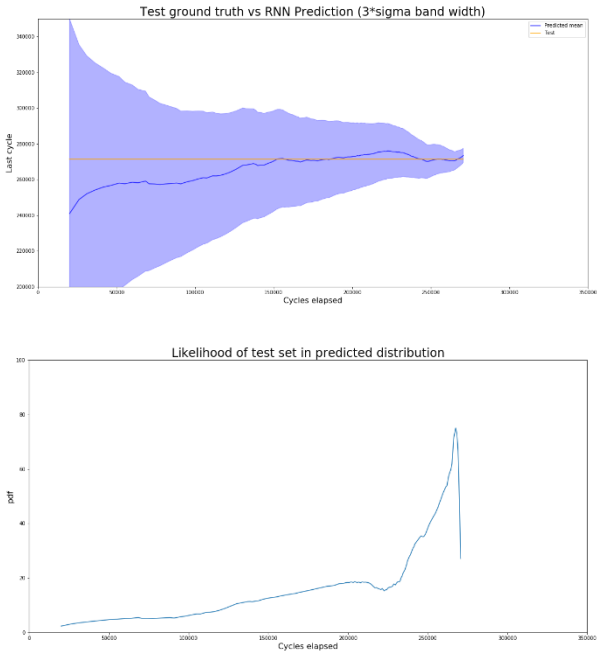


Figure 5: Evaluation graphs on RNN (trained on Virkler's data) predictions on sample sequence of the test set

4 CONCLUSIONS

The main question to answer on the research was if it was possible to estimate real degradation behavior probabilistically by using simulated (or analytical) sequence. This was obviously benefited from previous information about Virkler's data, but nevertheless it was shown to be feasible. If possible, a further study on the variability of Paris' law's parameters within the points of each sequence might provide better approximations and deliver better results in reality. Additionally, the 'real' strain measurements were generated by converting crack length measurements to strain through XFEM. If strain measurement data is available, the results obtained would be much more reliable.

As for the models themselves, it is worth noting that one of the features that improved the performance of both algorithms significantly was the normalization of the output (RUL). Visual assessment of the predictions was also proven to be vital, as the metrics will never reveal the whole truth about the performance, which was significantly better for the SVR model than the RNN model. The SVR model was selected for its generalization potential (it is based on the Structural Risk Minimization principles) and translated to better performance on the real data. Nevertheless, the results on training and validation also suggest that more realistic data would favor the RNN model.

There was also an aspect about Virkler's dataset that should be considered. The sequences from this dataset showed a sudden acceleration in the degradation (crack growth rate) towards the end of the sequences that made all the tested models perform worse at the very end. This acceleration could be explained by the crack growth process entering the region where the crack grows much more rapidly (region III). If this was the case, a more conservative threshold might have to be considered.

The work presented belongs to a research on RUL estimation for simulated data that will continue by developing a third model and focusing on the crack length instead of the strain. Further work that would be useful to perform is the training of models on real data. As shown in the RNN, this approach can have successful results and is a good option if the data is available. Also, most of the approaches to fatigue crack growth try to model the whole propagation process, not just the RUL. Some of these models, based on time series analysis, that have been tested are: ARIMA [SOLO91], Gaussian Process [MOHA09] or ANN [MOHN09]. It would be interesting to compare the performance of these models with the ones shown in this work.

5 REFERENCES

[VIRK78] Virkler, Dennis Andrew, Brnm Hillberry, and P. K. Goel. "The statistical nature of fatigue crack propagation." Journal of Engineering Materials and Technology 101.2 (1979): 148-153.

[BARU18] Baru, Aditya. “Three Ways to Estimate Remaining Useful Life for Predictive Maintenance.” MATLAB & Simulink, MathWorks, 2018, https://www.mathworks.com/company/newsletters/articles/three-ways-to-estimate-remaining-useful-life-for-predictive-maintenance.html.

[KHEL16] Khelif, Racha, et al. "Direct remaining useful life estimation based on support vector regression." IEEE Transactions on industrial electronics 64.3 (2016): 2276-2285.

[WANG17] Wang, Wenyi, Weiping Hu, and Nicholas Armstrong. "Fatigue crack prognosis using Bayesian probabilistic modelling." Mechanical Engineering Journal 4.5 (2017): 16-00702.

[SOLO91] Solomos, G. P., and V. C. Moussas. "A time series approach to fatigue crack propagation." Structural Safety 9.3 (1991): 211-226.

[MOHA09] Mohanty, Subhasish, et al. "Gaussian process time series model for life prognosis of metallic structures." Journal of Intelligent Material Systems and Structures 20.8 (2009): 887-896.

[MOHN09] Mohanty, J. R., et al. "Application of artificial neural network for predicting fatigue crack propagation life of aluminum alloys." (2009).

Index

1. Introduction and project approach 9

2. Description of the technologies (State of the Art) 13

2.1. Probabilistic approach to RUL through Surrogate Models 13

2.1.1. Type of data available 13

2.1.2. Weight of physical knowledge 14

2.2. Extended Finite Element Method 20

3. Description of the developed models 25

3.1. Goals and specification 26

3.2. Data 27

3.2.1. Virkler's dataset 27

3.2.2. Paris Law Approximation 30

3.2.3. Strain and XFEM 31

3.3. Algorithms 40

3.3.1. SVR 40

3.3.1.1. Structural Risk Minimization 40

3.3.1.2. Support Vector Classification 42

3.3.1.3. Generalization of the Optimal Separating Hyperplane 46

3.3.1.4. Higher Dimensional Feature Space 48

3.3.1.5. Kernel Functions 49

3.3.1.6. Support Vector Regression 50

3.3.1.7. Distribution Estimation 53

3.3.1.8. Training, validation and test sets 54

3.3.2. RNN 57

3.3.2.1. Neural Networks Review 57

3.3.2.2. Recurrent Neural Networks 59

3.3.2.3. LSTMs 61

3.3.2.4. Mixture Density Network 64

3.3.2.5. Training, validation and test sets 65

4. Results analysis 67

4.1. SVR 68

4.1.1. Parameters 68

4.1.2. Results 74

4.2. RNN 81

4.2.1. Hyperparameters 81

4.2.2. Results 82

5. Conclusions 89

5.1. Methodology conclusions 89

5.2. Results conclusions 90

5.3. Recommendations for future work 91

6. Literature 93

7. Appendix I – Crack length data generation 97

8. Appendix II – Strain data generation 101

9. Appendix III – Strain data from Virkler's tests 103

10. Appendix IV – SVR model 105

11. Appendix V – RNN model 119

List of figures

Figure 1: Materials used in Boeing 787 Dreamliner [DOOR15] 9

Figure 2: Support Vector Machine for classification [IVAN14] 11

Figure 3: Recurrent Neural Network vs regular Neural Network [XPER18] 11

Figure 4: Types of data available for RUL estimation. From left to right: Lifetime Data, Threshold Data and Run-to-Failure Data [BARU18] 14

Figure 5: The three fracture modes [TURK16] 15

Figure 6: Fatigue crack growth rate [MECH] 15

Figure 7: Crack closure form Elber's theory 17

Figure 8: Windows associated mean, slope and RUL [KHEL16] 18

Figure 9: RUL estimation through sliding window [KHEL16] 19

Figure 10: Kalman Filter estimations [WANG17] 20

Figure 11: Crack crossing finite elements [DASS09] 21

Figure 12: Heaviside function [DASS09] 22

Figure 13: Crack Tip Enrichment Functions [DASS09], modified 22

Figure 14: Phantom Node Approach [MOHA14] 23

Figure 15: Level Set Method [DASS09], modified 23

Figure 16: Damage stages [DASS09], modified 24

Figure 17: Typical linear (a) and nonlinear (b) traction-separation response [DASS09] . 24

Figure 18: Remaining Useful Life Surrogate Model functional overview 25

Figure 19: Virkler's Test Specimen (dimensions in inches) [VIRK78] 28

Figure 20: Crack growth sequences from Virkler's dataset 30

Figure 21: Crack length increment convergence for analytical data generation 32

Figure 22: Histograms of the sampled Paris' law parameters 33

Figure 23: Crack growth analytically generated examples 34

Figure 24: Histogram of the number of cycles elapsed on the analytically generated examples 34
Figure 25: Simulated piece on XFEM based on Virkler's tests 34
Figure 26: Longitudinal strain through crack lengths on the simulated piece 35
Figure 27: Strain distributions at half piece per crack length on the simulated piece 36
Figure 28: Strain evolution through crack lengths near the crack on the simulated piece 37
Figure 29: Strain difference along Y-axis on the simulated piece 38
Figure 30: Strain evolution at chosen point on the simulated piece 38
Figure 31: Generated “measured” strain sequences 39
Figure 32: Strain sequences derived from Virkler's dataset 39
Figure 33: Bound on the risk based on the empirical risk and the VC dimension 42
Figure 34: SVM Optimal Separating Hyperplane [GHAN12] 42
Figure 35: Canonical Hyperplanes [GUNN98] 43
Figure 36: Constrained Canonical Hyperplanes [GUNN98] 44
Figure 37: Generalized Optimal Separating Hyperplane 46
Figure 38: Nonlinear Boundary through Higher Dimensional Space [AMID18] 48
Figure 39: Loss Funtions [GUNN98] 50
Figure 40: Support Vector Regression [MOUS18] 51
Figure 41: SVR distribution estimator 53
Figure 42: Complete generated sequence split by non-overlapping windows starting at the end of the complete sequence, training set 54
Figure 43: Extracted mean and slope of a window 55
Figure 44: Complete generated sequence split by sliding windows for each point starting at the end of the complete sequence , validation set 56
Figure 45: Basic Neuron and its corresponding computation graph 57
Figure 46: Shallow (a) and deep (b) neural network structures 58

Figure 47: Conventional vs Recurrent Neuron Structure 59
Figure 48: RNN architectures based on number of inputs and outputs 60
Figure 49: Standard RNN repeating cell [OLAH15], modified 61
Figure 50: LSTM repeating cell [OLAH15], modified 61
Figure 51: Cell state within an LSTM cell [OLAH15], modified 62
Figure 52: Forget gate in a LSTM cell [OLAH15], modified 62
Figure 53: Input Gate and Vector of Candidate Values in a LSTM cell [OLAH15], modified 63
Figure 54: Cell state calculation within an LSTM cell (https://colah.github.io/posts/2015-08-Understanding-LSTMs/, modified) 63
Figure 55: Output gate and hidden state calculation in a LSTM cell [OLAH15], modified 64
Figure 56: Sigmoid function 64
Figure 57: Exponential Linear Unit (ELU) and ELU+1 functions 64
Figure 58: RNN distribution estimator 65
Figure 59: Complete sequence split in partial sequences with 20 additional datapoints per partial sequence 66
Figure 60: Probability Density Function of a standard normal distribution (a) vs a sample normal distribution prediction (b) 68
Figure 61: RBF, Sigmoid and Polynomial kernels comparison through predictions versus the true RUL values with default values on Scikit-Learn 69
Figure 62: Epsilon values comparison through predictions versus the true RUL values 70
Figure 63: C values comparison through predictions versus the true RUL values 70
Figure 64: Gamma values comparison through predictions versus the true RUL values 71
Figure 65: R2 results of the analyzed KNN through 5 splits in Cross Validation 72

Figure 66: Sample of last cycle prediction by the SVR-KNN model on the variance validation set (K=120)..... 73
Figure 67: R2 results of the analyzed greater KNN through 5 splits in Cross Validation 73
Figure 68: Sample of last cycle prediction by the SVR-KNN model on the variance validation set (K=500)..... 74
Figure 69: Evaluation graphs on best SVR prediction sample of test set 76
Figure 70: Evaluation graphs on standard SVR prediction sample of test set 78
Figure 71: Evaluation graphs on standard SVR prediction sample of test set 80
Figure 72: Batch Gradient Descent (a) versus Mini-Batch Gradient Descent (b) with Adam optimization on the studied RNN model, 500 epochs 81
Figure 73: Sample of last cycle prediction by the RNN model on the validation set 82
Figure 74: Evaluation graphs on RNN prediction sample of test set 84
Figure 75: Evaluation graph on RNN (with 50% input dropout) prediction sample of test set 85
Figure 76: Evaluation graphs on RNN (trained with real data) prediction sample of test set 87
Figure 77: Fatigue crack growth rate [MECH] 91

List of tables

Table 1: Average experimental error [VIRK78], modified.....29

Table 2: Obtained parameters from Paris' law on different studies (with ΔK in MPa√m)31

Table 3: Crack length increment assessment through error on the number of cycles elapsed.....33

Table 4: Default values for the SVR parameters (Scikit-Learn).....69

Table 5: Selected values for the SVR parameters.....71

Table 6: Scores of the SVR model on the different sets (training 1&2, validation 1&2 and test).....75

Table 7: Scores of the RNN model on the different sets (training, validation and test) ...83

Table 8: Scores of the RNN (with 50% input dropout) model on the different sets (training, validation and test) 85

Table 9: Scores of the RNN model trained with real data on the different sets (training and test) 86

Table 10: Scores of the SVR and RNN models on the different sets (training, validation and test) 90

1. Introduction and project approach

Structural integrity must be ensured during the operation of aircrafts. The components of these aircrafts operate under conditions of fatigue loading, which can cause the material to fracture by progressive brittle cracking. In order to understand and be able to predict this phenomenon, appropriate fatigue testing of the materials employed is required. With these tests, a solid statistical base of the fracture properties of the material can be created. Various materials are employed for the body of aircraft structures, but the study focuses on aluminum.

Aluminum alloys are employed for aircraft structures in an important number of components due to their low weight. They have lately been partially replaced by composites, but they are still necessary for many reasons. Amongst them are the cost, the low weight and the technical development [MRAZ14]. As seen in Figure 1, they are preferred for locations where impact is possible. The alternative, composites, can weaken due to their leading edges and potential delamination. In Figure 1, the materials used for the body of the Boeing 787 are represented:

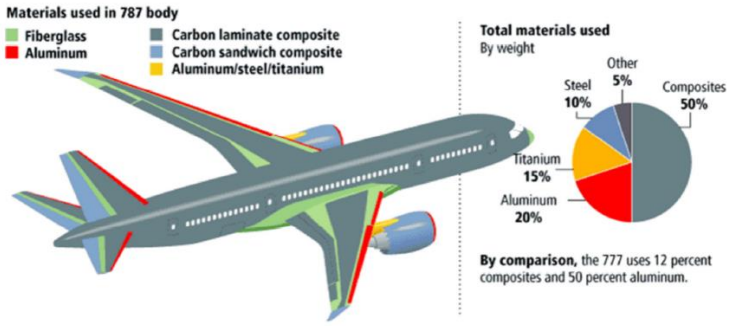


Figure 1: Materials used in Boeing 787 Dreamliner [DOOR15]

Since Virkler *et al.* published the results of his tests for fatigue crack growth under homogeneous cyclic loading on 2024-T3 aluminum alloy [VIRK78], this material has become a common ground for the research of diverse statistical analysis to fatigue crack growth. For this reason we decided to employ this specific aluminum alloy as study subject and test our approach on it.

The cracks that appear inside of these aluminum components can grow when exposed to a cyclic loading. The Paris' Law has traditionally been used to describe this crack growth with the stress-intensity range (ΔK) and the corresponding material constants, C and m, which are usually estimated through empirical observations.

$$\frac{da}{dN} = C(\Delta K)^m$$

Where a represents the crack size and N the number of fatigue loading cycles. The stress-intensity range is defined as the difference between the maximum and the minimum stress intensity factors ($\Delta K = K_{max} - K_{\min}$). It can also be calculated as:

$$\Delta K = \Delta\sigma\sqrt{\pi a}$$

Where $\Delta\sigma$ is the difference between the maximum and the minimum stress applied ($\Delta\sigma = \sigma_{max} - \sigma_{\min}$). This approach, as well as other similar analytical approaches (Forman, Priddle, Elber, etc.), has been successful when determining the fatigue crack growth for specific simple configurations. It can become complex when facing more complicated loadings, geometries and configurations. Another way of facing the problem is using FE modelling and analysis. This allows a much wider range of possibilities: different placement of the crack, different modes of fatigue load, various specimen geometries and physical properties, etc. It also provides information about the stress and strain states, as well as the crack length by using the technique of XFEM (Extended Finite Element Method). The issue with FEA is the computational cost it has, due to having to make calculations on each point of the created mesh. Finally, another way of facing the problem is through surrogate models. Surrogate models are a way to capture the essential features of the previous models, in a reliable way, but being computationally much more efficient. This data-driven models use the provided data to reveal the system's behavior. This is the approach taken for the research.

These explained approaches can be used to estimate how a crack will grow, which gives a good understanding of the mechanisms behind the fracture of the components. Nevertheless, the biggest question with the cracks is when will the specimen fail. The failure of the piece can be physically defined through the critical stress intensity factor (K_{Ic}). If the stress intensity at the piece surpasses K_{Ic}, the crack will start to propagate rapidly. Additionally, for homogeneous cyclic loading, the σ_{max} is constant through the whole cycling process and a critical crack length a_{Ic} can be obtained ($K_{Ic} = \sigma_{max}\sqrt{\pi a_{Ic}}$). Once we have a criteria for determining failure, estimating the time until this event (failure) will be the goal. This is the question that will be answered through the research, not focusing on how the crack will grow, but how longer can the component operate. The problem is encapsulated in the field of remaining useful life estimation, (RUL) which is the remaining time until the specimen fails. In our case, we define the failure of a component with the critical crack length, as mentioned above. A probabilistic approach will be taken, so that an estimation of the RUL is made with a confidence interval. A certain distribution (Gaussian distribution) will be assumed for the RUL to build this confidence interval.

Another factor considered in the research was the feasibility of the measurements. Measuring the crack length directly implies significant efforts. Alternatively, strategically placed sensors (strain gauges) can provide information about the strain state of the specimen, which is linked to the crack length. The measurements of these sensors can be used as condition indicators to model the response of the specimen to the fatigue loading and its Remaining Useful Life.

Putting it all together, the goal is to estimate how long an Aluminum specimen (Al 2024-T3) has until failure (Remaining Useful Life) based on strain measurements on the piece. The estimations are made through the so called Surrogate Models, that are data-driven and unveil more about the actual behavior the more data they are fed. Although the measurements of the strain are performed on a cyclic-basis, we will approach the measurements as a time series, which means that we take the measurements as successive equally spaced points in time. The Surrogate Models are therefore based on time-series analysis algorithms brought from the area of statistical analysis and machine learning. The two algorithms that have been employed are shortly described:

Support Vector Regression:

Support Vector Machines is a widely spread supervised learning model used mainly for classification. In classification, as well as in regression, SVM takes as reference the closest points (Support Vectors, shown in Figure 2) in order to build a separating hyperplane. This algorithm is based on Structural Risk Minimization principles, which results in reliable models with a good generalization potential, which can be a strong reason to use it for estimating RUL.

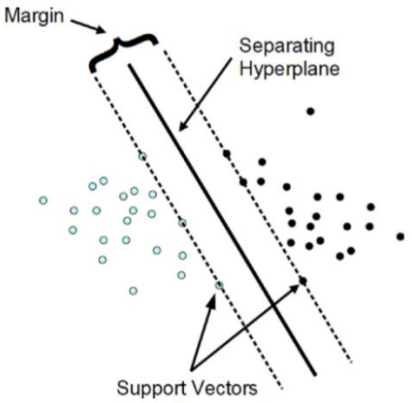


Figure 2: Support Vector Machine for classification [IVAN14]

Support Vector Regression is SVM's variant for regression problems. It doesn't inherently deal with time series, but we will preprocess the measurements to feed the algorithm and make estimations.

Recurrent Neural Networks:

This supervised leaning method is a type of Artificial Neural Networks that allows analyzing time-dependent data. Regular Neural Network architectures take a fixed number of inputs and make predictions based on them. In time series, instead of using regular Neural Networks and make a prediction for each point, Recurrent Neural Networks' inputs are all the points in the time series. Their utility relies on the ability to capture underlying features based on the sequences of the time series. These features are gathered in the so called "hidden state" vector, which is represented in the left image of Figure 3 as a closed-loop arrow:

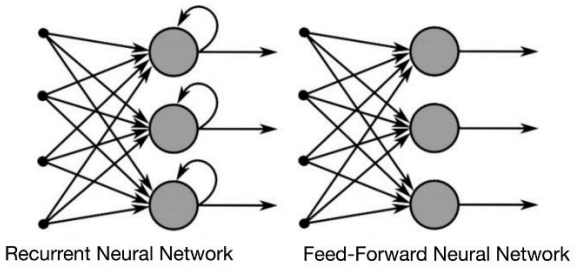


Figure 3: Recurrent Neural Network vs regular Neural Network [XPER18]

Both algorithms, with their corresponding variants, will be discussed in detailed in the “3.3. Algorithms” section.

The goal of these models is to predict the RUL based on strain measurements from real tests. The available data to make the estimations comes from Virkler’s dataset and there are a couple of issues with this approach:

- Data size: Virkler’s dataset consists on only 68 series of crack growth. Not sufficient to have reliable models (mostly the RNN)
- Crack measurement: Virkler measured the crack sizes, and we aim at performing our estimations using strain measurements

As a solution, Finite Element Methods (XFEM more specifically) are employed to create a training set. The models built through this training will be tested with the real values from Virkler’s dataset. If proven successful, this approach would suggest that estimations with a confidence interval for real tests can be made with Finite Element simulations through the use of surrogate models.

2. Description of the technologies (State of the Art)

In this section, the current state of the Remaining Useful Life Estimation through probabilistic approaches will be outlined. RUL applied to other fields apart from crack growth is also relevant for the subject, so applications in various fields and their results will also be mentioned.

The current state of the Extended Finite Element Method and how it is employed to predict crack growth will also be explained. XFEM is being employed to build the training set for the model by performing changing crack simulations and outputting the strains at various points of the piece. Therefore it is important to understand it, along with its use and potential complications.

2.1. Probabilistic approach to RUL through Surrogate Models

Remaining Useful Life (RUL) refers to how long the machine (battery, piece, specimen, etc.) is likely to operate before failure occurs (usually the time where repair or replacement will be needed). RUL is especially relevant for Predictive Maintenance, considering that RUL essentially points out when maintenance should be performed. This provides the means to scheduling maintenance, optimizing operating efficiency and avoiding unplanned breakdowns with their corresponding time delays. It also is essential that the RUL model provides a confidence interval with a given likelihood. These models to estimate Remaining Useful Life can be classified according to two criteria: what type of data is available [BARU18] and how much weight does previous knowledge about the physical phenomenon have on the modelling [KHEL16].

2.1.1. Type of data available

The best way to understand a machine, component or other specimen's behavior and estimate future events is to have real measurements, providing as much information about them as possible. It is beneficial having very frequent measurements through the whole operation of the specimen, but also having many variables that can affect the state of the specimen and its useful life. Independently from the number of variables, the RUL model employed will depend on the type of data available for the tests. They can be classified, from less to more information, as:

Lifetime Data: when only the times of failure are known, probability distributions are used to estimate the RUL of the specimen. The models that use this type of data are called "Survival Models".

Threshold Data: sometimes, there is an available established threshold that shouldn't be crossed, as the health of the specimen rapidly deteriorates after it gets to that point. A time series model can be fitted in order to predict how long it will take the current specimen to reach that threshold, with a confidence interval. The models that employ this kind of data are the "Degradation Models".

Run-to-Failure Data: finally, you can also have measurements of the whole life of the specimens, from the original completely healthy state until failure occurs. In this case, the degradation profile of the "n" closest specimens are employed to estimate the RUL of

our studied specimen. This models, which search for similar degradation profiles, are called "Similarity Models".

This different types of information used for RUL estimation are illustrated on Figure 4 through the example of the health of an aircraft engine:

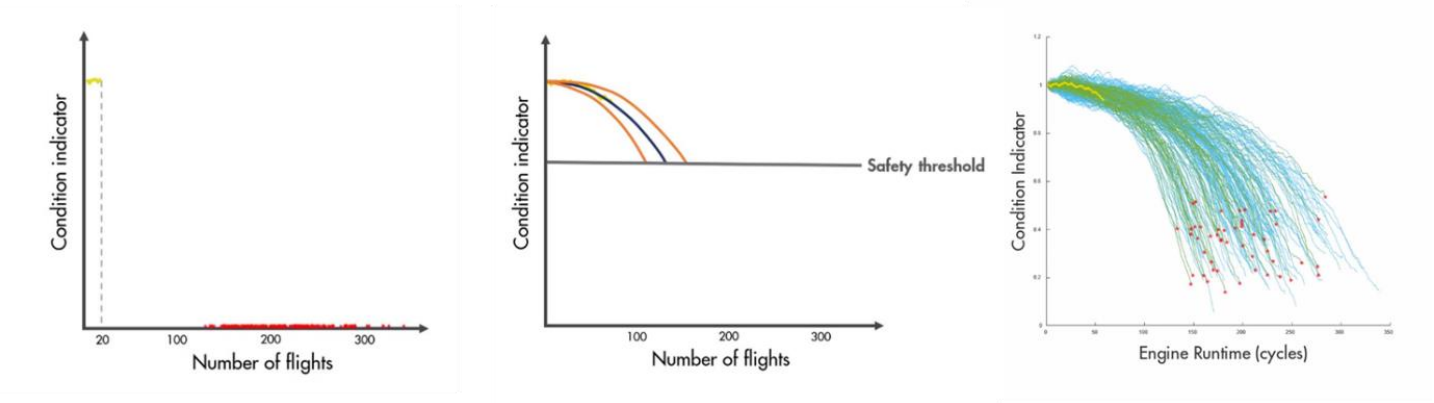


Figure 4: Types of data available for RUL estimation. From left to right: Lifetime Data, Threshold Data and Run-to-Failure Data [BARU18]

In the case of crack growth, there is usually a critical stress intensity factor, and for homogeneous loading a crack length (as explained in the "1. Introduction and project approach" section), established that marks a threshold. After this theshold the crack grows very rapidly. In Virkler's dataset the measurements are taken until this threshold is reached (half crack length = 50 mm) and therefore we will also generate our data with XFEM until this same threshold. Hence, the type of data studied belongs to the "Threshold Data" class and thus requires a degradation model to estimate RUL. The condition indicator can be a combination of relevant features when plenty of sensors and their measurements are available. In the studied case the only feature available is the strain at a startegically placed sensor, so this strain will be used as the condition indicator of the health of the specimen.

2.1.2. Weight of physical knowledge

Physical phenomena have traditionally been studied through the search for the underlying laws that govern them. This requires extensive and deep knowledge on the topic, which isn't always easily obtainable. The other approach is using data of different features that affect the phenomenon to find the underlying behavior behind it. This leads to the categorization of the possible models in: model-based, data-driven and hybrid.

Model-based

These approaches use physical models to describe the degradation state of the machine. Let's take the example of fatigue crack growth. To understand the physical models better, a short summary of the relevant aspects of fracture mechanics will be presented. Fracture mechanics is the field of solid mechanics responsible of studying the propagation of cracks. Questions like "Will a crack grow under a given load?", "When a crack grows, what is its speed and direction?" or "Will the crack growth stop?" are answered through the studies in this field. These cracks can appear from directly applied loads, as

well as from secondary self-equilibrating stress fields, like residual stresses. Fracture mechanics will try to characterize the local deformation around the crack tip based on the loads applied and the geometry of the specimen.

There have been many theories trying to illustrate this fracture development, including Linear Elastic Fracture Mechanics (LEFM), Elasto-Plastic Fracture Mechanics (EPFM) and other Cohesive zone models. There are three ways that a force can be applied to enable a crack to propagate. LEFM differentiates them as three different fracture modes:

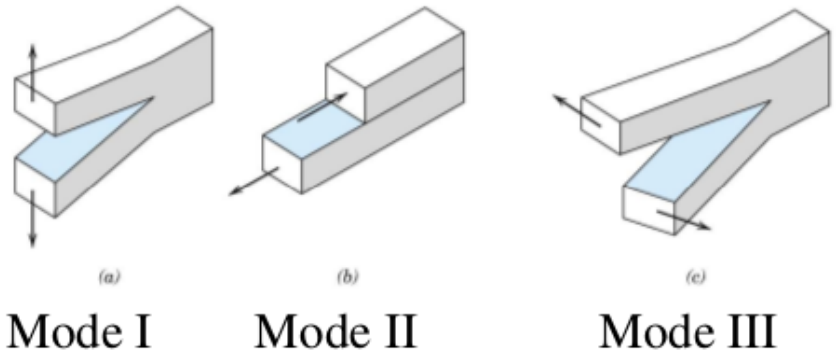


Figure 5: The three fracture modes [TURK16]

Mode I – Opening Mode: the tensile loads are applied perpendicularly to the crack, opening the crack. This is the most common fracture mode, and it is used for fracture toughness tests like the one that is studied on this research. The critical value of stress intensity associated with this mode is defined as K_{Ic}.

Mode II – In-Plane Shear Mode: shear stress is applied in the crack's in-plane direction, normal to its leading edge.

Mode III - Tearing Mode: shear stress is applied out of plane, parallel to the leading edge of the crack.

The test we are basing our analysis on studies Mode I – Opening fracture model. With the tensile load and the crack lengths, stress intensities are defined. The rate at which the crack grows can be described with the stress intensity range (ΔK), as shown in Figure 6.

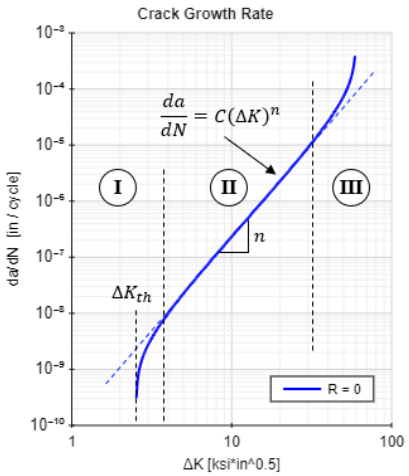


Figure 6: Fatigue crack growth rate [MECH]

In this log-log scaled representation of the crack growth, there are three clearly defined regions. Region I corresponds to low stress intensity ranges and is characterized by slow crack growth rates. Most o the fatigue life of a specimen is spent here. Region II is where Paris' law and its variations are thought to predict the crack growth rates. Region III, with high stress intensity ranges, corresponds to the end of the useful life of the specimen and is characterized by the rapid crack growth to fracture. Temporally, it represents a very small portion of the crack growth. We can observe through this difference between regions that Paris' law only applies for a specific part, and it might oversimplify the crack growth during the whole lifetime. Paris' law, also known as Paris-Erdogan law, is shown below:

$$\frac{da}{dN} = C(\Delta K)^m$$

Where $\frac{da}{dN}$ is the crack growth per cycle at that point, C and m are material constants and ΔK is the stress intensity range. Due to it not describing the crack growth faithfully during the whole process, other models were later developed. These models also took into account the load ratio ($R = \frac{K_{min}}{K_{max}}$), the effects of the maximum stress intensity factor K_{max} amongst other factors.

For example, Elber *et al.* tried to explain crack growth through crack closure concepts [ELBE70]. He figured out that crack closure is present in cyclic loading, even for loads that are greater than zero, and that it decreases the fatigue crack growth rate through the effective stress intensity range.

$$\Delta K = K_{max} - K_{min}$$

$$K_{min} = \max[K_{min}, 0]$$

Crack closure at $K = K_{op}$ gives:

$$\Delta K_{eff} = K_{max} - K_{op}$$

And the result of employing the effective stress intensity range is:

$$\frac{da}{dN} = C\left(\Delta K_{eff}\right)^m$$

K_{op} can be empirically calculated as:

$$K_{op} = \varphi(R) K_{max}$$

$$\varphi(R) = 0.25 + 0.5 * R + 0.25 * R^2\,, \qquad -1 \le R \le 1$$

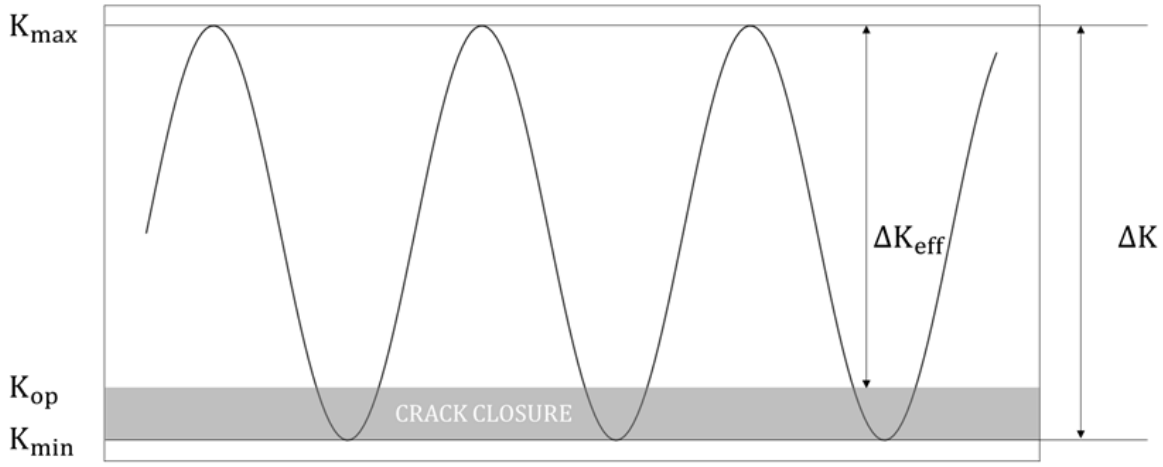


Figure 7: Crack closure form Elber's theory

In a practical way, Elber's theory just applies a higher K_{min}. This results in more conservative predictions than the Paris' law. This law also allows to add the model a dependence on the load ratio R.

There are many other theories that try to predict the phenomena of fatigue crack growth, and all of them have in common that they require a deep mechanical understanding and therefore have a low applicability.

Data-driven

Data-driven approaches making estimations from collected data based on statistical and machine learning methods, looking to unveil the system's behavior. In order to achieve a high precision, these models can become complex. The applicability of these models must therefore always be considered when evaluating the precision with certain complexity. Some of this data-driven approaches are the conventional auto-regressive models, but also Artificial Neural Networks, Support Vector Machines, Kriging, Bayesian Networks and stochastic processes.

One of the most relevant models to our studied case was recently published by Khelif et al. on 2016 [KHEL16]. It implements a Support Vector Regression model to predict the Remaining Useful Life of Turbofan engine. The dataset utilized is available at the NASA prognostics data repository. This dataset contains multiple multivariate time series measurements generated by simulations on the Commercial Modular Aero-Propulsion System Simulation (CMAPSS). The most relevant features (sensor measurements) for the model are selected through a wrapper selection technique. This step is of great importance, because redundant or irrelevant features can affect the speed and accuracy of the algorithms. With the most important sensor measurements, the time series are preprocessed before feeding them to the algorithm. In this case, the trend features of the time series were extracted by dividing the sequence in windows and extracting the mean and slope of each sequence. There were two different procedures to create the training and the test set.

To create the training set, each time series T_i was divided in concatenated windows of size L. For example, the k^{th} window of the time series T_i encapsulates the points from $t^i_{(k-1)L+1}$ to t^i_{kL}. From each of this windows, the average value a and the trend coefficient or slope s are extracted, obtaining a feature vector of side $2 * d$ from each window, where d is the number of dimensions of the time series. The RUL of each window is calculated from the end of the window. This is, for the k^{th} window from the time series T_i the RUL would be $l(T_i) - kL$.

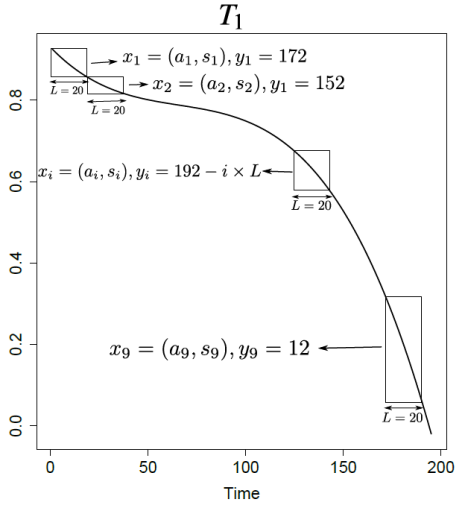


Figure 8: Windows associated mean, slope and RUL [KHEL16]

The procedure taken to create the test set is a little bit different from the training set. The test set aims at replicating real situations where measurements are taken on-line and can start at any given health condition of the specimen. The time series U has a last monitoring measurement at $l(U)$ and the RUL is the difference between the time of failure and $l(U)$. Here the time series U is divided in windows of size L. The windows are sliding windows this time, so each time a new measurement is taken, the window of length L is displaced one unit of time. This way, the average value a and the trend coefficient or slope s are extracted from the current window and inputted in the SVR model to predict RUL at that instant $l(U)$.

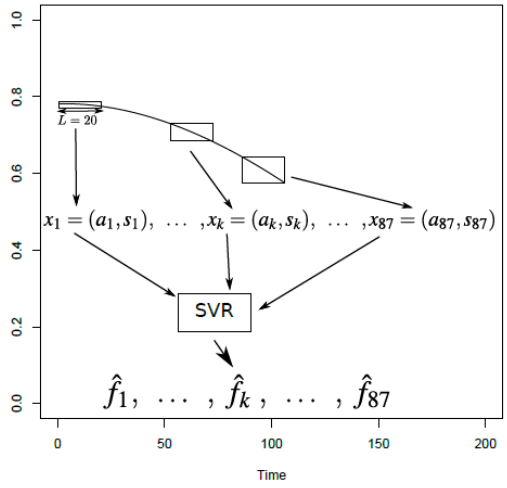


Figure 9: RUL estimation through sliding window [KHEL16]

The Support Vector Regression model is treated as a black box for the moment. It will be described in detail in the "3.3. Algorithms" section.

Hybrid

The last approach to solving the problem of RUL estimation are the hybrid approaches. They use a model obtained from the knowledge of the phenomenon, and update the model's parameters based on the observed information. This approach has been used to estimate crack growth, as well as battery life [AN__13], by using their respective physical models and updating the parameters with a particle filter.

One of the most recent and relevant studies on the topic of fatigue crack growth prognosis to employ a hybrid model was published by Wang *et al.* on 2017 [WANG17]. In this paper, the fatigue crack growth from Virkler's dataset of 2024-T3 aluminum alloy (the same dataset that is employed in this research) is estimated through a Bayesian probabilistic model, more specifically, applying an Extended Kalman Filter. Taking Paris' law as foundation, it derives the corresponding parameters from half of Virkler's crack growth data. The other half is used to test the model.

Kalman Filters are applied for making estimations when the variable of interest isn't directly accessible. In this case, the variable of interest is the crack length, but a random walk is applied on Virkler's data to generate artificial set of measurements for these tests. The most relevant equations from Kalman filters are the state-space equations:

$$x_k = f(x_{k-1}) + w_k$$

$$y_k = h(x_k) + v_k$$

Where x_k is the state variable at k^{th} time instant, which is a function of the previous value of the state variable (x_{k-1}) plus w_k, an uncorrelated random variable. y_k is the actual measurement, described as function of the state variable (x_k) plus v_k, another uncorrelated random variable. In this study, the state variable is the crack length and the measurements are created through a simple random walk model:

$$y_k = a_k + v_k$$

Where a_k is the crack length at k^{th} time instant. Explaining the Bayesian model with the Extended Kalman Filter and how it works is out of the scope of this section. If there is an interest on the subject, the model is clearly illustrated in the publication from Wang et al [WANG17].

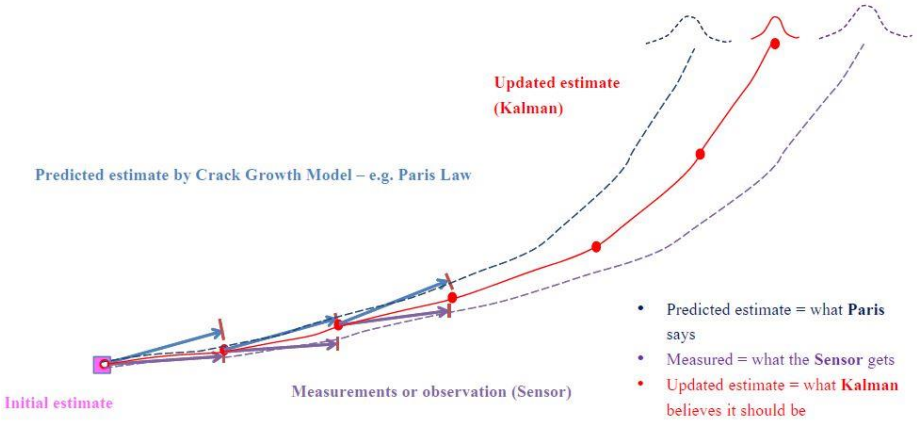


Figure 10: Kalman Filter estimations [WANG17]

The results of this approach were good accuracy of the predictions and a prove of the method's robustness when managing uncertainties in Paris' law's material constants C and m, as well as uncertainties in the measurements. The optimal material constants' values were $m = 2.9$ and C Gaussian distributed with a mean of $C_{med} = 8.586 * 10^{-11}$ and a standard deviation of $C_{std} = 0.619 * 10^{-11}$ (with ΔK in $MPa\sqrt{m}$).

This mixture of model-based and data-driven approaches can deliver reliable results like we have seen in the previous example. The drawback of this hybrid approaches is that it still needs a deep and specific physical understanding and is computationally expensive.

2.2. Extended Finite Element Method

Extended Finite Element Method (XFEM) enhances the piecewise polynomial function space of conventional finite element methods with extra functions, also called "enrichment functions". It was originally developed by Belytschko and Black in 1999 [BELY99], based on the partition unity method of Babuska and Melenk [MELE96]. It can be applied to various problems such as fracture, void growth or phase change. It finds its place where traditional Finite Element Methods fail of are computationally too costly.

The traditional fracture modeling methods only allow crack propagation through predefined element boundaries, while XFEM also allows the crack to be positioned inside an element. This way, the position of the crack becomes independent of the mesh, which can be a very powerful property to exploit.

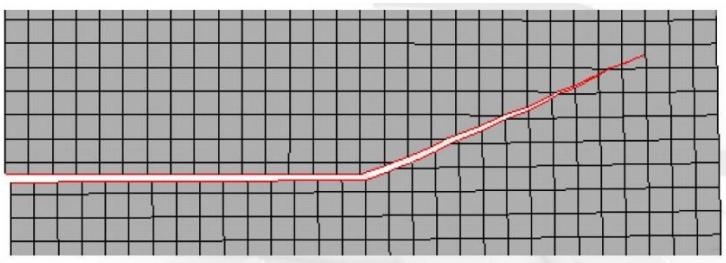


Figure 11: Crack crossing finite elements [DASS09]

XFEM can also be used conjointly with other techniques such as the Cohesive Zone model or the Virtual Crack Closure Technique. It also allows calculating the load carrying capacity of a specimen with a crack. Additionally, this method has the following advantages:

- Simplicity of the initial crack characterization, due to the mesh being created independently form the crack and the lack of need of a partitioned geometry
- Nonlinear material and geometric analysis
- The crack path doesn't have to be defined *a priori*, it depends on the solution
- Remeshing efforts are significantly lower
- The convergence rate for the FE solution is boosted, thanks to the singular crack tip enrichment

Concerning the creation of mesh-independent models with cracks, there are four main requirements, and each one is met through the use of different techniques.

a) Integrating the crack as a discontinuous geometry and the discontinuous solution field in the FE basis functions.
   Solution approach: Extended Finite Element Method (XFEM)
b) Determining the magnitude of the displacement difference (discontinuity) between the crack faces
   Solution approach: Cohesive Zone Model (CZM)
c) Locating the discontinuity
   Solution approach: Level Set Method (LSM)
d) Initiation of the crack and propagation criteria

The previously mentioned enrichment functions are selected for a variety of problems where the a priori knowledge of partial differential equations is integrated in the FE space. The two enrichment functions implemented in fracture modelling are:

- Heaviside function: to illustrate the displacement difference between crack faces
- Crack tip asymptotic function: to model singularity

$$u^h(x) = \sum_{I \in N} N_I(x)[u_I + H(x)a_I + \sum_{\alpha=1}^{4} F_\alpha(x) b_I^\alpha]$$

Where u_I is the nodal DOF of traditional shape functions. In the XFEM Displacement Interpolation equation (shown above) there are two terms to be distinguished:

Heaviside enrichment term: $H(x)a_I \ (\in N_\Gamma)$

- $H(x)$: Heaviside distribution
- a_I: nodal enriched DOF (jump discontinuity)
- N_Γ: nodes belonging to elements cut by the crack

Crack tip enrichment term: $\sum_{\alpha=1}^{4} F_\alpha(x) b_I^\alpha \ (\in N_\Lambda)$

- $F_\alpha(x)$: crack tip asymptotic functions
- b_I^α: nodal DOF (crack tip enrichment)
- N_Λ: nodes belonging to elements containing crack tip

The Heaviside function can be found below. It takes into account the displacement difference over the crack.

$$H(x) = \begin{cases} 1 \ if \ (x - x^*) * n \geq 0 \\ -1 \ otherwise \end{cases}$$

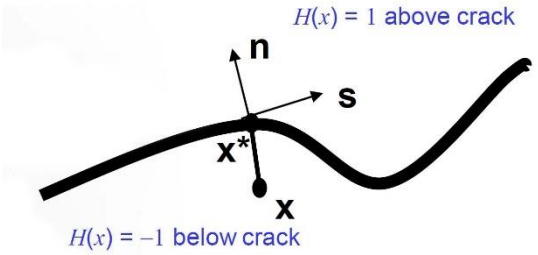


Figure 12: Heaviside function [DASS09]

This enrichment function is multiplied by the traditional shape functions, resulting in a local enrichment around the crack and preservation of the resulting matrix equations. The crack is placed through the Level Set Method, which will be reviewed in a couple of paragraphs.

The Crack Tip Enrichment Functions can only be employed for stationary cracks. They clarify the crack tip singularity and are based on displacement field basis functions for sharp crack in an isotropic linear elastic material:

$$[F_\alpha(x), \alpha = 1 - 4] = [\sqrt{r}\sin\left(\frac{\theta}{2}\right), \sqrt{r}\cos\left(\frac{\theta}{2}\right), \sqrt{r}\sin\theta\sin\left(\frac{\theta}{2}\right), \sqrt{r}\sin\theta\cos\left(\frac{\theta}{2}\right)]$$

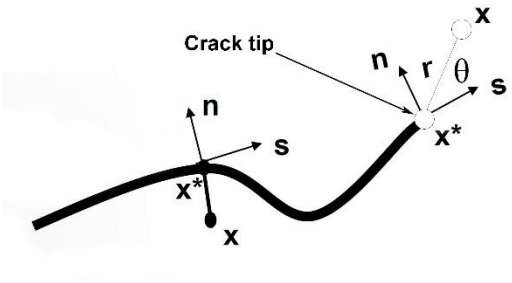


Figure 13: Crack Tip Enrichment Functions [DASS09], modified

Belytschko and his coworkers introduced in 2006 the Phantom Node Approach for the Crack Propagation Implementation [SONG06]. This way, XFEM can be integrated in the traditional FEM framework. The discontinuous element with Heaviside enrichment is considered as a superposition of two continuous elements with phantom nodes. This approach does not include the crack tip enrichment functions.

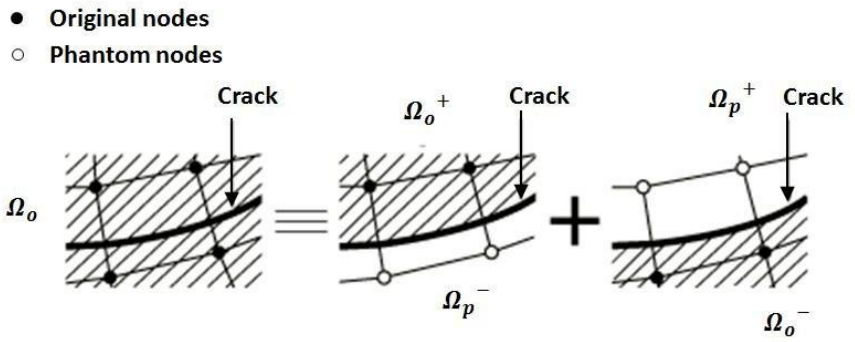
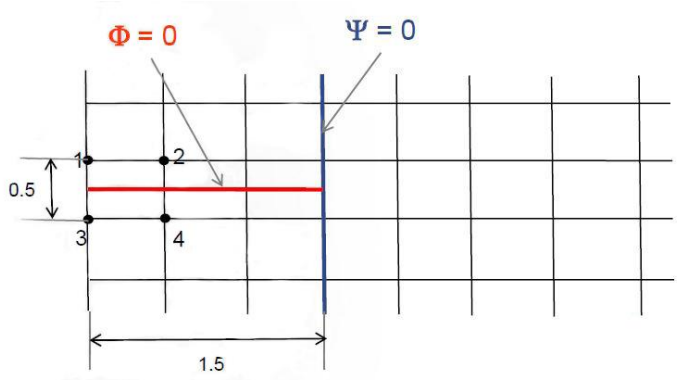


Figure 14: Phantom Node Approach [MOHA14]

As mentioned before, the crack is placed through the Level Set Method. A level set, level surface or isosurface of a function is the set of all points where the function achieves certain value. It is a popular technique to represent surfaces in interface tracking problems. The functions employed to characterize the crack are Φ and Ψ.

- Level set $\Phi = 0$ represents the crack face
- The intersection of $\Phi = 0$ and $\Psi = 0$ represents the crack front
- The value of the function Φ is the distance (with sign) of the node from the crack face
- The value of the function Ψ is the distance (with sign) of the node from an orthogonal surface that passes through the crack front.



Node	Φ	Ψ
1	+0.25	-1.5
2	+0.25	-1.0
3	-0.25	-1.5
4	-0.25	-1.0

Figure 15: Level Set Method [DASS09], modified

Finally, the basic concepts applied for damage modelling will be presented. The damage is modelled by a traction-separation law across the fracture surface and three different stages are differentiated:

- Damage initiation
- Damage evolution
- Traction-free crack faces at failure

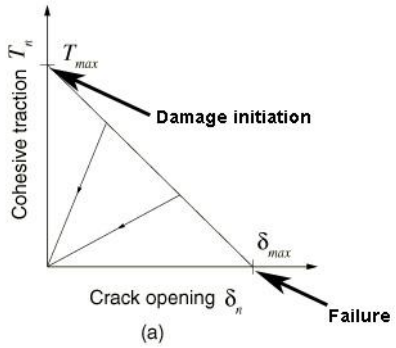


Figure 16: Damage stages [DASS09], modified

Damage Initiation: two criteria can be applied:

- Maximum Principal Stress criterion: damage initiation when maximum principal stress reaches the critical value
- Maximum Principal Strain criterion: damage initiation when maximum principal strain reaches the critical value

The crack plane will be perpendicular to the maximum principal stress of strain and it will start at the middle of the element, although it can propagate in any direction through the mesh.

Damage Evolution: there are various linear and non-linear damage evolution models that can be applied for this stage. It is not needed to define the undamaged traction-separation response.

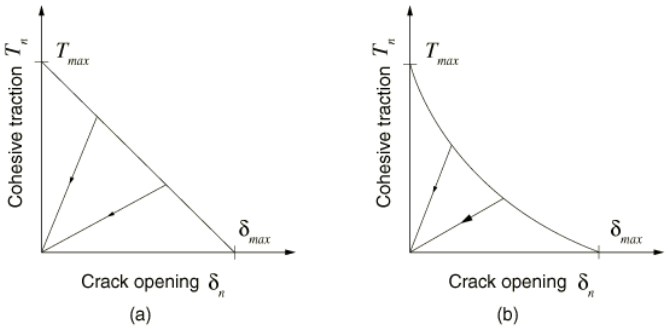


Figure 17: Typical linear (a) and nonlinear (b) traction-separation response [DASS09]

Damage Stabilization: at this stage, the crack makes the structural response nonlinear and non-smooth, complicating the convergence of numerical methods. To solve this problem, viscous regularization is utilized with the Newton method.

[DASS09]

3. Description of the developed models

After overviewing the different probabilistic approaches taken to estimate Remaining Useful Life and their classification, now the chosen models (there are two: SVR and RNN) and the reasons for selecting them are explained in detail. The application of XFEM to our case is also described, with the technical justifications behind the design of the geometry, crack, mesh, etc.

Let's recap the question to be answered before exploring the model selection. The analyzed problem is fatigue crack growth, where the goal is to find the remaining useful life (RUL) at a given point in the crack growth process. A threshold was established, where if the crack length (half crack length) surpasses 50 mm, the specimen will be deemed to have failed.

The information used as condition indicator of the health of the structure is the strain at strategically placed point of the piece (placement explained in the "3.2. Data" section). This will be the input of the model, and the output will be the RUL. An overview of the process is illustrated on Figure 18, with the surrogate model as a black box for simplification.

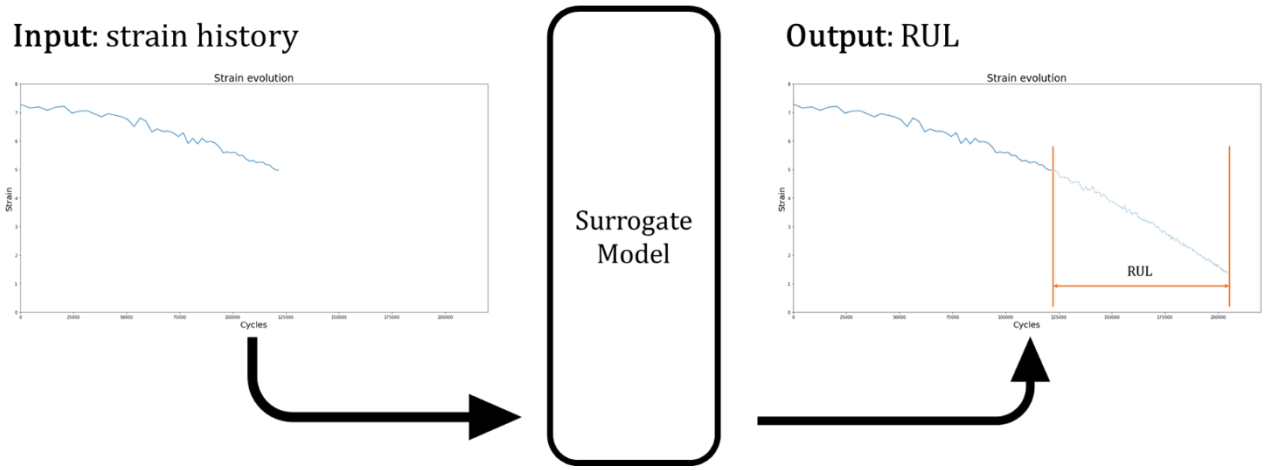


Figure 18: Remaining Useful Life Surrogate Model functional overview

According to the model classification by type of data presented on the section "2.1.1. Type of data available", the model to be employed should be a Degradation Model. The data had a given threshold and a degradation model with a time series approach should be chosen. As for the model classification on "2.1.2. Weight of physical knowledge", a data-driven approach will be taken. The physical models are already well analyzed and they will be used as a benchmark to evaluate the obtained models. Virkler's data has already been approximated with the Paris' law by tuning the parameters C and m for this specific case. It gives a very close approximation, and the models' performance will be compared against it.

As mentioned, a time series approach must be taken for the modelling. A time series is a set of observations, each recorded at a specific time. This observations can be discrete, where the observation times are a discrete set, or continuous, where observations are recorded continuous over a certain time interval. In the case of the research, the

observation are considered continuous. The observation times aren't made at fixed time intervals, but at fixed crack length intervals, which the model should take into consideration.

Usually, time series analysis aims at drawing inferences from the time series in order to find the underlying distribution. Through this approaches, more specifically through the linear model ARIMA, the time series of crack growth has been accurately described [SOLO91]. By way of the inference, the RUL at a given point can be estimated, but a much more direct approach will be taken, not trying to estimate the whole crack growth process but instead only the RUL.

For this purpose, and due to the fact that the input time series have a variable length, the chosen models (RNN and SVR) follow two different approaches:

- **RNN**: the input are sequences of variable length, and through the hidden state vector as a time series embedding (will be explained in the "3.3. Algorithms" section) the behavior of the sequence is captured.
- **SVR**: the input is a window of fixed length. The values of the mean and the trend coefficient are extracted. This way, a regression through sliding window approach is taken.

In the second approach, many other regressors are available that could perform the same task, but the Support Vector Machines tend to deliver very reliable performances with good generalization potential, which is the reason why it was chosen. Other regression models through sliding window will be continued to be explored after the finalization of the thesis.

The additional challenge that both models have to face is not just making an estimation about the RUL, but also provide a confidence interval for it. This requires two different extensions for each model:

- **RNN**: the output nodes of the network are two, one predicting the mean and another predicting the standard deviation, each one with the corresponding activation function. This is called a Mixed Density Network architecture "3.3. Algorithms").
- **SVR**: through another regression model, the error of the initial prediction will be estimated. This receives the name of Distribution Estimator ("3.3. Algorithms").

3.1. Goals and specification

The research had the structure of a data analytics project, where certain goals can be set for each stage of the analysis. The final goal was to obtain a reliable model that can estimate the Remaining Useful Life of the specimen, so all the goals headed towards this objective.

1. Generate realistic and useful data

The models weren't trained on real data, given that the crack length isn't easily measured directly, and instead the strain measurements (would be measured through strain gauges) were used. For this reason, the data was generated from both analytical models and finite element methods. This data had to be as close to values measured in

reality as possible, meaning that it should not only have the tendencies associated to the real crack length, but also features like the noise of the measurements on the strain gauges.

2. Develop algorithms that estimate the Remaining Useful Life of the specimens accurately

These algorithms used as surrogate models should be able to capture the most important features of the sequence of fatigue crack growth in order to make an accurate estimation for the remaining useful life that the studied specimen has. The algorithms that were explored have been proven to deliver sufficient results in different studies on the field of remaining useful life for similar purposes as ours, which leads to think that they should also perform well on our studied case.

3. Find the limitations and potential downfalls of the developed algorithms

Given that the models explored are machine learning algorithms, one must be careful when evaluating its performance. The potential downfalls of each algorithm had to be identified in order to make a reasonable use of their estimations. These models can also suffer from a lack of interpretability, which is a motive strong enough to be careful when drawing conclusions from the results.

4. Compare algorithms, taking into account their generalization potential

At the end of the research, the obtained models were compared and conclusions about the suitability of them were made. Apart from the accuracy and computational costs, other aspects had to be considered. For example, by utilizing the data from a specific test's conditions (Virkler's), there is the potential danger of overfitting the model to the conditions that were created. This kind of model would be of no use in a real application, so this aspect (amongst many others) was taken into account when assessing the suitability.

3.2. Data

This analysis of fatigue crack growth uses Virkler's dataset as a platform to test the models, so the tests from Virkler's work [VIRK78] is first described in detail. Then, the approximation by Wang *et al.* [WANG17] through the update of the parameters of Paris' law is presented. Finally, the implementation of XFEM to obtain the strain and the assumptions for the final dataset is exposed.

3.2.1. Virkler's dataset

The tests performed by Virkler, Hillberry and Goel from the School of Mechanical Engineering have been a reference for statistical analysis of fatigue crack growth until today. They are presented on the publication "The Statistical Nature of Fatigue Crack Propagation" (1978) [VIRK78].

These tests were performed in a center cracked panel with homogeneous cyclic loading, favoring it over a constant stress intensity range (ΔK) due to the easier control and replicability, as well as for giving a wide range of ΔK to analyze. The loading was chosen with an amplitude of $\Delta P = 4200\ lbs\ (= 18682.5\ N)$ through $P_{min} = 1050\ lbs\ (=$

4670,6 N), $P_{max} = 5250\ lbs$ ($= 5250\ N$) and a stress intensity ratio $R = 0.2$ to stay far from the compression region.

To obtain a wide enough range of crack growth rates, it was estimated through theoretical investigation that the crack should grow until, at least, 40.0 mm. Therefore, the threshold of 50.0 mm was established. Furthermore, the steady state condition wouldn't be reached until the crack was 9.0 mm due to the load shedding process. This 9.0 mm were selected as the initial crack size for the test specimens. The measurements were not made with a fixed time interval, but instead with a fixed crack growth interval Δa, and it was chosen to be $\Delta a = 0.2\ mm$ to reduce the data error as much as possible. The end of the series had the problem of the crack growth being too fast for the equipment employed (optical system and printer) and thus the crack growth interval was increased to $\Delta a = 0.4\ mm$ and $\Delta a = 0.8\ mm$.

With this requirements, the sixty eight specimens of 2024-T3 aluminum alloy were manufactured. The thickness of the specimen was 0.100 $inch$ ($= 2.54\ mm$), with a crack of 9.0 mm in the middle. The geometry of the specimen is shown in the original sketch on Figure 19, which defines the dimensions in inches. The unit employed on the rest of the research for the crack length was mm. On the research, the piece's geometry is also described in mm.

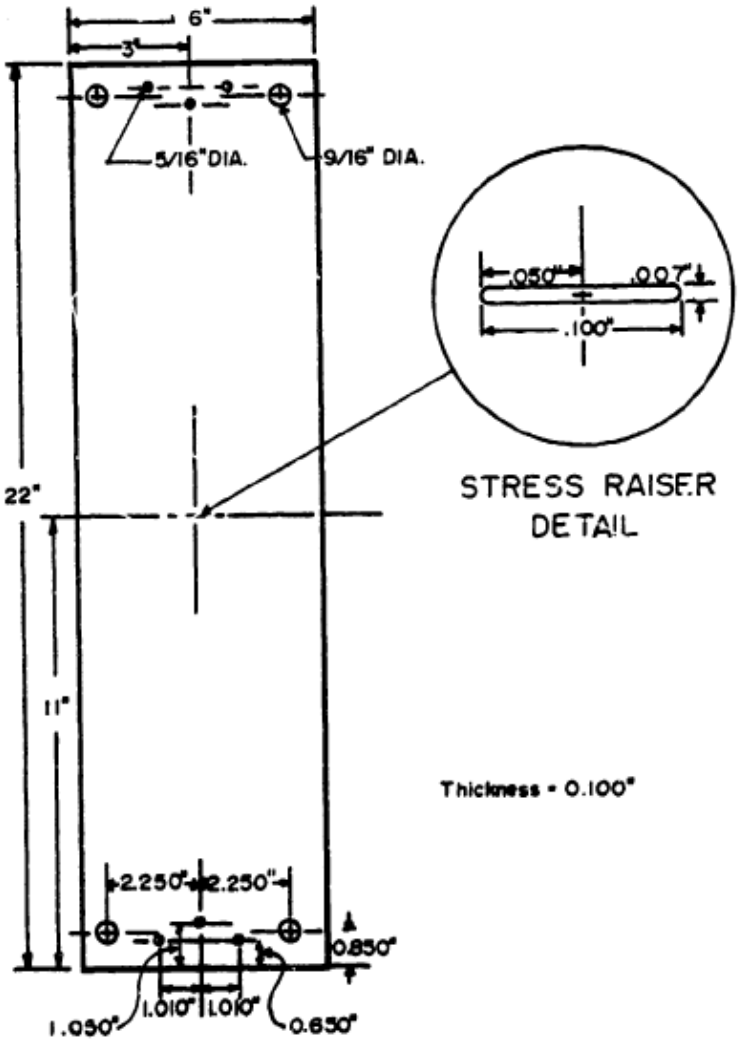


Figure 19: Virkler's Test Specimen (dimensions in inches) [VIRK78]

As for the equipment for the tests, a electro-hydraulic closed-loop system operated in load control was employed. A sinusoidal voltage signal, with a DC set point voltage were used to create the wanted loading. This loads were controlled within 0.2% percent of the target load through the machine's measurement system. The crack size was monitored with a mounted stereo microscope. The tests were performed at very similar conditions of room temperature $24ºC$ and desiccated air.

Initially, through a load amplitude of $\Delta P = 15000\ lbs\ (= 66723.3\ N)$ the crack was initiated at the stress raiser (Figure 19). After some growth of the crack, the load was gradually decreased by 10% every around 0.5 mm. The frequency of the fatigue cycle was initially 10 Hz until 5.4 mm and 20 Hz afterwards. The defined test load amplitude was reached 1 mm before reaching the initial 9.0 mm crack length in order to avoid unwanted load effects. After this point (crack length $2a = 18mm$), the loading was kept homogeneous during the entirety of the test.

Additionally, the measurement accuracy was calculated through the following methodology. The experimental error originates from the cycle count and the crack length measurement. As mentioned before, the machine's applied load error was determined to be 0.2% thanks to a control loop with the feedback signal of the loading. As for the error in measurements, they arise mainly from the alignment of the microscope and the crack. The microscope has a crosshair to align it with the piece, more specifically with a marked reference line. If the microscope moves during the test, it will induce a measurement error due to the lack of this alignment. A measurement error can also appear if the mentioned crosshair of the microscope isn't aligned with the crack tip.

The study of the errors made during the alignment were therefore deemed to be crucial for the experiment, so the error made by the observer's eye was analyzed. After the crack was initiated, the subject observing the crack was asked to define the moment when it reached 9.0 mm, as well as ten other measurements afterwards. This procedure was repeated nine more times with different crack lengths. The results were:

- Mean error: 0.001414 mm
- Standard deviation of the errors: 0.001390 mm

It is also worth mentioning that the experimental error was determined to be smaller for bigger increments of crack length.

Table 1: Average experimental error [VIRK78], modified

$\Delta a\ [mm]$	Average error %
0.20	0.71
0.40	0.35
0.80	0.17

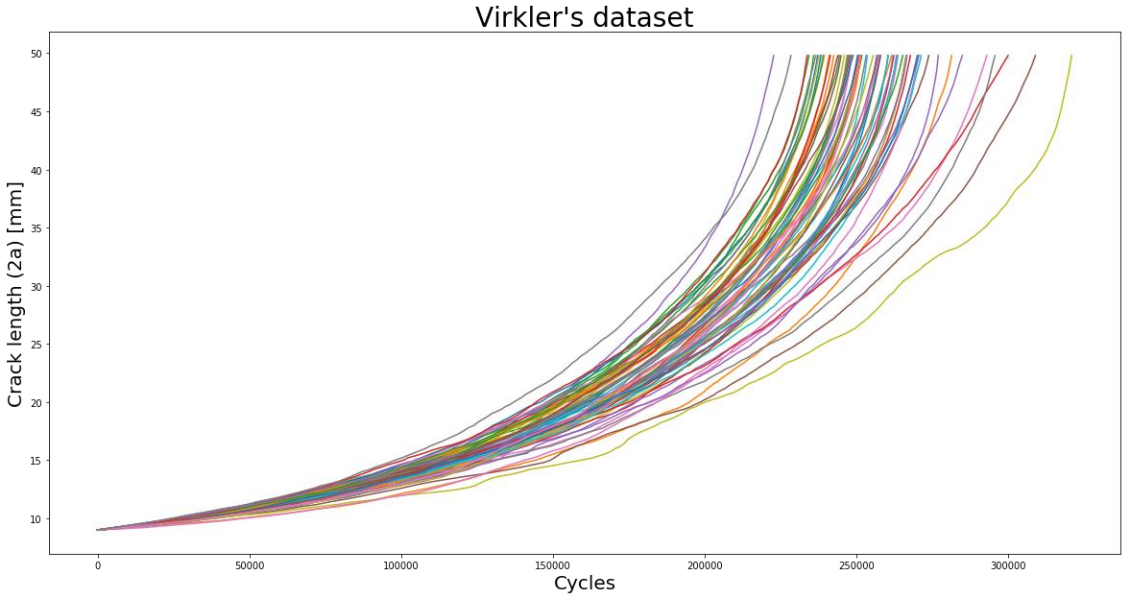


Figure 20: Crack growth sequences from Virkler's dataset

3.2.2. Paris Law Approximation

Many studies on the field of probabilistic approach to fatigue crack growth have taken place since this paper was published. Wang et al. [WANG17] used the 68 tests for a hybrid model ("2.1.2 Weight of physical knowledge") where the crack growth was estimated through a Bayesian probabilistic model, by applying an Extended Kalman Filter. For this model, the Paris' law was employed, and therefore the corresponding parameters C and m had to be defined. Therefore, a statistical study was performed on half of the tests (the other half would be used for predictions) to determine the most appropriate values of C and m.

These parameters are, for certain conditions like the load ratio and the maximum load under constant amplitude loading, material-specific. They are also known to be correlated, and thus the taken approach was treating the parameter m (exponent in Paris' law) as a constant and C as Gaussian distributed. Based on the considered Virkler tests variability, the distribution of C was estimated (median and standard deviation) using a fixed value for m. The optimal value for m was determined to be 2.9 based on the minimum total Norm-2 power of the Paris' law fitting errors. With this parameter fixed, the median and the standard deviation of the parameter C were calculated. These were determined to be $C_{med} = 8.586x10^{-11}$ and $C_{std} = 0.619x10^{-11}$ respectively (with ΔK in $MPa\sqrt{m}$).

It is also mentioned that in another approach, this time based on the median crack growth trajectory (this time from all 68 tests), the optimal Paris' law parameters were determined to be $m = 3.0$ and $C_{med} = 6.57x10^{-11}$ and $C_{std} = 0.474x10^{-11}$ (with ΔK in $MPa\sqrt{m}$).

Apart from Wang, Spencer et al. [SPEN89] had also tried to estimate these parameters, in this case through linear regression. The obtained values for the parameters were $m = 2.9123$ and $C = 8.096x10^{-11}$ (with ΔK in $MPa\sqrt{m}$).

Table 2: Obtained parameters from Paris' law on different studies (with ΔK in MPa√m)

Approach	m	C_{med} $(x10^{-11})$	C_{std} $(x10^{-11})$
Wang – 1	2.9	8.586	0.619
Wang – 2	3.0	6.57	0.474
Spencer	2.9123	8.096	-

The parameters used as a reference to generate the training dataset will be the ones obtained by Wang in the first approach. The reason for this selection being that a distribution of the C parameter is preferred rather than a single value in order to generate different training examples, and that this first approach delivers similar results to the study of Spencer, which makes it more reliable.

3.2.3. Strain and XFEM

After knowing the parameter's value and the defined distribution, Paris' law is completely defined and the data can already be generated. These parameters can't be introduced directly in XFEM, as a result of it not employing the traditional Paris' law formulation. XFEM formulates the crack growth based on the energy release rate (G), and not based on the stress intensity factor (K).

$\frac{da}{dN} = C(\Delta K)^m$ traditional Paris' law formulation

$\frac{da}{dN} = c_3(\Delta G)^{c_4}$ XFEM's formulation

The energy release rate (G) from the crack tip quantifies the rate of change of the potential energy of a cracked elastic solid as the crack grows [BOWE08]. Luckily, for Mode I of a linearly-elastic material, the relation between the stress intensity ratio and the energy release rate can be defined as: [ZEHN12]

$$G = \frac{K_I^2}{E'}$$

Where E' is:

$$E' = \begin{cases} E, & plane\ stress \\ \dfrac{E}{1-\nu^2}, & plane\ strain \end{cases}$$

In our case, the plane stress assumption is taken. The resulting parameters c_3 and c_4 that will be employed in XFEM can be calculated as:

$$c_3 = C \cdot E^{\frac{m}{2}}$$

$$c_4 = \frac{m}{2}$$

Other parameters regarding the energy release strain and crack growth also have to be given to XFEM, but they won't be discussed here. The case being studied is the rectangular piece with a crack in the middle from Virkler's tests. This piece can be treated as an infinite plate with a crack in the middle and employing Paris' law for fixed crack length increments, delivering very similar

results as the ones obtained on XFEM for Virkler's configuration and crack growth. The simulations of dynamic crack growth are computationally very expensive (a few hours on the university's computers), and in order to generate a sufficient number of examples, a much bigger computational capability should be available. In industrial environments this shouldn't be a problem through the use of available computational clusters. But for Virkler's configuration on XFEM, Paris' law gives a good approximation on dynamic crack growth, so the time series are generated using Paris' law for fixed increments of crack length.

A converged solution for the crack growth can be calculated with steps of $\Delta N = 1$ cycle, and it will be used as reference. Each cycle, the crack length increment and thus the new crack length is calculated. In the case of XFEM, steps aren't taken each increment of number of cycles ΔN, but each time the simulated crack breaks a finite element. The elements in the crack path are designed to have the same length, which means that for a crack growing on its longitudinal direction XFEM provides a datapoint each time the crack length increases that element length. This will be replicated analytically by taking steps each fixed crack length increment Δa. In order to determine a sufficiently accurate Δa (the smaller, the more exact), the crack growth sequence calculated with steps of $\Delta N = 1$ cycle will be used. The convergence of different crack length increments will be checked: $\{0.02, 0.2, 2\}$ mm.

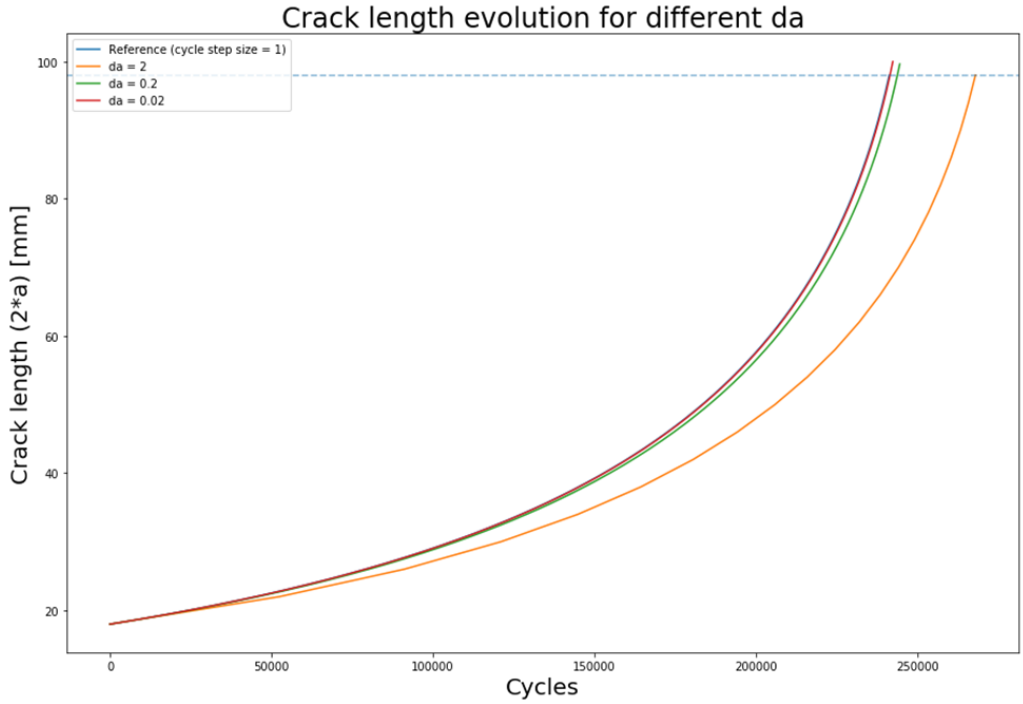


Figure 21: Crack length increment convergence for analytical data generation

The series in blue (the reference series with $\Delta N = 1$ cycle) is almost identical to the series generated by the crack length increment of $0.02\,mm$ (red). The crack length increment of $2\,mm$ (yellow) makes a very inaccurate approximation. The three series are compared to the reference series with the number of cycles elapsed by the last common crack length ($2a = 98\,mm$). The errors in percentage were:

Table 3: Crack length increment assessment through error on the number of cycles elapsed

Series	Number of cycles elapsed	Error
$\Delta N = 1\ cycle$	241 313	0%
$\Delta a = 0.02\ mm$	241 564	0.10%
$\Delta a = 0.2\ mm$	243 853	1.05%
$\Delta a = 2\ mm$	268 017	11.06%

An error of 1% on the number of cycles elapsed was acceptable, and saved a great amount of time when generating the datapoints, so 0.2 mm was the chosen crack length increment to generate the crack growth series. It is worth mentioning that in Virkler's tests, the datapoints were recorded each 0.2 mm of crack length growth.

After having determined that the most appropriate crack length increment was 0.2 mm, the crack growth series were generated. More specifically, 1000 examples were generated by using the Paris' law parameter $m = 2.9$ and sampling the C parameter from the distribution $C_{med} = 8.586x10^{-11}$ and $C_{std} = 0.619x10^{-11}$ (with ΔK in $MPa\sqrt{m}$). The sampled values, as well as the underlying distribution for C are shown below.

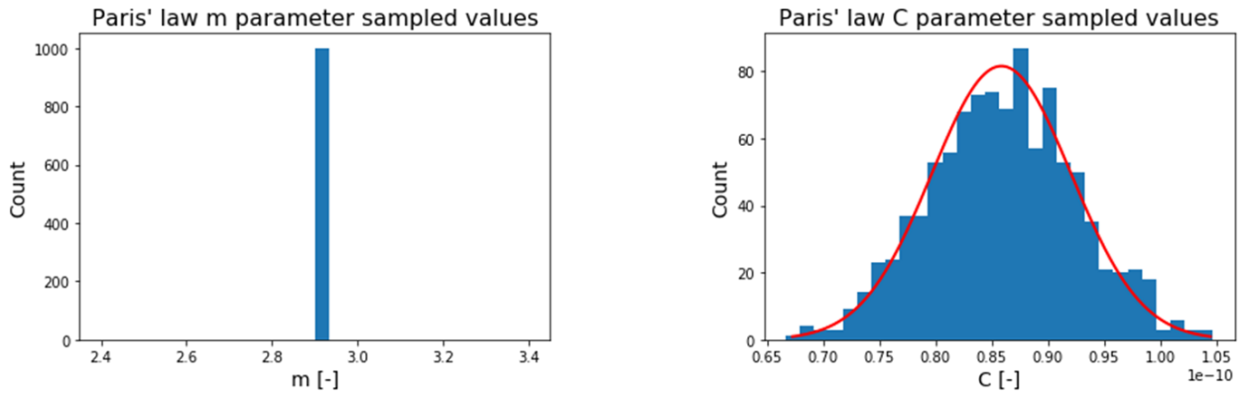


Figure 22: Histograms of the sampled Paris' law parameters

The parameter C has a complicated unit of measurement. The values shown above are in $\left[MPa^{-\boldsymbol{m}} \cdot m^{1-\frac{\boldsymbol{m}}{2}}\right]$ being m meters and $\boldsymbol{m}$ the other Paris' law parameter (exponents in the unit of measure). Other relevant values and properties employed are the initial crack length $2a_0 = 18\ mm$ and the critical fracture toughness $K_{Ic} = 29\ MPa\sqrt{m}$. With all the necessary values defined, the crack growth examples were generated.

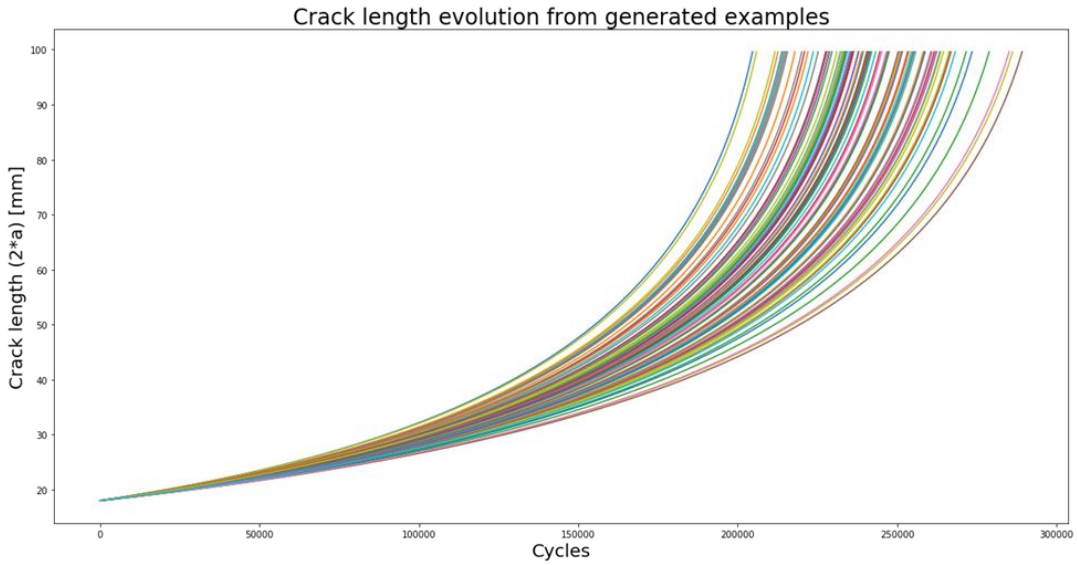


Figure 23: Crack growth analytically generated examples

The resulting last cycle of each series was recorded, showing a log-normal shape with most of the series being around 240000 cycles long.

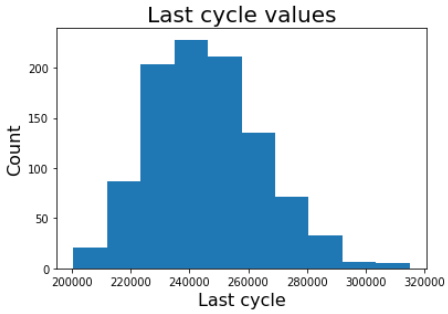


Figure 24: Histogram of the number of cycles elapsed on the analytically generated examples

With the series of crack growth generated, the corresponding strain series were then computed. In order to do so, static crack simulations were performed on XFEM for each crack length considered (from 9.0 mm, each 0.2 mm). The piece built on XFEM had the same dimensions as Virkler's specimens. This is, length of 558.8 mm, width of 152.4 mm and a thickness of 2.54 mm. This geometry was loaded with the maximum load of Virkler's tests $\sigma = 60.34\ MPa$.

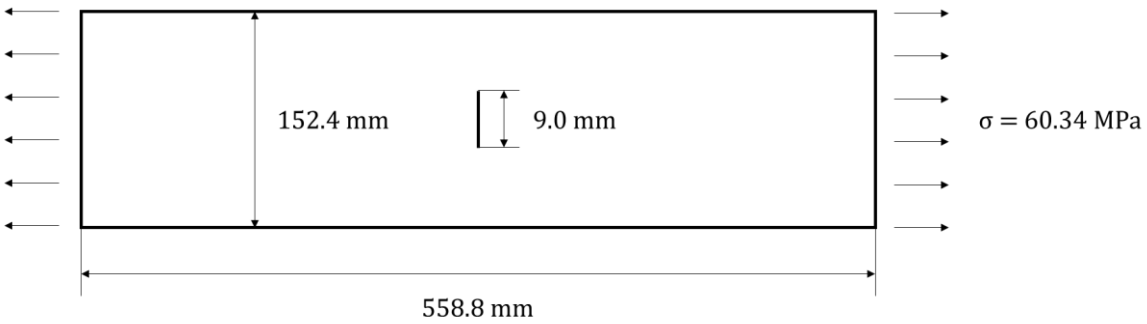


Figure 25: Simulated piece on XFEM based on Virkler's tests

Additionally, the material properties for the 2024-T3 aluminum alloy were introduced. More specifically the material was considered to be elastic and isotropic, and the Young's Modulus $E = 72000\,MPa$ and the Poisson's Ratio $\nu = 0.33$. To determine which strain and at which location the strain should be measured, the strain on the crack's longitudinal direction ε_{xx}, lateral direction ε_{yy} and the shear strain ε_{xy} were calculated. The distribution of each strain for different crack lengths can be seen in Figure 27. The symmetry of the geometry and loads simplifies the simulation, which was performed on half of the piece.

The longitudinal strain (ε_{xx}) was discarded by cause of three main reasons. First of all, the strain through crack lengths isn't monotonic, which means that the same strain is assigned to many different crack lengths. Besides, it presents spikes at certain crack lengths and it maintains a constant value of zero once reached some crack length. These would result in a very poor performance of the algorithm and therefore the longitudinal strain is discarded.

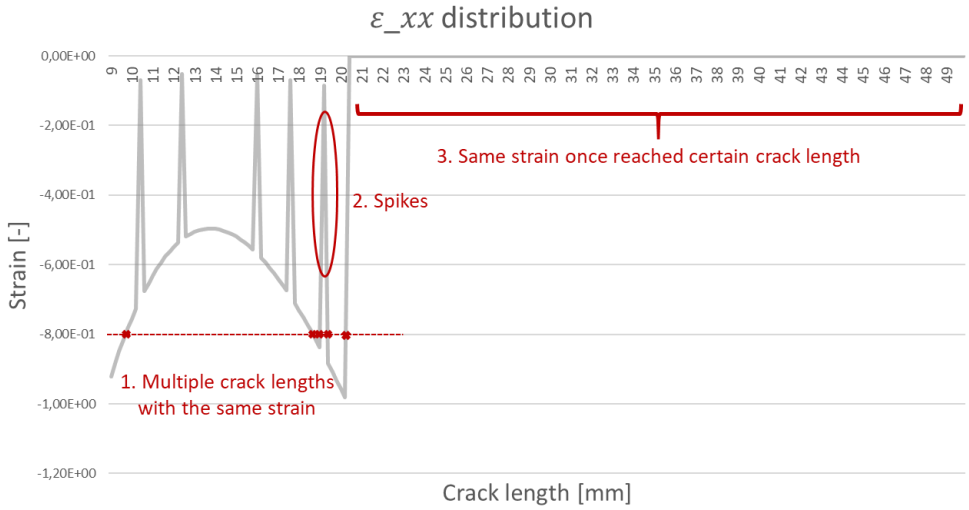


Figure 26: Longitudinal strain through crack lengths on the simulated piece

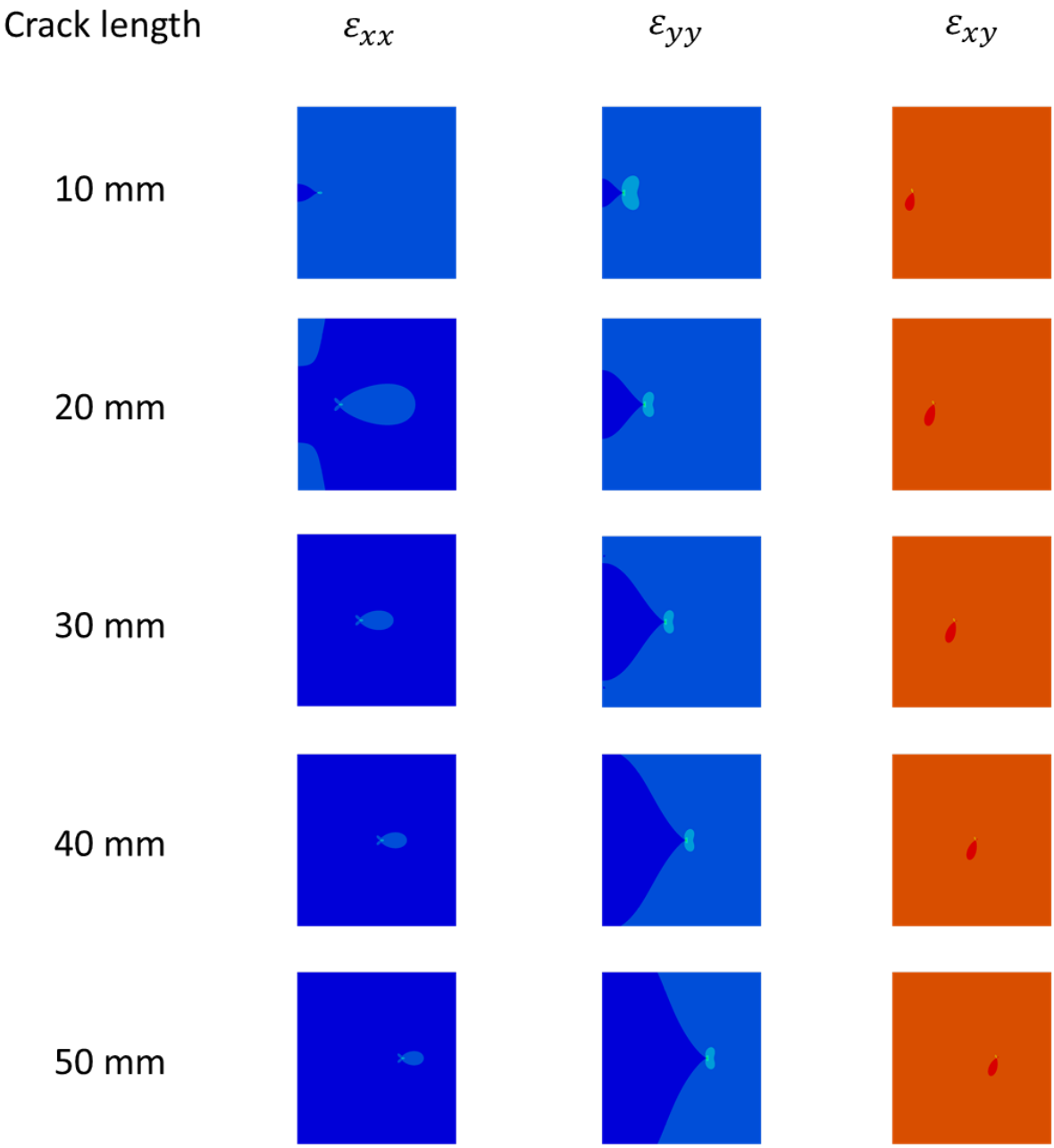


Figure 27: Strain distributions at half piece per crack length on the simulated piece

The shear strain doesn't vary much, which leads to choosing the strain in the lateral direction ε_{yy} as the input variable for our algorithms. The question is where to measure this strain. The lateral strain distributions on Figure 27 lead to think that the biggest variation of strain between crack lengths will occur in the center of the piece (left side of the half piece illustrated). Two different element sizes were employed on the piece: the general mesh size was 30.4 *mm* and in the section close to the crack, the mesh was refined to 0.2 *mm*, with a progressive elements size change between them. For the area closer to the crack, where the mesh was finer, the strain evolution through crack growth is portrayed in Figure 28.

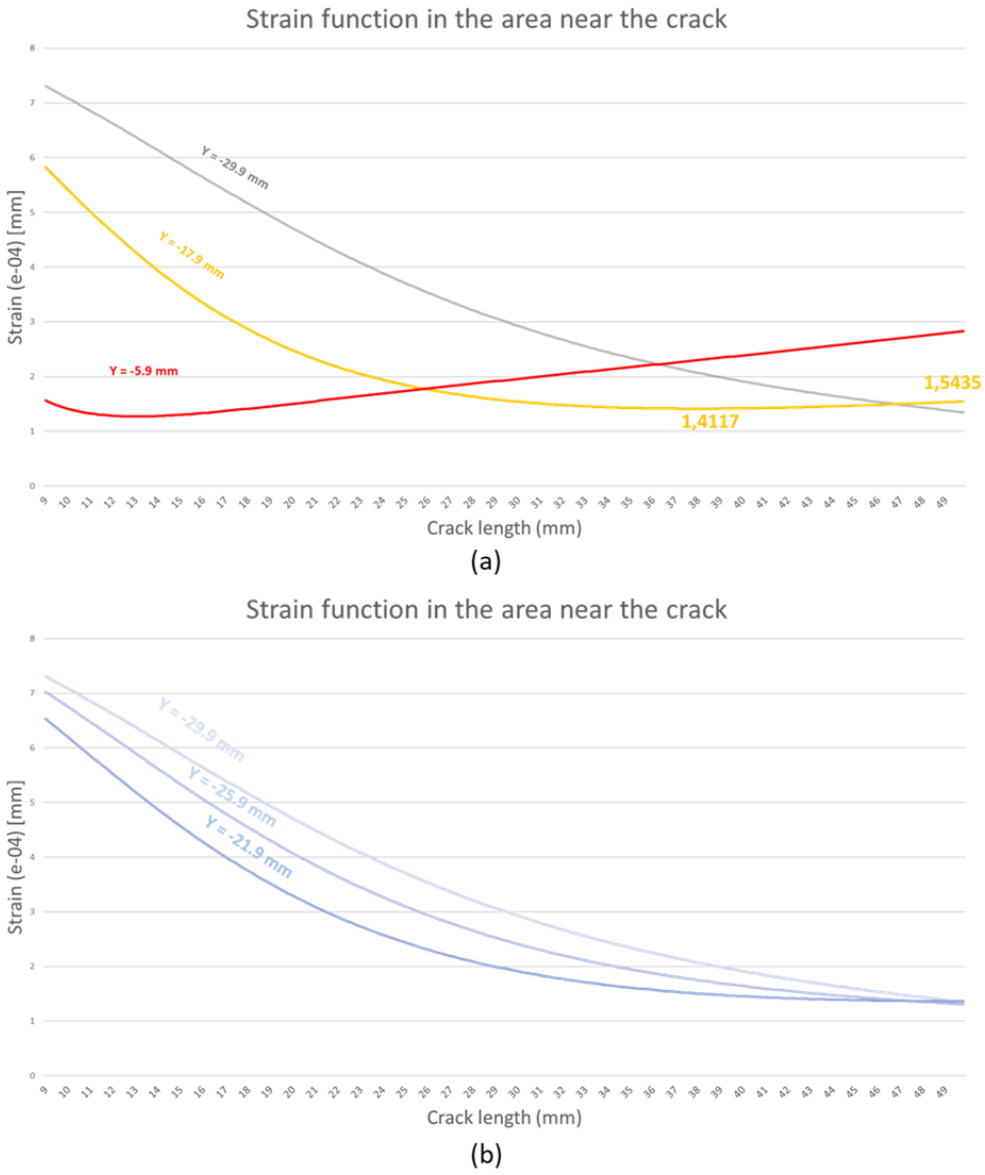


Figure 28: Strain evolution through crack lengths near the crack on the simulated piece

The strain function becomes non-monotonic when measuring too close to the crack as seen in Figure 28 (a). Furthermore, the further from the crack , the greater the variation of strain between crack lengths and the more linear the function. When looking at the points further from the crack, whose strains were computed with bigger element sizes, the results suggested taking points closer to the crack due to a bigger variation between strains for different crack sizes. This variation of strain (between strain at crack length 9.0 *mm* and 49.8 *mm*) was calculated for all the points in the vertical axis (Figure 29). The line in yellow corresponds to the points closer to the crack, with a finer mesh, and the line in blue to the points outside of that region, further from the crack and with a coarser mesh. The initial dip in the yellow line (closer to the mesh) is due to the strain evolution not being monotonic, as seen in Figure 28 (a). The results suggest taking the point at 29.5 *mm* from the crack for a maximum strain variation through crack lengths.

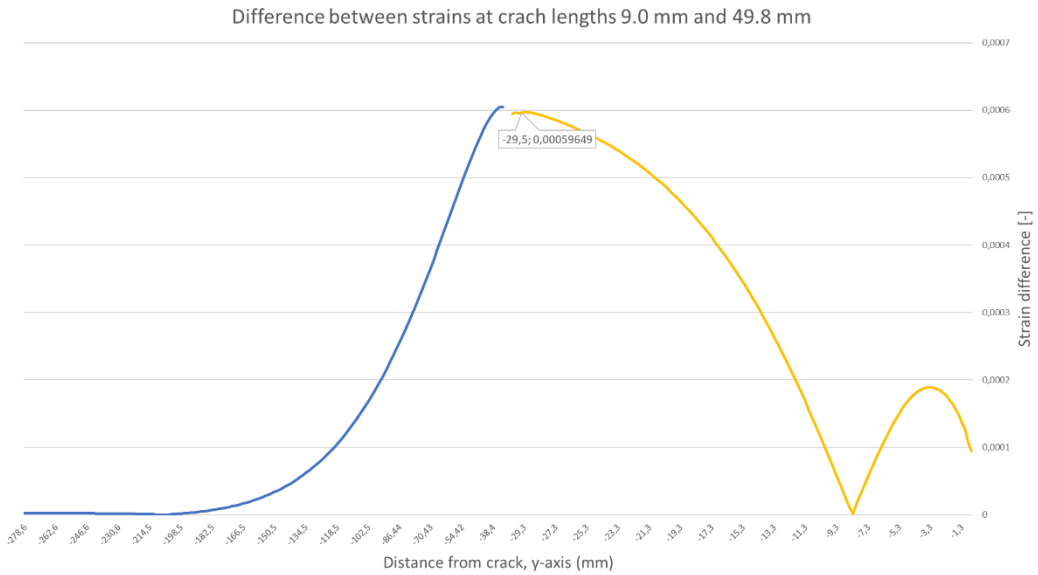


Figure 29: Strain difference along Y-axis on the simulated piece

Therefore, the chosen point for measuring the strain was $x = 0.1\ mm, y = 29.5\ mm$. The strain couldn't be calculated exactly in the middle ($x = 0.0\ mm$) due to the simulation being of half piece, and $x = 0.0\ mm$ being in the edge. The resulting strain evolution for the considered crack lengths can be seen in Figure 30. The variation of strain is one of the highest and the relation between crack length and strain is almost linear, both desired properties for this transformation.

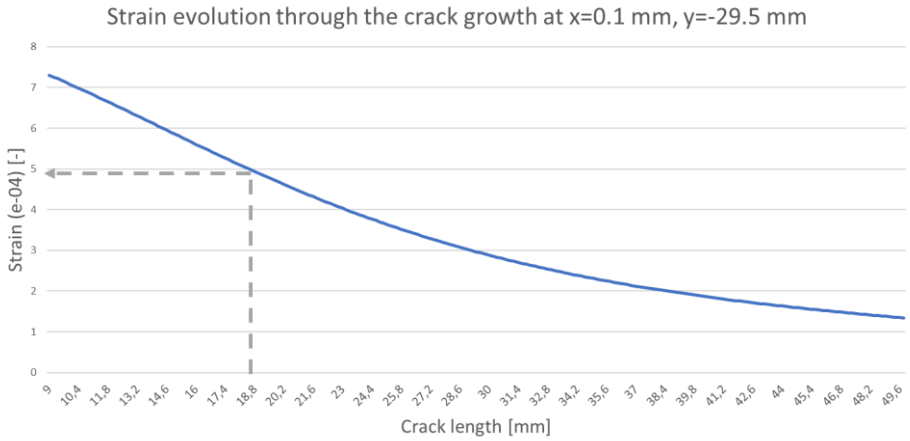


Figure 30: Strain evolution at chosen point on the simulated piece

Given the relationship between crack lengths and strain, the points from the original crack length sequences (Figure 23) were transformed to their respective strain at the chosen point. The crack length sequences' points were taken for the same crack lengths (each 0.2 mm), so the same strains will be obtained, being the difference in the number of cycles elapsed.

To replicate the operation of real measurements of the strains with a strain gauge, a virtual measurement error was introduced on each point. This error would be generated through a random walk, sampling values from a normal distribution of mean zero. As for

the standard deviation, the value to be selected had to emulate the error of strain gauges, which is usually up to ±5%. Thus, 5% of the mean strain value ($5\% * 3.479 = 0.174$) was determined to be the bound error, which for a normal distribution means that the standard deviation is one third of this value for 99.7% of the values having a smaller error than it. The resulting "measured" strain sequences are shown in Figure 31. These sequences are the ones employed as input for the algorithms in order to predict the Remaining Useful Life at every given point. The real crack growth data from Virkler's dataset was also converted to strain following the same transformation (Figure 32). In this case, no error was added because the measured crack lengths by Virkler already had an associated error.

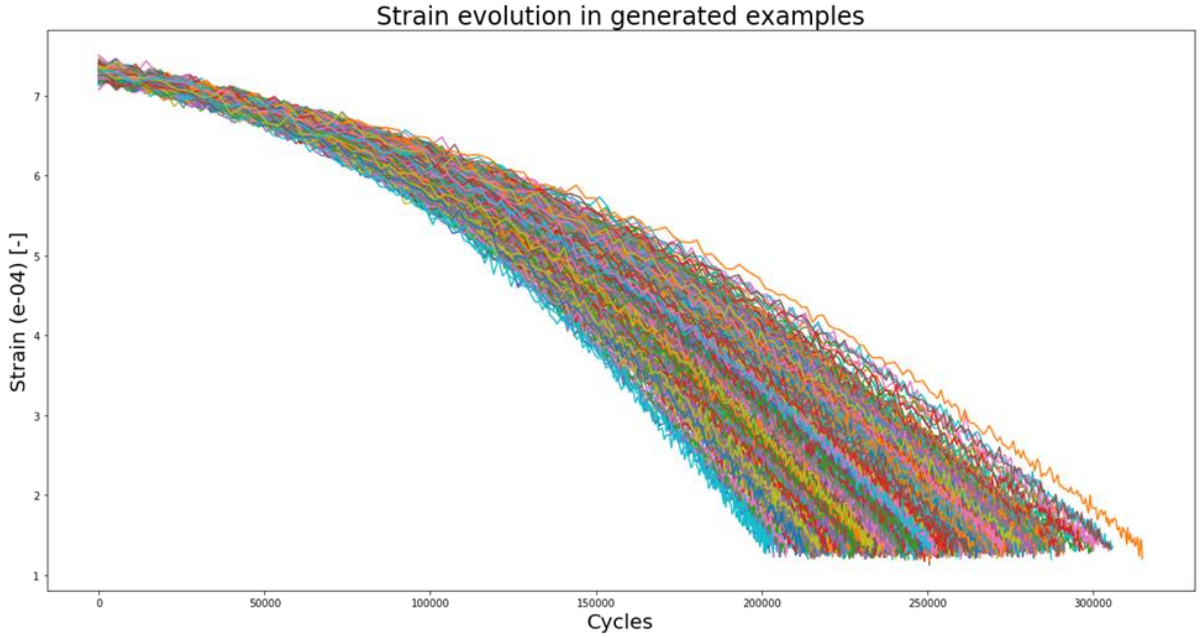


Figure 31: Generated "measured" strain sequences

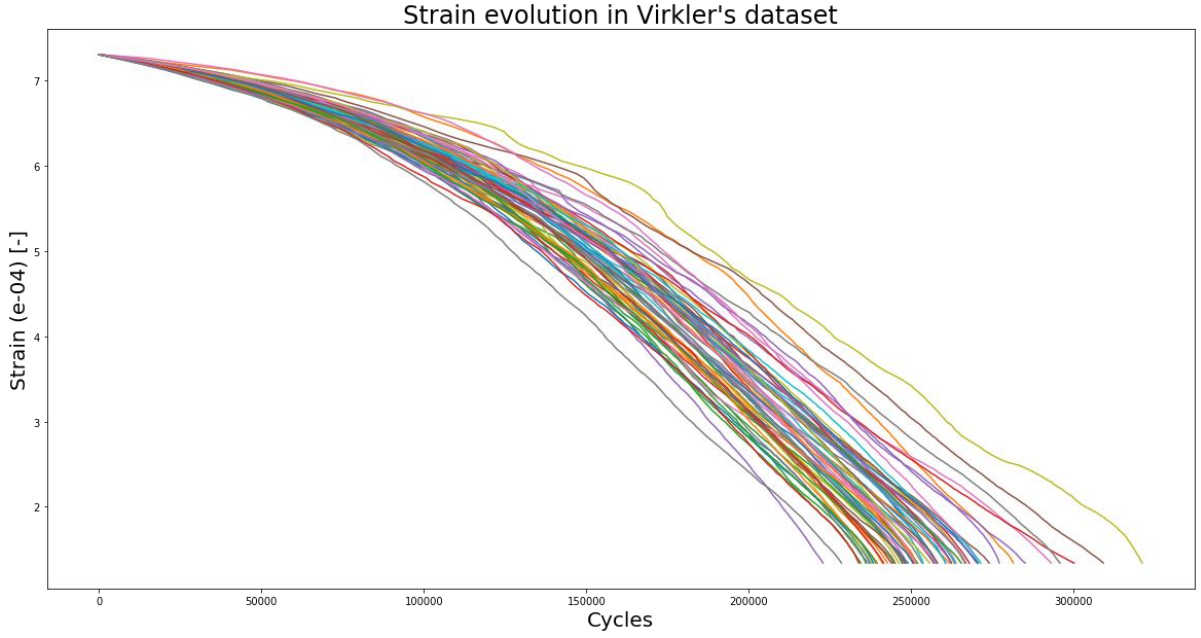


Figure 32: Strain sequences derived from Virkler's dataset

3.3. Algorithms

This section holds a great weight in the dissertation, due to encapsulating the theoretical background behind the two algorithms handled: Support Vector Regression and Recurrent Neural Networks. The models that make the estimations will be characterized, as well as the extensions to make the models capable of providing a confidence interval .

3.3.1. SVR

The main concepts and the foundations of Support Vector Machines were first introduced by Vapnik in 1995 [VAPN95]. The most important concept introduced, and the base of this kind of methods is the Structural Risk Minimization (SRM) principle. For comparison, the traditional Neural Networks are based on the Empirical Risk Minimization (ERM) principle. SRM reduces the upper bound on the Vapnik-Chervonenkis (VC) dimension, or generalization error, while ERM aims at minimizing the error on the training data. This is what provides SVM a great generalization potential that makes it a popular option in statistical learning.

Support Vector Machines are supervised machine learning algorithms capable of solving classification and also regression problems. The nomenclature usually refers to the SVM applied to classification as simply SVM and the SVM applied to regression as SVR. Both type of algorithms will be described in this chapter. To solve nonlinear problems, the mapping to a higher dimensional space through kernel functions will also be explained. SVMs are used for both classification and regression problems.

3.3.1.1. Structural Risk Minimization

In any machine learning algorithm, the training error can be defined as the average error of the predictions from the data in the training set. The error of the prediction can be described through different metrics, but it will be illustrated with the Mean Squared Error.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where y_i is the true value and $\hat{y}_i$ the prediction. Taking MSE as the prediction error function, the training error can be defined as:

$$E_{train} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f_D(\boldsymbol{x}_i)\right)^2$$

Where $(\boldsymbol{x}_i, y_i)$ corresponds to the i^{th} observation from the training set and $f_D(\boldsymbol{x}_i)$ to the prediction made by the best model found based on the training set. Another type of error, which is of great importance in this theory, is the generalization error. The generalization error tells how well the algorithm will do on future data. The future data that will be given to the algorithm is unknown, but the "range" of possible values of $(\boldsymbol{x}, y)$ is available. For example, in a classification problem of images of digits, the input "range" would be all the possible $20x20$ black/white $\{0,1\}$ bitmaps of the image and the output

range are the 10 possible digits {0,1,2 … 9}. The generalization error will be denominated as risk $R[f]$ and can be determined as:

$$R[f] = \int \left(y_i - f(\boldsymbol{x}_i)\right)^2 P(\boldsymbol{x}, y) d\boldsymbol{x} dy$$

Where the error of all possible values of $(\boldsymbol{x}, y)$ is multiplied by the joint probability $P(\boldsymbol{x}, y)$, which reflects how often we expect to see such $(\boldsymbol{x}, y)$. The final goal is to select the most adequate function f to minimize this risk, but the risk can't be calculated because the probability $P(\boldsymbol{x}, y)$ is unknown. Hence an approximation will be made, according to the Empirical Risk Minimization principle:

$$R_{emp}[f] = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\boldsymbol{x}_i)\right)^2$$

Through the Empirical Risk Minimization principle, the selected function would be:

$$f^*(\boldsymbol{x}) = \arg\min_{\mathrm{f}\in \mathrm{H_n}} R_{emp}[f]$$

Where $\mathrm{H_n}$ is the set of all possible functions f. The Empirical Risk is a good approximation if it is an unbiased estimate of the risk, this is:

$$\lim_{n\to\infty} R_{emp}[f] = R[f]$$

This holds true when the training set is very big, but it must also satisfy:

$$\lim_{n\to\infty} \min_{\mathrm{f}\in \mathrm{H_n}} R_{emp}[f] = \min_{\mathrm{f}\in \mathrm{H_n}} R[f]$$

Which will be true if $\mathrm{H_n}$ is a small enough set of functions and the minima converge. As noted before, the risk can't be determined and the empirical risk is used as approximation. The result is a defined bound between the two risks. This bound is determined with probability $1 - \delta$.

$$R[f] \leq R_{emp}[f] + \sqrt{\frac{h\left(\ln\frac{2n}{h} + 1\right) - \ln\frac{\delta}{4}}{n}}$$

Where h is the VC dimension of the hypothesis space. The VC dimension is a scalar that measures the capacity (size or complexity) of a set of functions. In classification problems, h can be defined as: "the largest n such that there exist a ser of examples D_n such that one can always find a function which gives the correct answer for all examples in D_n, for any possible labeling". The capacity can also be calculated for regression problems, but it is more complicated and it can also be derived from an equivalent classification problem. The risk associated to the confidence interval grows with the capacity h (or complexity for easier understanding) of the set of functions considered. The aim is creating a structure such that S_h is a hypothesis space of VC dimension h and solve the Structural Risk Minimization by:

$$\min_{S_h}\left[R_{emp}[f] + \sqrt{\frac{h\left(\ln\frac{2n}{h} + 1\right) - \ln\frac{\delta}{4}}{n}}\right]$$

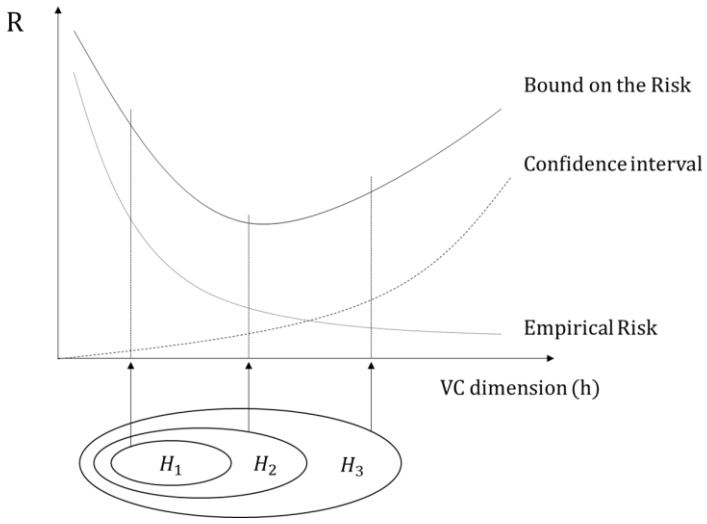


Figure 33: Bound on the risk based on the empirical risk and the VC dimension

The resulting chosen structure that minimizes the structural risk would be H_2.

3.3.1.2. Support Vector Classification

In this section, the SVM principles applied to classification problems are now presented. To simplify the explanation, a classification problem with only two classes will be considered. The goal of the model is to generalize well, this is, to work correctly on unseen examples. In practice this means choosing, from all the possible separating hyperplanes, the hyperplane that maximizes the margin (distance to the closest point of each class, also known as support vectors).

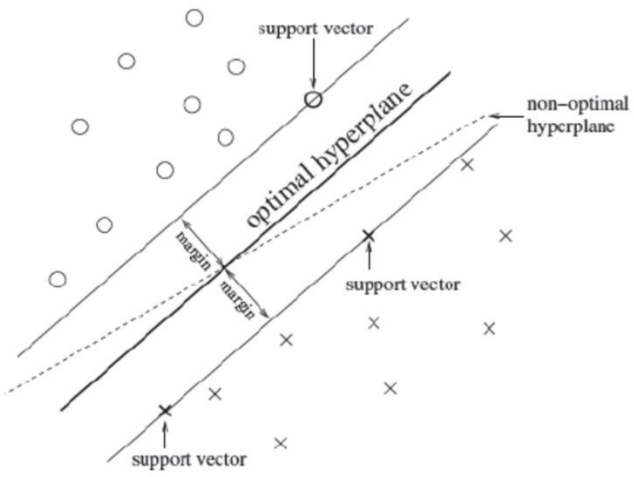


Figure 34: SVM Optimal Separating Hyperplane [GHAN12]

The goal is to separate the set of training vectors $(\boldsymbol{x}_1, y_1) \dots (\boldsymbol{x}_n, y_n), \boldsymbol{x} \in R^n, y \in \{-1, +1\}$ with a hyperplane $(\boldsymbol{w} \cdot \boldsymbol{x}) + b = 0$. A canonical hyperplane will be considered for the explanation:

$$\min_{x_i} |(\boldsymbol{w} \cdot \boldsymbol{x}) + b| = 1$$

Which can be enunciated as: "the norm of the weight vector should be equal to the inverse of the distance, of the nearest point in the dataset to the hyperplane". This distance from the nearest points to the hyperplanes are illustrated in Figure 23.

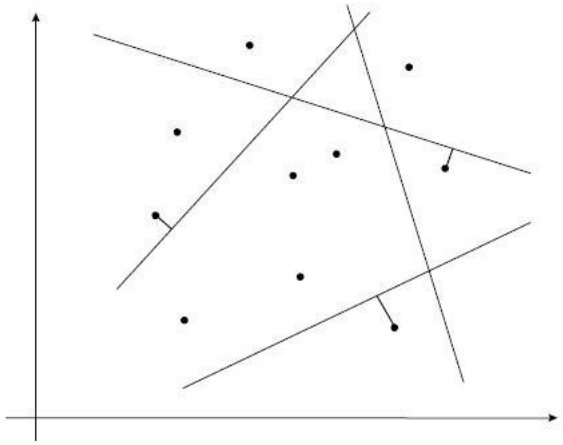


Figure 35: Canonical Hyperplanes [GUNN98]

The separating canonical hyperplanes in a two-classes classification problem must meet the constraints:

$$y_i[(\boldsymbol{w} \cdot \boldsymbol{x_i}) + b] \geq 1, i = 1 \dots n \quad (1)$$

The distance from a point to the hyperplane can be defined as:

$$d(\boldsymbol{w}, b; \boldsymbol{x}) = \frac{|\boldsymbol{w} \cdot \boldsymbol{x} + b|}{\|\boldsymbol{w}\|} \quad (2)$$

As explained before, the goal is to separate the points maximizing the margin. Therefore, with the given constraints, the margin can be calculated as:

$$\begin{aligned} \rho(\boldsymbol{w}, b) &= \min_{\{\boldsymbol{x_i}, : y_i = +1\}} d(\boldsymbol{w}, b; \boldsymbol{x_i}) + \min_{\{\boldsymbol{x_j}, : y_j = -1\}} d(\boldsymbol{w}, b; \boldsymbol{x_j}) \\ &= \min_{\{\boldsymbol{x_i}, : y_i = +1\}} \frac{|\boldsymbol{w} \cdot \boldsymbol{x_i} + b|}{\|\boldsymbol{w}\|} + \min_{\{\boldsymbol{x_j}, : y_j = -1\}} \frac{|\boldsymbol{w} \cdot \boldsymbol{x_j} + b|}{\|\boldsymbol{w}\|} \\ &= \frac{1}{\|\boldsymbol{w}\|} \left(\min_{\{\boldsymbol{x_i}, : y_i = +1\}} |\boldsymbol{w} \cdot \boldsymbol{x_i} + b| + \min_{\{\boldsymbol{x_j}, : y_j = -1\}} |\boldsymbol{w} \cdot \boldsymbol{x_j} + b| \right) = \frac{2}{\|\boldsymbol{w}\|} \end{aligned}$$

This margin should be maximized, so the function to be minimized can be written as:

$$\Phi(\mathbf{w}) = \frac{1}{2} \|\boldsymbol{w}\|^2 \quad (3)$$

This minimization implies applying the Structural Risk Minimization. This can be illustrated by assuming the following:

$$\|\boldsymbol{w}\| \leq A$$

Taking into account equations (1) and (2):

$$d(\boldsymbol{w}, b; \boldsymbol{x}) \geq \frac{1}{A}$$

Which means that the hyperplanes should be no closer than $\frac{1}{A}$ from the datapoints. This reduces the possible hyperplanes to be chosen, and therefore the capacity h.

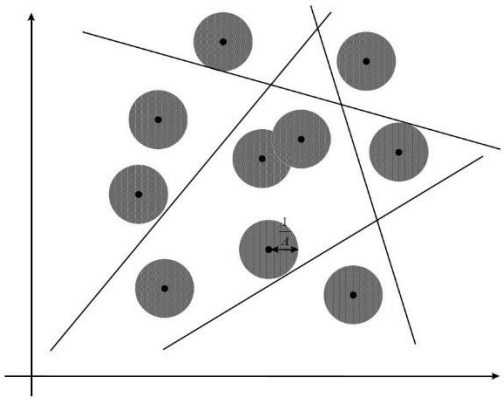


Figure 36: Constrained Canonical Hyperplanes [GUNN98]

If the dimensional space is n, the cited capacity h (VC dimension) can be computed as:

$$h \leq \min[R^2 A^2, n] + 1 \tag{4}$$

Where R is the radius of a hypersphere that includes all the datapoints. This way, minimizing equation (3) is equivalent to minimizing the upper bound on h.

Putting it all together, the optimization problem from equation (3) considering the constraints (1) can be calculated through the Lagrangian:

$$L(\boldsymbol{w}, b, \alpha) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{n} \alpha_i \{[(\boldsymbol{x}_i \cdot \boldsymbol{w}) + b] y_i - 1\} \tag{5}$$

This Lagrangian must be minimized with respect to $\boldsymbol{w}, b$ and maximized with respect to the Lagrange multipliers from the constraints, $\alpha_i \geq 0$. Through the classical Lagrangian duality, the primal problem (5) can be transformed to a dual problem:

$$\max_{\alpha} W(\alpha) = \max_{\alpha} \left\{ \min_{\boldsymbol{w}, b} L(\boldsymbol{w}, b, \alpha) \right\} \tag{6}$$

Derivation with respect to $\boldsymbol{w}$ and b yields:

$$\frac{\partial L}{\partial b} = 0 \Longrightarrow \sum_{i=1}^{n} \alpha_i \cdot y_i = 0 \tag{7}$$

$$\frac{\partial L}{\partial \boldsymbol{w}} = 0 \Longrightarrow \boldsymbol{w} = \sum_{i=1}^{n} \alpha_i \cdot \boldsymbol{x_i} \cdot y_i \tag{8}$$

Through equations (4), (5), (6) and (7), the dual problem can be formulated as:

$$\max_{\alpha} W(\alpha) = \max_{\alpha} -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j(\boldsymbol{x_i} \cdot \boldsymbol{x_j}) + \sum_{j=1}^{n} \alpha_i \tag{9}$$

And thus the solution to this problem can be calculated as:

$$\bar{\alpha} = \arg\min_{\alpha} \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j(\boldsymbol{x_i} \cdot \boldsymbol{x_j}) - \sum_{j=1}^{n} \alpha_i \tag{10}$$

With the following constraints:

$$\begin{gathered} \alpha_i \geq 0, i = 1 \ldots n \\ \sum_{i=1}^{n} \alpha_i \cdot y_i = 0 \end{gathered} \tag{11}$$

The Lagrangian multipliers α_i can then be calculated by solving the equation (10) with the constraints given in (11). And hence, the resulting optimal separating hyperplane is described as:

$$\bar{\boldsymbol{w}} = \sum_{i=1}^{n} \bar{\alpha}_i \cdot \boldsymbol{x_i} \cdot y_i$$

$$\bar{b} = -\frac{1}{2}\bar{\boldsymbol{w}} \cdot [\boldsymbol{x_r} + \boldsymbol{x_s}]$$

Where $\boldsymbol{x_r}$ and $\boldsymbol{x_s}$ are any of the support vector from each of the class that satisfy:

$$\bar{\alpha}_r, \bar{\alpha}_s > 0, \qquad y_r = +1, \qquad y_s = -1$$

Accordingly the (hard) classifier's function is:

$$f(\boldsymbol{x}) = sign(\bar{\boldsymbol{w}} \cdot \boldsymbol{x} + \bar{b})$$

A soft classifier can also be employed through interpolation within the margin. For unknown data that falls in the region of the margin, this soft classifier outputs a real value according to the proximity to the considered classes.

$$f(\boldsymbol{x}) = h(\bar{\boldsymbol{w}} \cdot \boldsymbol{x} + \bar{b}) \quad with \quad h(x) = f(x) = \begin{cases} -1, & x < -1 \\ x, & -1 \leq x \leq 1 \\ 1, & x > 1 \end{cases}$$

From the KKT (Karush–Kuhn–Tucker) conditions:

$$\bar{\alpha}_i[y_i(\bar{\boldsymbol{w}} \cdot \boldsymbol{x_i} + \bar{b}) - 1] = 0$$

Only the points where $y_i(\bar{\boldsymbol{w}} \cdot \boldsymbol{x_i} + \bar{b}) = 1$ the Lagrange multipliers will be non-zero. These points are the Support Vectors and the hyperplane is determined only by this small subset of datapoints. If the data is linearly separable, all the Support Vectors will lie on the edge of the margin, and the following equation will remain true:

$$\|\boldsymbol{w}\|^2 = \sum_{i=1}^{n} \bar{\alpha}_i = \sum_{SVs} \bar{\alpha}_i = \sum_{SVs}\sum_{SVs} \bar{\alpha}_i \cdot \bar{\alpha}_j \cdot y_i \cdot y_j(\boldsymbol{x_i} \cdot \boldsymbol{x_j})$$

Finally, through equation (4) the capacity h (VC dimension) is bounded by:

$$h \leq \min\left[R^2 \sum_{SVs} \overline{\alpha_i}, n\right] + 1$$

Which if the data is normalized (hypersphere of radius $R = 1$) can be expressed as:

$$h \leq 1 + \mathrm{n} \cdot \min\left[\sum_{SVs} \overline{\alpha_i}, 1\right]$$

3.3.1.3. Generalization of the Optimal Separating Hyperplane

The previous explanation was performed assuming that the data is linearly separable, but this is generally not the case. There might be inherent noise in the training dataset that makes it non-linearly separable or the optimal hyperplane that separates the classes might not be linear. Depending on which case applies, two different approaches are taken: the approach for the first case will be detailed in this section, and the approach for the second case, on the section "3.3.1.4 Higher Dimensional Feature Space".

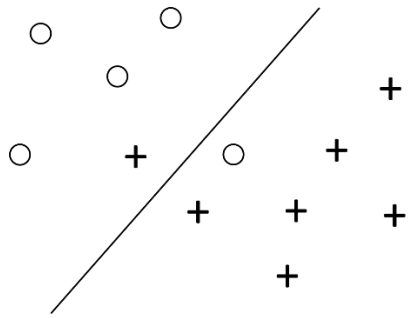


Figure 37: Generalized Optimal Separating Hyperplane

To solve the problem portrayed on Figure 37, the non-negative variables $\xi_i \geq 0$ and a penalty function will be utilized.

$$F_\sigma(\xi) = \sum_{i=1}^{n} \xi_i^\sigma, \quad \sigma > 0$$

Being the variables ξ a measure of the classification error, which we will be tried to minimize. The new optimization problem will now involve minimizing the bound on the capacity h (VC dimension) and minimizing the classification error. The constraints (1) will be rewritten as:

$$y_i[(\boldsymbol{w} \cdot \boldsymbol{x_i}) + b] \geq 1 - \xi_i, \qquad i = 1 \ldots n \tag{12}$$

And the new function to be minimized, previously (3) for the linear case, is now:

$$\Phi(\mathbf{w}, \xi) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_{i=1}^{n} \xi_{\mathrm{i}} \tag{13}$$

C is a hyperparameter that will have to be tuned. This function must also hold true under the constraints (12). The resulting Langrangian for this optimization problem can be written as:

$$L(\boldsymbol{w}, b, \xi, \alpha, \beta) = \frac{1}{2}(\boldsymbol{w} \cdot \boldsymbol{w}) + C\sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i\{[(\boldsymbol{x}_i \cdot \boldsymbol{w}) + b]y_i - 1 + \xi_i\} - \sum_{i=1}^{n} \beta_i \cdot \xi_i \tag{14}$$

This Lagrangian must be minimized with respect to $\boldsymbol{w}, b, \xi$ and maximized with respect to the Lagrange multipliers from the constraints, $\alpha_i, \beta_i \geq 0$. Through the classical Lagrangian duality, the primal problem (14) can be transformed to a dual problem.

$$\max_{\alpha,\beta} W(\alpha) = \max_{\alpha,\beta}\left\{\min_{\boldsymbol{w},b,\xi} L(\boldsymbol{w}, b, \xi, \alpha, \beta)\right\} \tag{15}$$

Derivation with respect to $\boldsymbol{w}$, b and ξ_i yields:

$$\begin{gathered} \frac{\partial L}{\partial b} = 0 \Longrightarrow \sum_{i=1}^{n} \alpha_i \cdot y_i = 0 \\ \frac{\partial L}{\partial \boldsymbol{w}} = 0 \Longrightarrow \boldsymbol{w} = \sum_{i=1}^{n} \alpha_i \cdot \boldsymbol{x_i} \cdot y_i \\ \frac{\partial L}{\partial \xi} = 0 \Longrightarrow \alpha_i + \beta_i = C \end{gathered} \tag{16}$$

Through equations (14), (15) and (16), the dual problem can be formulated as:

$$\max_{\alpha} W(\alpha) = \max_{\alpha} -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j(\boldsymbol{x_i} \cdot \boldsymbol{x_j}) + \sum_{j=1}^{n} \alpha_i \tag{17}$$

And thus the solution to this problem can be calculated as:

$$\bar{\alpha} = \arg\min_{\alpha} \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j(\boldsymbol{x_i} \cdot \boldsymbol{x_j}) - \sum_{j=1}^{n} \alpha_i \tag{18}$$

With the following constraints:

$$\begin{gathered} 0 \leq \alpha_i \leq C, i = 1 \ldots n \\ \sum_{i=1}^{n} \alpha_i \cdot y_i = 0 \end{gathered} \tag{19}$$

Which yields exactly the same solution as for the linearly separable problem except for a slight adjustment through the coefficient C in the bounds of the Lagrange multipliers α_i. As explained before, this coefficient is a hyperparameter whose value has to be selected based on previous knowledge on the noise of the data.

3.3.1.4. Higher Dimensional Feature Space

Both of the approaches that have been discussed so far consider a linear boundary as the most adequate solution. In some cases the dataset is better separated through a non-linear boundary. The approach for these cases implies mapping the input vectors $\boldsymbol{x}$ to a higher dimensional feature space $\boldsymbol{z}$. Through a non-linear mapping, SVM builds a separating hyperplane in this higher dimensional space and is then mapped back to the original dimensions. This idea can is illustrated on

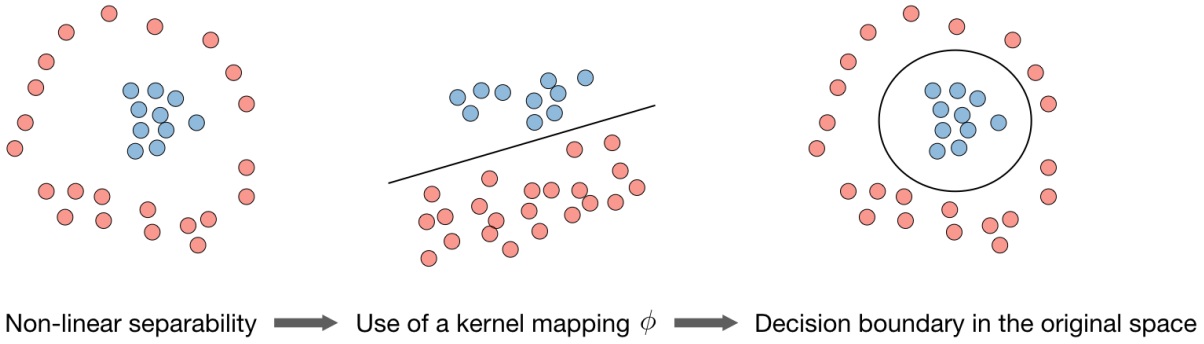


Figure 38: Nonlinear Boundary through Higher Dimensional Space [AMID18]

The functions employed for this mapping to higher dimensional spaces must meet certain requirements that will be discussed in the next section "3.3.1.5 Kernel Functions". Fortunately, the optimization problem (10) doesn't change much, and becomes:

$$\bar{\alpha} = \arg\min_{\alpha} \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j \cdot K(\boldsymbol{x_i}, \boldsymbol{x_j}) - \sum_{j=1}^{n} \alpha_i \qquad (20)$$

Where $K(x, y)$ is the output of the kernel function, the non-linear mapping to the feature space (higher dimensional space). The constraints remain the same:

$$\begin{gathered} \alpha_i \geq 0, i = 1 \ldots n \\ \sum_{i=1}^{n} \alpha_i \cdot y_i = 0 \end{gathered} \qquad (21)$$

And again, the Lagrange multipliers can be calculated by solving equation (20) with the constraints on equation (21). The resulting boundary for a separating hyperplane in a higher dimensional space can be expressed as:

$$f(\boldsymbol{x}) = sign\left(\sum_{SVs} \bar{\alpha}_i \cdot y_i \cdot K(\boldsymbol{x_i}, \boldsymbol{x}) + \bar{b}\right)$$

Where the resulting weight and bias vectors can be calculated as:

$$\bar{\boldsymbol{w}} \cdot \boldsymbol{x} = \sum_{SVs} \bar{\alpha}_i \cdot y_i \cdot K(\boldsymbol{x_i}, \boldsymbol{x})$$

$$\bar{b} = -\frac{1}{2}\sum_{SVs} \bar{\alpha}_i \cdot y_i[K(\boldsymbol{x_r}, \boldsymbol{x_i}) + K(\boldsymbol{x_s}, \boldsymbol{x_i})]$$

As explained in the previous section, $\boldsymbol{x_r}$ and $\boldsymbol{x_s}$ are any of the support vector from each of the class that satisfy:

$$\bar{\alpha}_r, \bar{\alpha}_s > 0, \qquad y_r = +1, \qquad y_s = -1$$

The Kernel used for the mapping might have a bias term, which simplifies the classifier. The equality constraint portrayed in equation (21) is no longer needed and the classifier's function becomes:

$$f(\boldsymbol{x}) = sign\left(\sum_{SVs} \bar{\alpha}_i \cdot y_i \cdot K(\boldsymbol{x_i}, \boldsymbol{x})\right)$$

3.3.1.5. Kernel Functions

As seen in the previous section, best boundary for a given dataset might not be a linear separating hyperplane in the currently considered dimensions, but mapping the datapoints to a higher dimensional space (or feature space) and calculate there a linear separating hyperplane. The downfall of this approach is the computational cost that representing all the points in a higher dimensional space has. The way around this problem is, instead of calculating the representation of the points in the higher dimension, calculating a similarity measure (dot product) in this feature space and employ algorithms that only need this value. A Kernel Function takes the points (vectors) in the original dimensional space and calculates the dot product of them in the feature space. This can formally be expressed as:

$$K(\boldsymbol{x}, \boldsymbol{y}) = k(\boldsymbol{x}) \cdot k(\boldsymbol{y})$$

Where $\boldsymbol{x}, \boldsymbol{y}$ are the input data in the original dimensional space. Furthermore, not any function can be used as the kernel function, Mercier's Condition has to be met. This condition reveals whether a potential kernel is indeed a dot product in some space. It can formally be enunciated as follows:

"If and only if, for any $g(\boldsymbol{x})$ such that $\int g(\boldsymbol{x})^2 \, d\boldsymbol{x}$ is finite, then $\iint K(\boldsymbol{x}, \boldsymbol{y}) g(\boldsymbol{x}) g(\boldsymbol{y}) \, d\boldsymbol{x} d\boldsymbol{y} \geq 0$"

The three most common Kernel Functions, which of course satisfy Mercier's Condition are:

Gaussian RBF Kernel: $K(\boldsymbol{x}, \boldsymbol{y}) = e^{-\frac{|x-y|}{2\sigma^2}}$

Polynomial Kernel: $K(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} \cdot \boldsymbol{y})^d$ or alternatively $K(\boldsymbol{x}, \boldsymbol{y}) = \big((\boldsymbol{x} \cdot \boldsymbol{y}) + 1\big)^d$ for $d \in \mathbb{N}$

Sigmoid Kernel: $K(\boldsymbol{x}, \boldsymbol{y}) = \tanh(a(\boldsymbol{x} \cdot \boldsymbol{y}) + b)$

3.3.1.6. Support Vector Regression

Finally, the SVM approach to regression problems is presented with its different variants. First, linear regression will be described and then its non-linear variant, which is the model employed in the research. The principles will be the same as for the classification problem, but this time with the addition of a loss function.

Linear Regression

This time, the goal is to approximate the set of training vectors $(\boldsymbol{x}_1, y_1) \dots (\boldsymbol{x}_n, y_n), \boldsymbol{x} \in R^n, y \in R$ with a linear function $f(x) = (\boldsymbol{w} \cdot \boldsymbol{x}) + b$. This function can be found through the minimization of the function:

$$\Phi(\mathbf{w}, \xi^*, \xi) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\left(\sum_{i=1}^{n} \xi_i + \sum_{i=1}^{n} \xi_i^*\right)$$

Where C is a selected hyperparameter that will have to be tuned, and ξ^*, ξ are slack variables that represent the upper and lower (soft) constraints. This function will be utilized as the loss function for regression, with some specifications. The loss function has to provide a measure of distance from the boundary. The following options would be available:

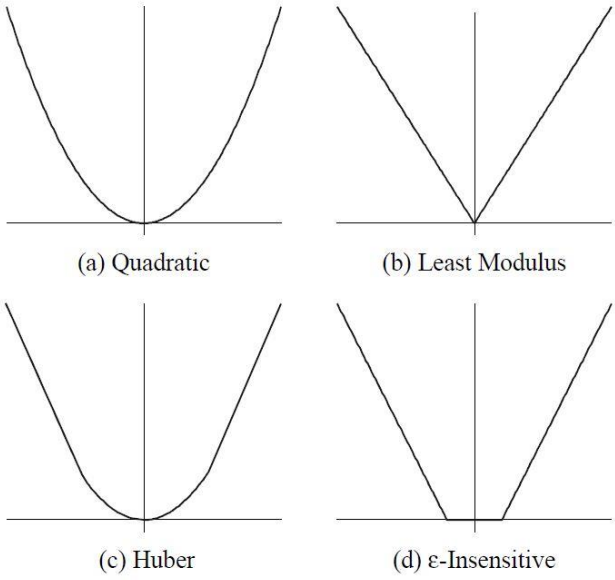


Figure 39: Loss Funtions [GUNN98]

Least squared error's loss function (Figure 39(a)) is more sensitive to outliers than the Laplacian loss function (Figure 39(b)). Huber's proposed loss function (Figure 39(c)) is a more robust, hybrid version of the previous two. But for generalization purposes, the loss function must produce sparseness in the support vectors, and none of the previously mentions loss functions does it. In consequence, Vapnik proposed an approximation to Huber's loss function that allows a sparse set of support vectors (Figure 39(d)). This function, also called ε-insensitive loss function, can be mathematically expressed as:

$$L_\varepsilon = \begin{cases} 0, & for \quad |f(\boldsymbol{x}) - y| < \varepsilon \\ |f(\boldsymbol{x}) - y| - \varepsilon, & otherwise \end{cases}$$

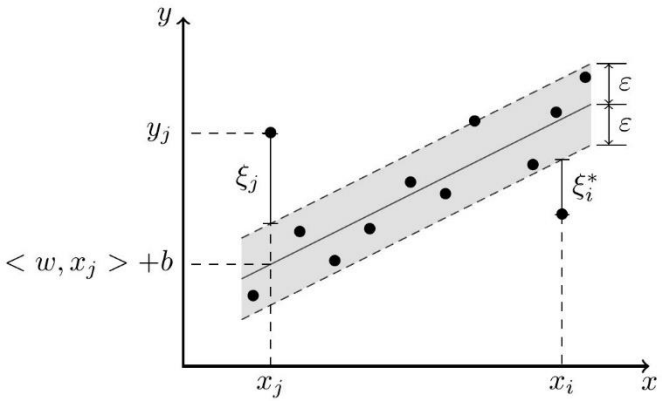


Figure 40: Support Vector Regression [MOUS18]

After calculating the Lagrangian and minimizing it with respect to $\boldsymbol{w}, b$, the solution can be obtained through:

$$\max_{\alpha,\alpha^*} W(\alpha,\alpha^*) = \max_{\alpha,\alpha^*} \left\{ -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}(\alpha_i-\alpha_i^*)(\alpha_j-\alpha_j^*)\left(\boldsymbol{x_i}\cdot\boldsymbol{x_j}\right) + \sum_{j=1}^{n}\alpha_i(y_i-\varepsilon)-\alpha_i^*(y_i+\varepsilon)\right\} \tag{22}$$

And the solution to this optimization is:

$$\bar{\alpha},\bar{\alpha}^* = \arg\min_{\alpha,\alpha^*} \left\{ \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}(\alpha_i-\alpha_i^*)(\alpha_j-\alpha_j^*)\left(\boldsymbol{x_i}\cdot\boldsymbol{x_j}\right) - \sum_{j=1}^{n}\alpha_i(y_i-\varepsilon)+\alpha_i^*(y_i+\varepsilon)\right\} \tag{23}$$

With the following constraints:

$$\begin{aligned} 0 \le \alpha_i \le C, \quad & i = 1 \dots n \\ 0 \le \alpha_i^* \le C, \quad & i = 1 \dots n \\ \sum_{i=1}^{n}(\alpha_i-\alpha_i^*) &= 0 \end{aligned} \tag{24}$$

The Lagrangian multipliers α_i, α_i^* can then be calculated by solving the equation (22) with the constraints given in (24). And hence, the resulting optimal regression function is:

$$\bar{\boldsymbol{w}} = \sum_{i=1}^{n}(\bar{\alpha}_i-\bar{\alpha}_i^*)\cdot\boldsymbol{x_i}$$

$$\bar{b} = -\frac{1}{2}\bar{\boldsymbol{w}}\cdot[\boldsymbol{x_r}+\boldsymbol{x_s}]$$

From the KKT (Karush–Kuhn–Tucker) conditions:

$$\bar{\alpha}_i, \bar{\alpha}_i^* = 0, \qquad i = 1 \dots n$$

Which means that the support vectors will be those where one of the Lagrange multipliers $\bar{\alpha}_i, \bar{\alpha}_i^*$ is greater than zero.

Non-linear Regression

As explained in section "3.3.1.4. Higher Dimensional Feature Space", in order to build a non-linear classification boundary the separation has to be performed in a higher dimensional space, or feature space. For the regression problem, to approach the points through a non-linear function, it will also be done with a linear regression the feature space. Here the kernel trick will also be used to face the cost of mapping every point to the higher dimensional space. The loss function will be, just like for the linear case, the ε-insensitive loss function.

$$\max_{\alpha,\alpha^*} W(\alpha, \alpha^*) = \max_{\alpha,\alpha^*} \left\{ -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} (\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j) \cdot K(\boldsymbol{x_i}, \boldsymbol{x_j}) + \sum_{j=1}^{n} \alpha_i^*(y_i + \varepsilon) - \alpha_i(y_i - \varepsilon) \right\} \tag{25}$$

With the following constraints:

$$\begin{gathered} 0 \le \alpha_i \le C, \qquad i = 1 \dots n \\ 0 \le \alpha_i^* \le C, \qquad i = 1 \dots n \\ \sum_{i=1}^{n} (\alpha_i^* - \alpha_i) = 0 \end{gathered} \tag{26}$$

Then the Lagrange multipliers are calculated by solving equation (25) with the constraints on equation (26). The resulting regression function can be expressed as:

$$f(\boldsymbol{x}) = \sum_{SVs} (\bar{\alpha}_i - \bar{\alpha}_i^*) \cdot K(\boldsymbol{x_i}, \boldsymbol{x}) + \bar{b}$$

Where the resulting weight and bias vectors can be calculated as:

$$\bar{\boldsymbol{w}} \cdot \boldsymbol{x} = \sum_{SVs} (\bar{\alpha}_i - \bar{\alpha}_i^*) \cdot K(\boldsymbol{x_i}, \boldsymbol{x})$$

$$\bar{b} = -\frac{1}{2} \sum_{SVs} (\bar{\alpha}_i - \bar{\alpha}_i^*)[K(\boldsymbol{x_r}, \boldsymbol{x_i}) + K(\boldsymbol{x_s}, \boldsymbol{x_i})]$$

Just like for the Support Vector Classification, the Kernel used for the mapping could have a bias term, which simplifies the regressor's function. The regressor's function becomes:

$$f(\boldsymbol{x}) = \sum_{SVs} (\bar{\alpha}_i - \bar{\alpha}_i^*) \cdot K(\boldsymbol{x_i}, \boldsymbol{x})$$

3.3.1.7. Distribution Estimation

Until this point, all the relevant information regarding the theory behind Support Vector Machines, and more specifically Support Vector Regression. The non-linear SVR is the chosen model to estimate the Remaining Useful Life based on the strain history. The problem concerning this regression model is that the estimation doesn't offer a probabilistic output from which a confidence interval can be built, but rather a single estimation for the given input. Consequently, some changes will be introduced to predict the distribution instead of just one value.

To do so, first it will be assumed that our desired prediction follows a normal distribution. The distribution estimator will predict an estimate error for the prediction our original model (SVR) is making. The resulting estimation is a gaussian function centered in the prediction of the SVR and a variance of the predicted error. The steps are detailed below:

1. Train the SVR model with the strain sequences as input and RUL as output
2. Make the base predictions (will be used as mean of the distribution)
3. Calculate the error (Mean Squared Error) of the mean predictions versus the real values
4. Train the "Variance" model with the strain sequences as input and the error from the previous step as output
5. Make the error predictions (will be used as variance of the distribution)

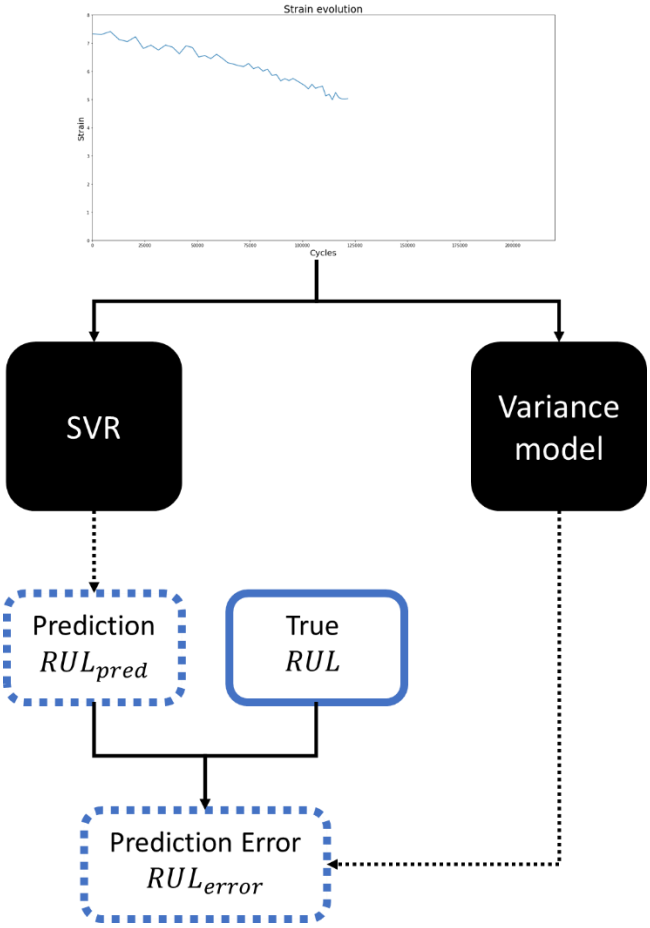


Figure 41: SVR distribution estimator

In step 3, when calculating the error of the mean predictions, the error measurement is the Mean Squared Error (MSE), which is computed as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

The chosen model to predict the error was K-Nearest Neighbors (KNN). A simple model was searched to approximate the error in order not to make the resulting confidence interval too complex, and the KNN model had proven to work better than alternatives such as Gradient Boosting when predicting this error in other cases [QUCI19]. KNN applied to regression chooses the 'n' closest points to the one being analyzed, and returns as prediction the average of those points.

3.3.1.8. Training, validation and test sets

To train any model, a training set has to be gathered. Once trained, testing the model on a validation set before testing it on the real data (test set) can help avoiding tuning the model to perform well explicitly on the given test set, which would mean losing generalization potential. The fact that the model itself is composed of two models requires some changes when deciding what the training and validation sets should be. In this case, each model (the SVR model to predict the mean and the KNN model to predict the standard deviation) will have assigned half of the available data. Each of these halves will be split in training and validation sets for each model.

The training sets are formed by many generated sequences of strain evolution. This sequences will be split in non-overlapping windows as shown in Figure 42. These windows split the complete sequence in partial sequences each defined number of cycles. Due to the nature of the data, most points are located towards the end of the complete sequence, which is also the most relevant part of the sequence because it is where the failure is closest. For this reason, the first created window will have the upper bound at the last cycle count, and the lower bound some defined number of cycles before (window length). The following window will have the upper bound where the previous window's lower bound was and the lower bound some defined number of cycles before (window length), and so on.

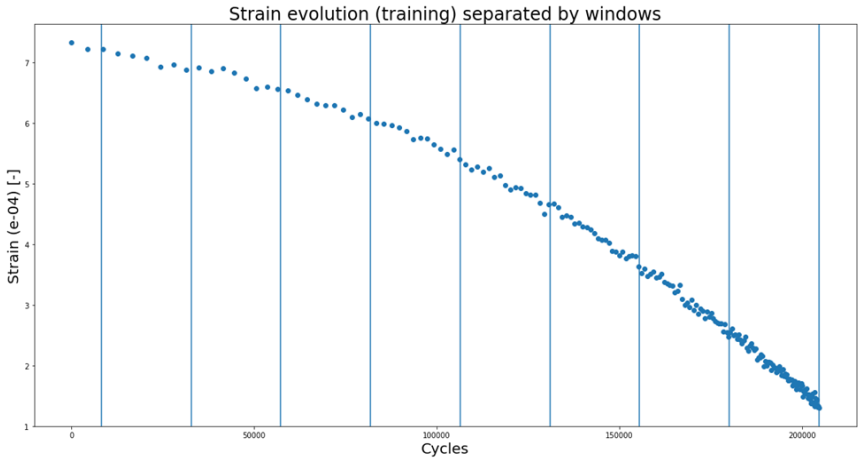


Figure 42: Complete generated sequence split by non-overlapping windows starting at the end of the complete sequence, training set

The window length was defined as the number of cycles that divide a complete sequence on 10 parts on average. Given that the average number of cycles elapsed per sequence was around 245353 cycles, the window length was determined to be $245353/10 = 24535$ cycles.

Regression models usually require a fixed size of input vectors, and both used models, the mean model (SVR) and the variance model (KNN), require it too. For this reason, the input vectors will be two dimensional, containing the mean and the slope extracted from each window. The mean is calculated as the average of all the values inside and the slope is extracted using a simple linear regression within the window. With an adequate fixed window length, the mean and slope should provide enough information about the Remaining Useful Life at that point, as shown in a similar approach by Khelif *et al.* [KHEL16]. The Remaining Useful Life of a window will be defined as the RUL of the last point in that window.

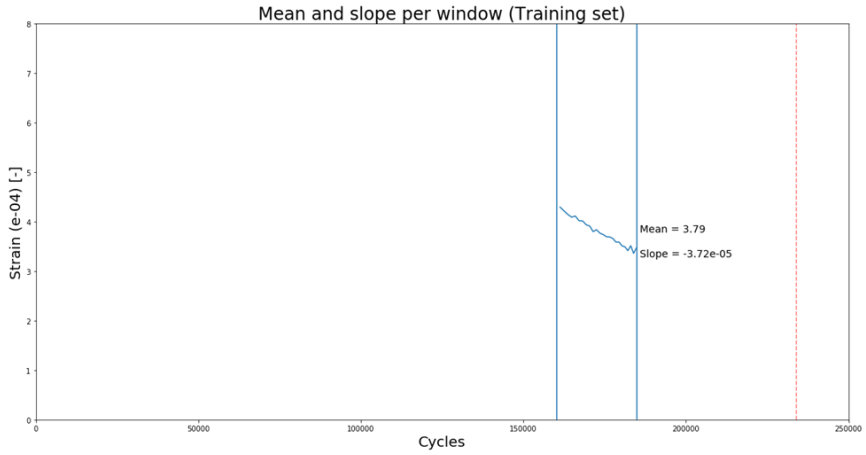


Figure 43: Extracted mean and slope of a window

The validation sets will try to emulate more realistic on-line operation by using a sliding window. Starting from the end, just like for the training set, the upper bound of the first window will be at the end of the complete sequence, and the lower bound the same defined number of cycles before (same window length as for training). Contrary to the training set, this time the upper bound will switch to the next point, and the lower bound the same defined number of cycles before (window length), and so on.

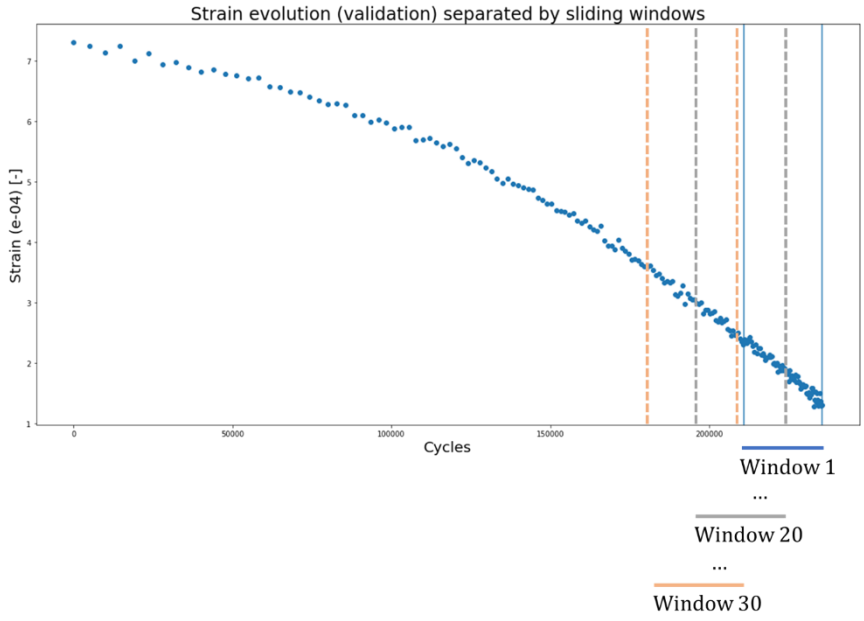


Figure 44: Complete generated sequence split by sliding windows for each point starting at the end of the complete sequence , validation set

Once both models have been explored and tuned, the pipeline involving both of them is tested with the test set. The test set will be created with the same sliding window technique as the validation set, but this time using the complete sequences from the real data, the actual Virkler's dataset.

3.3.2. RNN

A Recurrent Neural Network (RNN) is a class of neural network models capable of finding patterns in sequential data. A traditional neural network would assume that all the points are independent of each other, which isn't the case in many contexts. Taking as an example the case studied in the research, the strain-cycle datapoints are surely related to previous points in the strain evolution process, and it can be useful to consider these dependencies to predict the remaining useful life more accurately. The RNNs use the information from all the provided points of the sequence to build a "memory" that will store important information from all the points in the sequence. The RNNs have been used mostly for Natural Language Processing (NLP), and have been applied to various fields such as speech recognition, sentiment classification, video activity recognition, but also for time series data like clinical diagnosis datasets [CHE_18]. Before explaining more detailed the RNNs, a short review of traditional neural networks (feedforwards networks) will be presented.

3.3.2.1. Neural Networks Review

Neural networks are layered structures composed of layers with connected nodes. These nodes, also called neurons, are the most basic unit in neural networks. They take certain inputs, aggregate them and pass the result through an activation function, which generates an output. The simple case of two inputs (x_1 and x_2) and one output y is presented below. At the side of the illustration, the computation graph shows the inner calculations being performed on each stage.

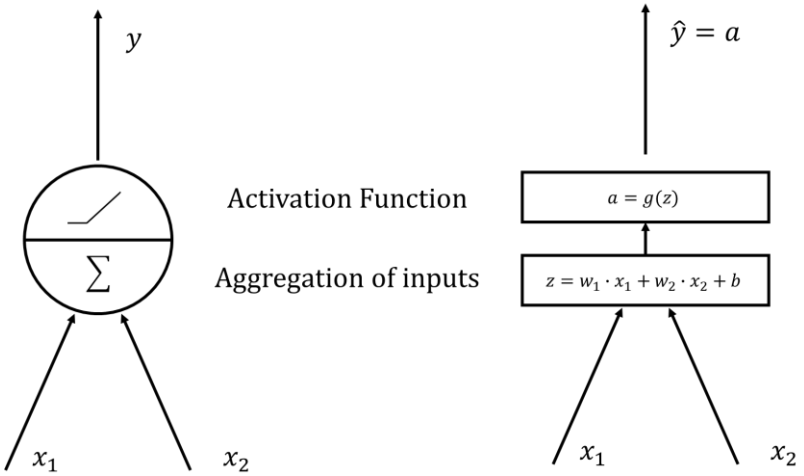


Figure 45: Basic Neuron and its corresponding computation graph

The inputs x_1 and x_2 are aggregated, weighed with weights w_1 and w_2, and the addition of a bias b, resulting in an intermediate variable z. Then, the variable z is passed through the activation function (ReLU in the shown neuron), outputting from the neuron the activation value $a = g(z)$. In this case the neuron is an output neuron, so the activation value will directly provide the prediction $\hat{y}$. With the prediction $\hat{y}$ the so called Forward Propagation would be completed. Neural networks architectures will normally use more

than one neuron to make a prediction. The inputs are connected to each neuron (these neurons form what is called a hidden layer), and each will provide an activation value. The activation values from all the neurons are aggregated in an output neuron, that performs the same calculations as the performed on the described neuron in Figure 45 to provide an output. This type of structures are called shallow neural networks (Figure 46 (a)), because they employ only one hidden layer of neurons. Deep neural networks will have several hidden layers (Figure 46 (b)).

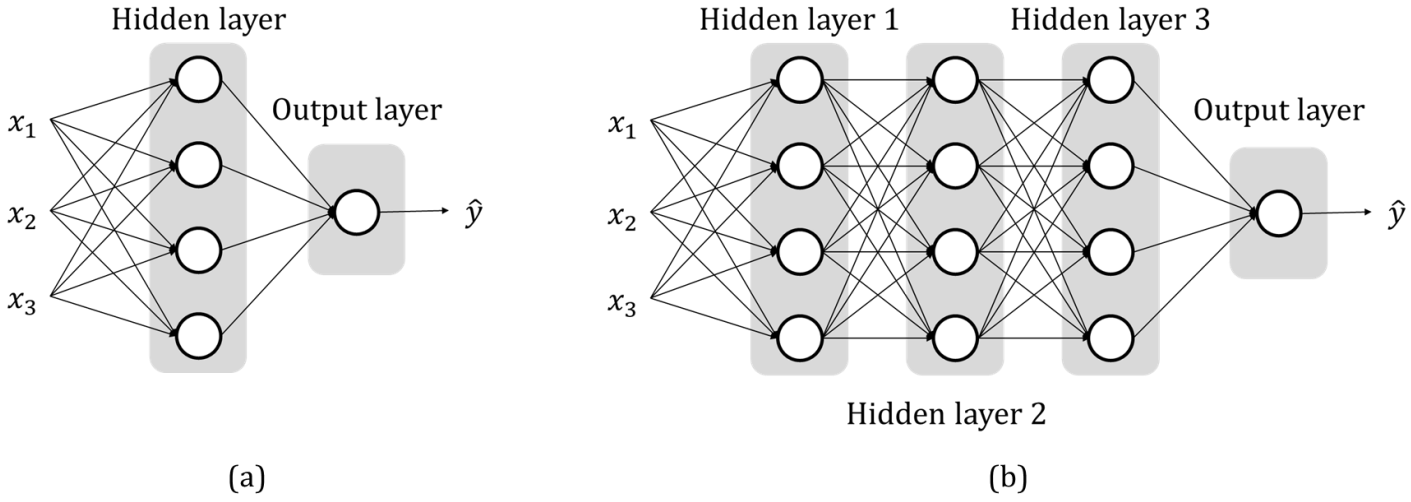


Figure 46: Shallow (a) and deep (b) neural network structures

Each neuron has their corresponding weights and biases, which will be updated through Gradient Descent. Once a prediction $\hat{y}$ is made, it is compared with the true value of the output y through the loss (or error) function $\mathcal{L}(\hat{y}, y)$. With the predictions in the inputted batch made, the cost function $J(W, b)$ can be calculated as the average of the loss function through the examples.

$$CF: J(W, b) = \frac{1}{m}\mathcal{L}(\hat{y}, y)$$

Where W is the matrix of weights and b the bias. Gradient Descent will try to find the values of W and b that minimize the value of the cost function $J(W, b)$. This function should be convex in order to facilitate the convergence of the optimization and this can be achieved through and adequate selection of loss function. The Gradient Descent algorithm updates the weights and biases by taking a step in the direction of the negative gradient in an attempt to minimize the loss function $J(W, b)$. For a certain weight and bias, the following equations would be used to update the parameters:

$$w = w - \alpha \frac{\partial J}{\partial w}$$

$$b = b - \alpha \frac{\partial J}{\partial b}$$

Where α is the learning rate. This hyperparameter will determine the size of the steps that will be taken. With the weights and biases of each neuron updated, the Back

Propagation process is completed, and the Forward Propagation will be performed again in what is called a new epoch.

3.3.2.2. Recurrent Neural Networks

As shown in the previous section, the conventional types of neural networks make the predictions $\hat{y}$ based on a fixed-sized input vector $\boldsymbol{x}$. RNNs not only allow employing multiple inputs, but they also take into consideration previous inputs, which in conventional neural networks are treated as independent. An RNN's structure can be derived from a regular neural network's structure. Taking a given time t, Figure 47 (a) shows a the simplest form of neural network, a neuron, that receives $\boldsymbol{x_t}$ as input and makes a prediction $\hat{y}_t$ based on it. For the following time $t+1$, $\boldsymbol{x_{t+1}}$ will be received by the same network with the same parameters, and the prediction $\hat{y}_{t+1}$ will be made. The same recurrent operation will be done for $t+2$ and the remaining timesteps. To take into account the information from previous (and potentially also future) times, the RNNs apart from making a prediction $\hat{y}_t$ also calculate a hidden state vector $\boldsymbol{a_t}$ at each time (Figure 47 (b)). This vector is then passed to the next time $t+1$, where prediction $\hat{y}_{t+1}$ will be made based on not just the input $\boldsymbol{x_{t+1}}$, but also based on the hidden state of the previous time, $\boldsymbol{a}_t$.

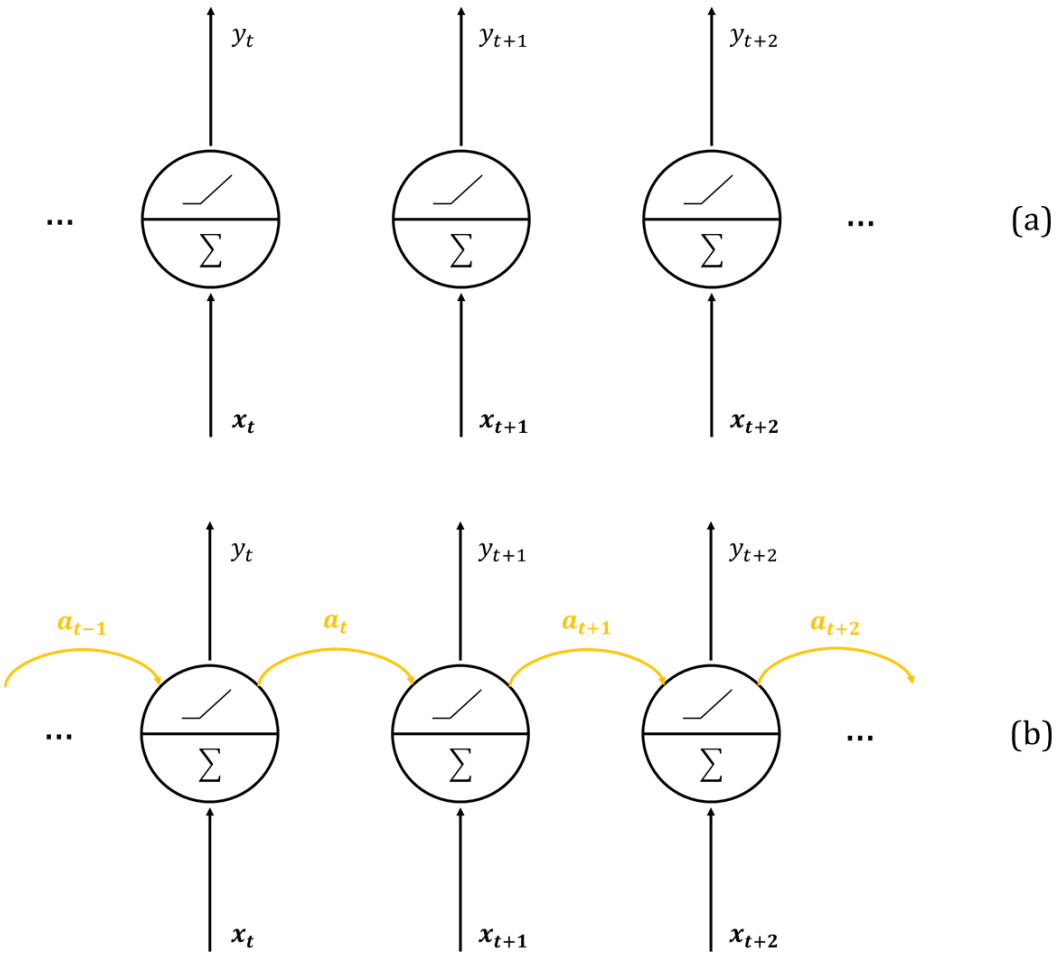


Figure 47: Conventional vs Recurrent Neuron Structure

$$\boldsymbol{a_{t+1}} = g_1(W_{aa} \cdot \boldsymbol{a_t} + W_{ax} \cdot \boldsymbol{x}_t + b_a)$$
$$\hat{y}_{t+1} = g_2(W_{ya} \cdot \boldsymbol{a_{t+1}} + b_y)$$

The shown common parameters from each timestep W_{aa}, W_{ax}, W_{ya}, b_a, b_y are updated using Back Propagation, as for conventional NNs, but now also updating not just the weights and biases for every time computed, but also the new weights and biases involved in the hidden state vectors $\boldsymbol{a_i}$. The denomination for this sort of Back Propagation is Back Propagation Through Time (BPTT). Of course, RNNs usually employ more than one neuron per "cell", but the operation remains the same. Structure-wise, RNNs don't necessarily have to make a prediction at every timestep. For example, in sentiment analysis RNNs take as input a sequence of words and output just one value: is the opinion from the text positive or negative. This leads to different types of RNNs according to their structures, number of inputs T_x and outputs T_y (Figure 48). The basic case (a) that has been treated until now is called "Many-to-many" architecture, where there the number of inputs and outputs are the same $T_x = T_y$. In the case that those inputs T_x only have one corresponding output, the architecture is called "Many-to-one" (b). There can also be the case where there are multiple inputs and outputs, but different number of them. In this case, the RNNs take the encoder-decoder structure (c), where first the inputs are used to calculate the last hidden state vector $\boldsymbol{a_{T_x}}$, which will encode the sequence. Then, the second part of the RNN will use this vector to make the predictions for the specified number of time steps. Lastly, the RNNs can also just take one input and make the following predictions based one it (more specifically on the output of the previous "cell"). This receives the name of "One-to-many" architecture.

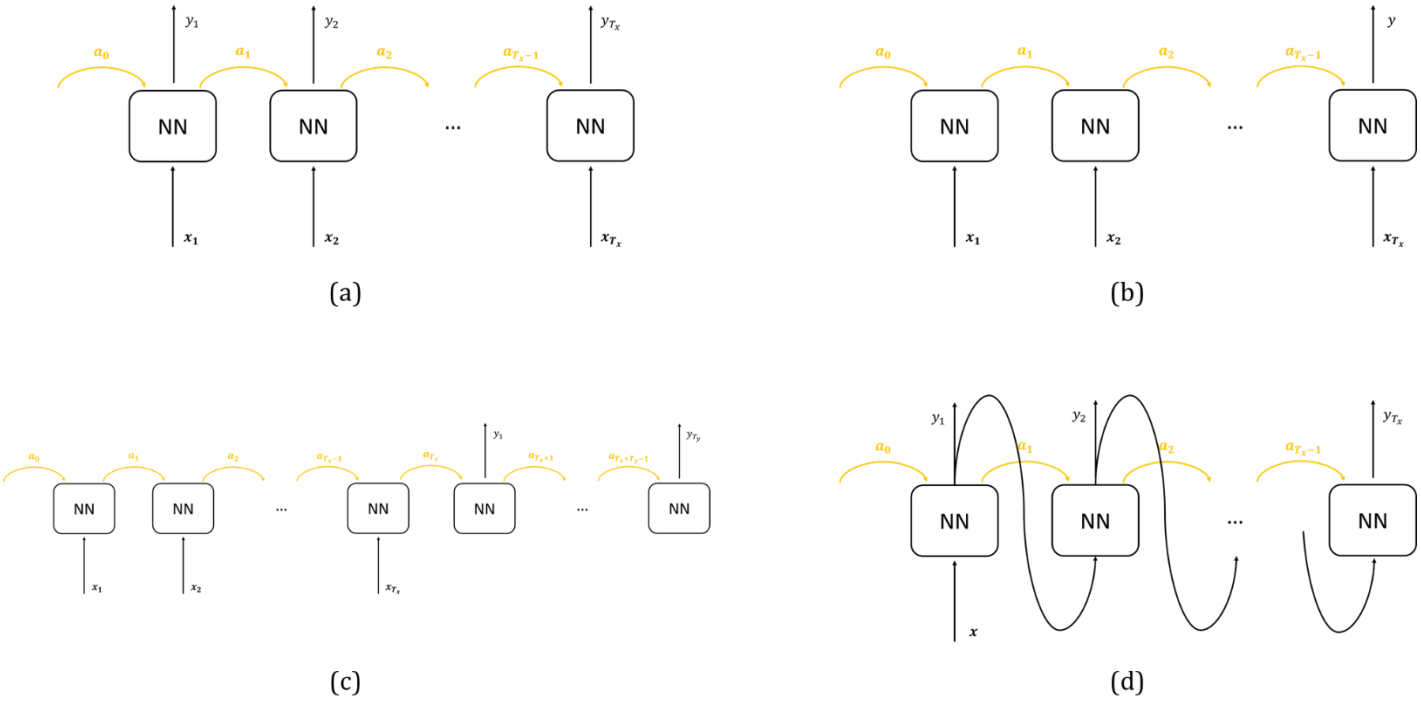


Figure 48: RNN architectures based on number of inputs and outputs

3.3.2.3. LSTMs

Despite being able to capture features from the introduced sequences, RNNs can have trouble with long term dependencies. Taking the "Many-to-many" structure on Figure 48 (a), the error obtained from the prediction at the end ($t = T_x$) barely affects the weight update at the beginning of the sequence $t = 1$. This type problem can also be found in very deep NNs, and happens due to vanishing (or exploding) gradients. The updating of the weights can become very small (or big) when employing very long sequential data.

This is the main shortcoming of traditional RNNs, but fortunately there have been new cell structures proposed to overcome this long term dependencies problems. The two main alternatives that have been employed are GRUs and LSTMs [HOCH97]. GRUs are a simpler version of LSTMs, which will be the ones explained in this section. LSTMs are complex cells that will substitute the regular NN cells displayed so far. To facilitate their understanding, the explanation will start from the interior of a simple NN (illustrated on Figure 49). The input $\boldsymbol{x}$, as well as the hidden state vector $\boldsymbol{a}$ are aggregated and passed through the activation function (tanh in this case).

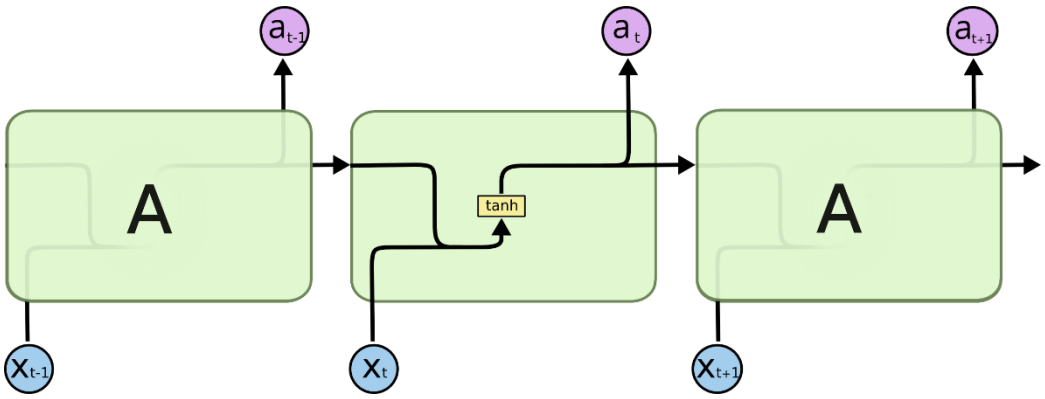


Figure 49: Standard RNN repeating cell [OLAH15], modified

LSTM's structure will be similar, but with two main differences: for a given time t, the aggregation of input $\boldsymbol{x_t}$ and hidden state $\boldsymbol{a_{t-1}}$ vector is passed through four activation functions instead of one, and a new vector called the cell state $\boldsymbol{c_t}$ is added.

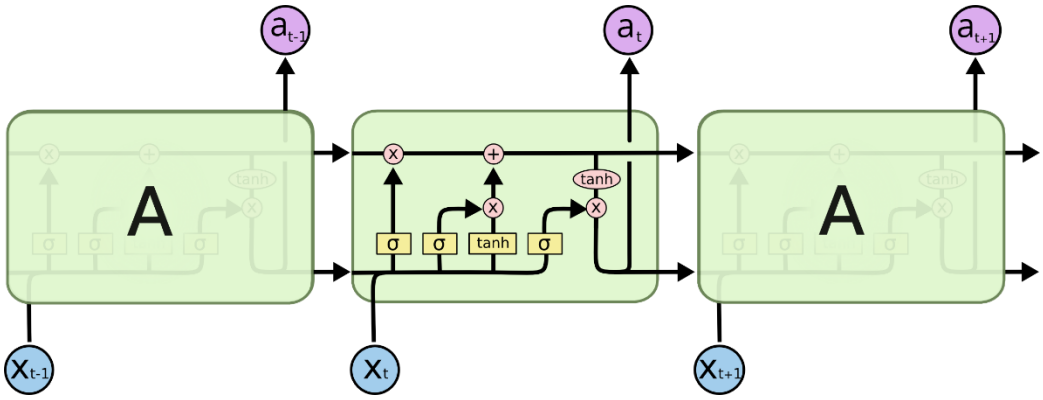


Figure 50: LSTM repeating cell [OLAH15], modified

The cell state vector $\boldsymbol{c_t}$ is the key addition of LSTMs, as it will carry the long time dependency information through all the timesteps. It goes through some minor linear interactions at each cell, and it can easily pass through a cell without being changed. The modification of the information contained in this cell state vectors is done thorough structures called gates. Gates allow (or not) information through with the use of a sigmoid layer, which will output values between vector 1 and 0 (1: letting everything through, 0: letting nothing though), and a point-wise multiplication with the cell state vector $\boldsymbol{c_t}$. An LSTM cell will have three gates like this.

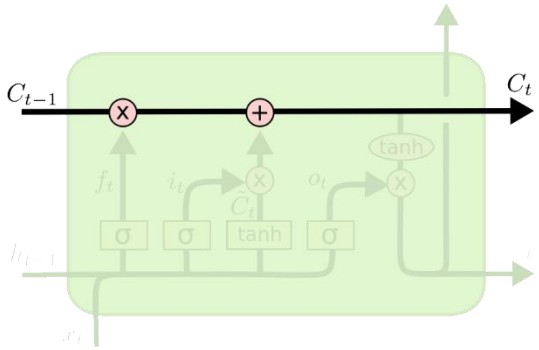
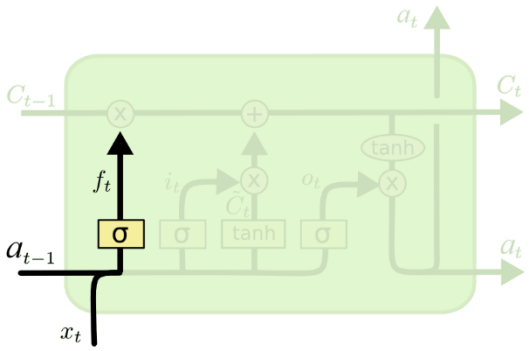


Figure 51: Cell state within an LSTM cell [OLAH15], modified

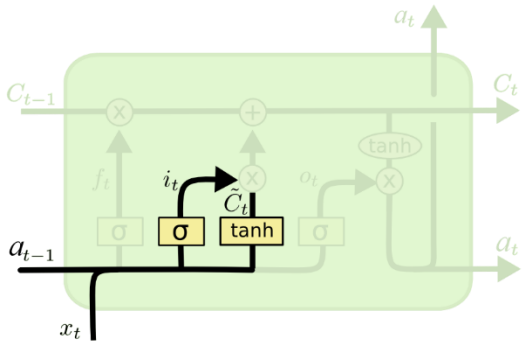
The flow of information through a cell will be described step by step to allow a better understanding of the gates and the cell state vector $\boldsymbol{c_t}$. Firstly, the LSTM cell decides what information to dismiss from the cell state (Figure 52). This is done through the "forget gate". After aggregating $\boldsymbol{a_{t-1}}$ and $\boldsymbol{x_t}$, it is passed through a sigmoid layer that will output values between vector 1 and 0 for each value in the cell state vector $\boldsymbol{c_{t-1}}$ (1: keep value intact, 0: dismiss value).



$$f_t = \sigma\left(W_f \cdot [a_{t-1}, x_t] \; + \; b_f\right)$$

Figure 52: Forget gate in a LSTM cell [OLAH15], modified

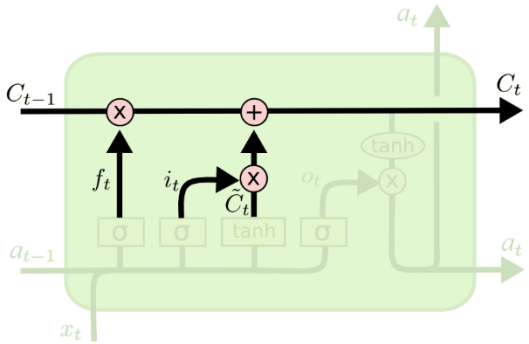
In the next step, it is decided what new information to include in the cell state. At first, the "input gate" will decide which values of the cell state vector to update through a sigmoid layer outputting i_t. Then, a vector with the candidate values $\boldsymbol{\tilde{c}_t}$ that can potentially be added to the cell state is created through a 'tanh' layer. This two vectors multiplied shape the cell state update.



$$i_t = \sigma\left(W_i \cdot [a_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [a_{t-1}, x_t] \; + \; b_C)$$

Figure 53: Input Gate and Vector of Candidate Values in a LSTM cell [OLAH15], modified

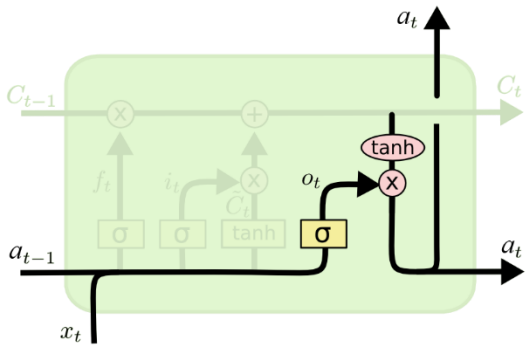
In the last step, the new cell state vector $\boldsymbol{c_t}$ is calculated. The old cell state $\boldsymbol{c_{t-1}}$ is multiplied by the "forget gate" f_t, dismissing the values that aren't useful. This is added to the vector of candidate values $\boldsymbol{\tilde{c}_t}$ multiplied by the "input gate" to include the new weighted information in the cell state.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 54: Cell state calculation within an LSTM cell (https://colah.github.io/posts/2015-08-Understanding-LSTMs/, modified)

With the cell state calculated, it is now time to compute the hidden state vector $\boldsymbol{a_t}$, which will also consider the cell state vector $\boldsymbol{c_t}$. First, an aggregation of the input and the hidden state vectors are passed through a sigmoid layer, deciding which parts of the cell state are going to be outputted. This is called the "output gate" o_t. Then, the cell state is passed through a 'tanh' layer (outputs values between -1 and 1) and multiplied by the "output gate" o_t to give the next hidden state vector $\boldsymbol{a_t}$.



$$o_t = \sigma\left(W_o\ [a_{t-1}, x_t]\ +\ b_o\right)$$

$$a_t = o_t * \tanh\left(C_t\right)$$

Figure 55: Output gate and hidden state calculation in a LSTM cell [OLAH15], modified

3.3.2.4. Mixture Density Network

Through the these structures, the RNN model can be able to make predictions about the RUL based on the strain evolution. If the output was just one node, this would output the predicted value for the RUL. In order to build a confidence interval based on a normal distribution, two nodes are utilized as output: one for the mean and another for the standard deviation of the RUL. Each one uses a different activation function:

Mean – **Sigmoid**: sigmoid activation functions are a common choice for output nodes where the desired output should lie between 0 and 1. In the studied case of the research, the output (RUL) will be normalized. Normalization or feature scaling scale the data between 0 and 1, making the activation function choice adequate.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

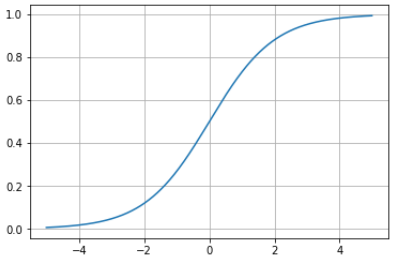


Figure 56: Sigmoid function

Standard deviation – **ELU+1**: in Mixture Density Networks, this function has proven to have particularly successful results. Any function that returns positive values could have also been an acceptable choice.

$$ELU(x) = \begin{cases} e^x - 1, & x < 0 \\ x, & x \geq 0 \end{cases}$$

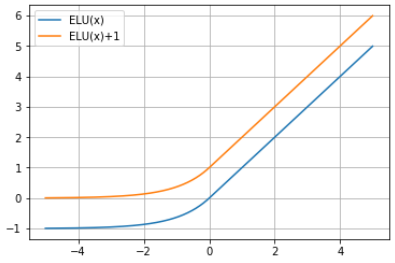


Figure 57: Exponential Linear Unit (ELU) and ELU+1 functions

Summarizing the most relevant shown concepts, the structure used to predict RUL based on the strain evolution is a Recurrent Neural Network (RNN) composed of LSTM cells that help capturing the long time dependencies. The two output nodes of the network will be the mean and the standard deviation of the distribution prediction.

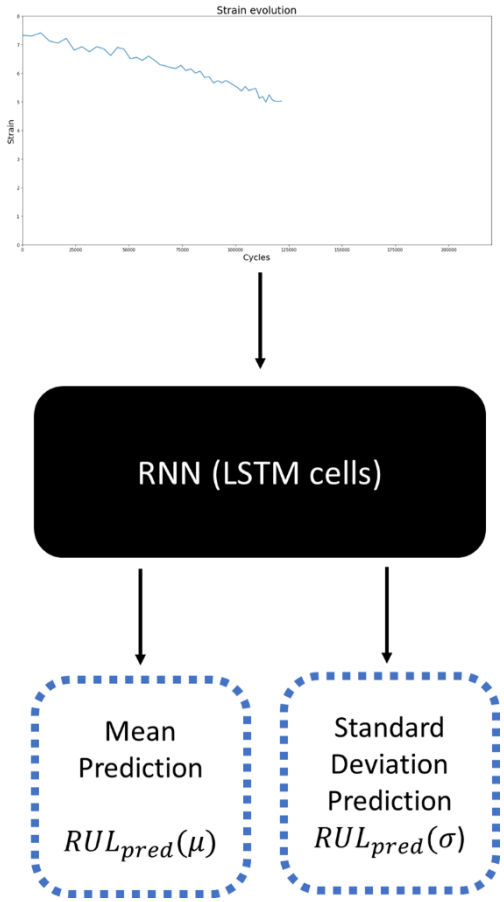


Figure 58: RNN distribution estimator

3.3.2.5. Training, validation and test sets

Unlike the SVR model, the RNNs can work with a variable number of inputs. This allows RNNs to consider all of the given points to make a prediction, not being restricted to those inside a fix-sized window. Thus, the training set, validation set and test set are developed accordingly. The goal is for the model to be able to predict with input sequences of any size, so form each sequence, gradually increasing partial sequences will be generated. The training and validation set are generated from the simulated values (90/10 split, the same as for the SVR model) and the test set from the real values from Virkler's dataset.

Each generated sequence have the same number of "measurements": 205. To create 10 partial sequences from each complete sequence (same number as in the SVR model's training data) each partial sequence should include around 20 new datapoints. Therefore, for each complete sequence, the initial partial sequence will contain the first 20 datapoints.

The next sequence will contain those 20 datapoints and the next 20 datapoints. The process is repeated until the end of the original complete sequence is reached.

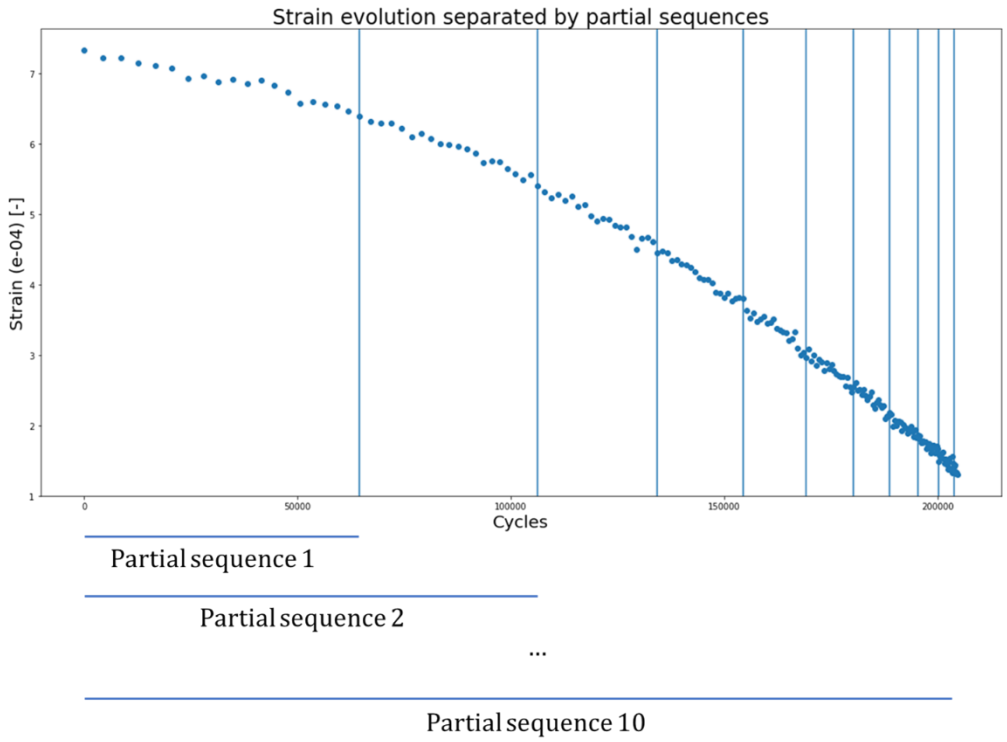


Figure 59: Complete sequence split in partial sequences with 20 additional datapoints per partial sequence

This approach has the potential downfall of the model learning only on the 10 sizes of partial sequences shown. To overcome this potential problem, the number of datapoints added to each partial sequence is randomized, picking the number of datapoints to add through random integers from the "discrete uniform" distribution of interval [1,40). On average, each partial sequence appends 20.5 new datapoints. This procedure will be the chosen one for the creation of the training set. For the validation and test set, the partial sequence will be selected so that the results are comparable to the obtained in the SVR model. In the SVR, a sliding window of length 245353 cycles with steps of one added datapoint per new window was used. The first partial sequence for each complete sequence will include all the points with less than 245353 cycles elapsed. The following windows will include one more point each until the end of the sequence is reached. The validation and the test set will be obtained from the generated (10%) and the real data respectively.

4. Results analysis

With both SVR and RNN model structures defined, it is now time to find the best parameters of the models by tuning the corresponding hyperparameters. Their obtention is illustrated, and then the predictions are be made. The performance of the models is measured and compared visually, but also using three metrics: R^2, MAE and MPDF. R^2 and MAE are calculated by comparing the mean predictions with the true values, while the MPDF takes into account the whole distribution prediction.

R^2 represents what percentage of the variation in the dependent variable (RUL in this case) is explained by the model. This metric takes values between negative infinity and one. R^2 gives an absolute idea about how well the model is predicting: if the R^2 of the model is close to zero, then it is performing poorly, without having to compare it to other R^2 values. It is calculated as one minus the mean squared error divided by the total variation of the dependent variable.

$$R^2 = 1 - \frac{\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2}{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$$

The second metric, the Mean Absolute Error (MAE) is the average of the mean absolute difference between the real values and the predictions. Unlike R^2, which is dimensionless, MAE has the unit of the dependent variable (cycles in the studied case). The alternative to this metric would have been the Root Mean Squared Error (RMSE), but MAE has more interpretability, as it gives directly the mean of the offsets, and also RMSE depends on the number of points it is calculated on, which isn't desired due to the different sizes of training, validation and test sets. The Mean Absolute Percentage Error (MAE) was also considered, but it presents very high values when the Y_i approaches zero, providing non-useful results.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|Y_i - \hat{Y}_i\right|$$

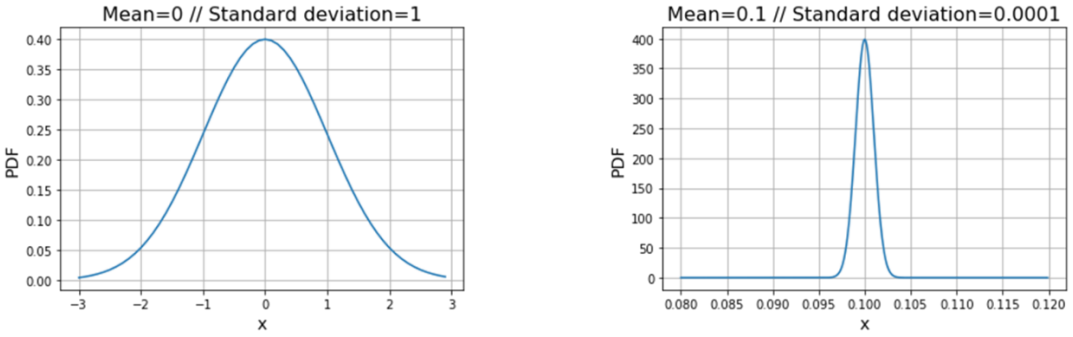
The past two presented metrics employ the mean predictions as prediction value $\hat{Y}_i$. The third metric, Mean Probability Density Function (MPDF), uses the whole distribution prediction to score the prediction. The distribution predictions follow a normal distribution. The probability density function of a certain point in the normal distribution can be calculated as:

$$PDF = \frac{1}{\sqrt{2\pi\sigma}} \cdot e^{-\frac{(y-\mu)^2}{2\sigma^2}}$$

Where y is the real value of the observation, and μ and σ^2 are the mean and the variance respectively. In the studied case, μ and σ^2 are the predictions and y is the true value of the RUL. The mean is calculated for all the points of the corresponding set, resulting in:

$$MPDF = \frac{1}{n}\sum_{i=1}^{n} PDF = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{\sqrt{2\pi\hat{\sigma}_i}} \cdot e^{-\frac{(y_i-\hat{\mu}_i)^2}{2\hat{\sigma}_i^2}}$$

For a given distribution, the PDF values increase the closer the true value is to the mean prediction. The values of the PDF in a normal distribution of mean $\mu = 0$ and standard deviation $\sigma = 1$ are displayed on (a). In the studied case, the PDF was calculated on the normalized values of the RUL ($y \in [0,1]$), which mean that the distribution predictions have standard deviations way smaller than $\sigma = 1$, resulting in much higher PDF values (b).



(a) (b)

Figure 60: Probability Density Function of a standard normal distribution (a) vs a sample normal distribution prediction (b)

4.1. SVR

The process to obtain the SVR's parameters is first described by visually assessing the performance and through R^2. After the SVR model (for the mean prediction) has been completely defined, the variance KNN model is then built. After both models are defined, the results on the training, validation and test sets are obtained and compared visually and also through the three previously explained metrics: R^2, MAE and MPDF.

4.1.1. Parameters

The SVR model was implemented through the Scikit-Learn library, commonly used for regression and classification algorithms, including Support Vector Machines, Gradient Boosting, Random Forests, etc. This library simplifies the designing and tuning of the model, allowing for many more iterations to be done. The model implemented is an ε-SVR model, where the margin is defined through ε. The SVR model has the main following parameters to be edited. Their default values are shown in Table 4.

Table 4: Default values for the SVR parameters (Scikit-Learn)

Parameter	Default
Kernel	RBF (also Polynomial, Sigmoid)
Degree (only for Polynomial kernel)	3
γ	1/n_features (=1/2)
r (for Polynomial and Sigmoid kernels)	0.0
C	1.0
ε	0.1

The first decision was the most adequate Kernel for the dataset. The three proposed kernels (RBF, Polynomial and Sigmoid) were compared with their default values in order to select the most promising kernel, and tune the parameters later.

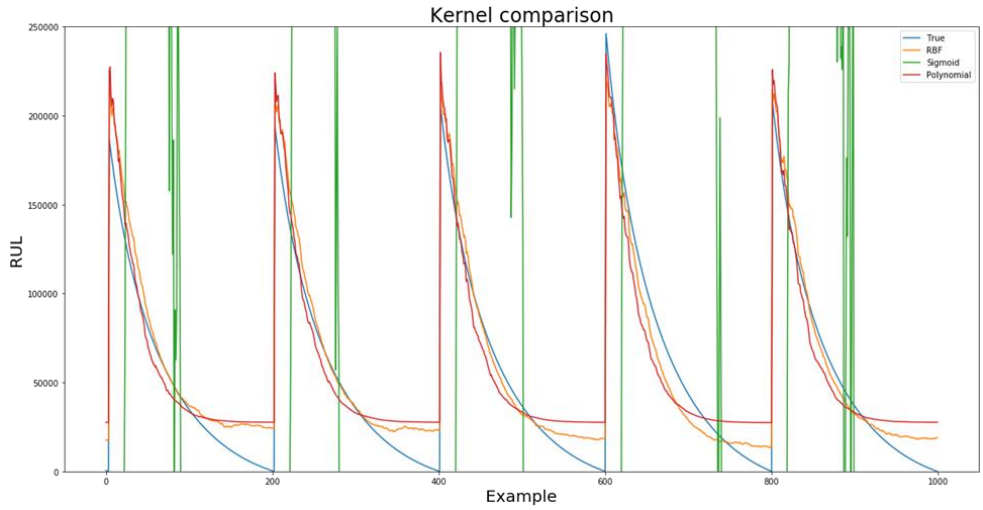


Figure 61: RBF, Sigmoid and Polynomial kernels comparison through predictions versus the true RUL values with default values on Scikit-Learn

The Sigmoid kernel doesn't predict close to the real values at all. From the other two candidates, the Polynomial kernel doesn't capture the end of the life of each sequence, stagnating towards it. Therefore, the selected kernel for the SVR model was RBF, the most common one. Once chosen the type of kernel, now the kernel's parameter γ, as well as the parameters corresponding to the cost function to minimize (C and ε) will be explored. The first parameter to explore is ε, which defines the margin of tolerance where no penalty is given to the errors. Choosing a value of ε too small can lead to a loss of generalization potential (overfitting). The values of ε explored were initially the logarithmic grid $\{0.01, 0.1, 1\}$ and then a finer tuning was performed. As seen in Figure 62, lower values of ε

deliver much better results, providing an $R^2 = 0.989$ on the validation set for $\varepsilon = 0.01$. Lower values might make the model too complex, and thus $\varepsilon = 0.01$ was selected.

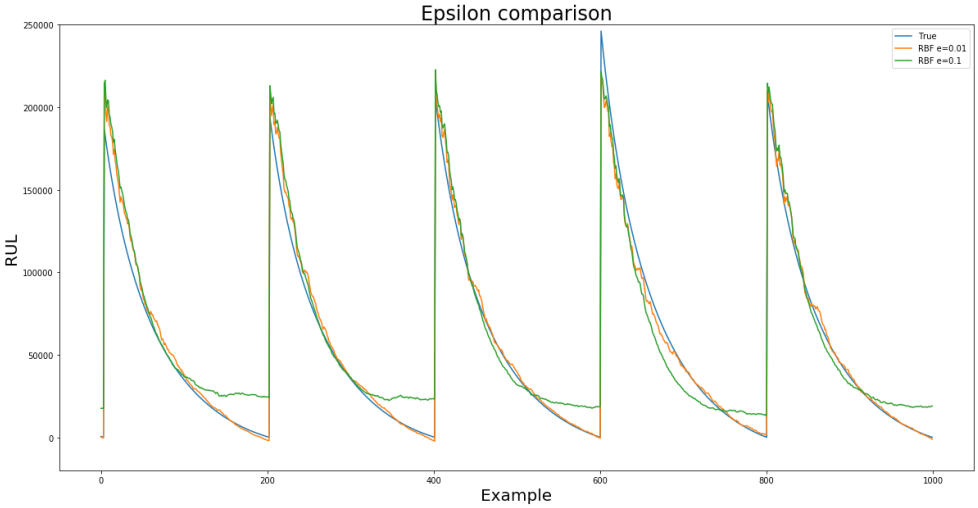


Figure 62: Epsilon values comparison through predictions versus the true RUL values

Once defined the tolerance margin, the penalization given to the errors of the points that fall outside of it is defined by C. Based on the suggestions of the developers at Scikit-Learn [SCIK19] a logarithmic grid from 1 and 1000 was explored. The graph this time doesn't reflect a great change in the performance, but the selected value for C was 1000 due to it providing a better R^2. The R^2 on the validation set was 0.990.

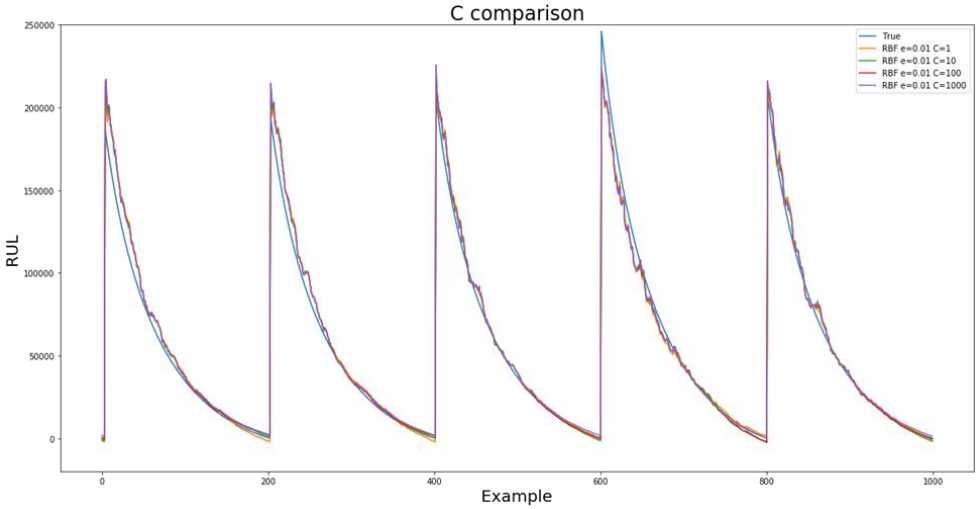


Figure 63: C values comparison through predictions versus the true RUL values

Finally, the last parameter to tune in the model was the RBF kernel's parameter γ, which defines how the influence reach of a training point. It is inversely related to the radius of influence of samples selected by the model as support vectors, which means that smaller values of γ reach further. Starting from the default proposed value based on the

number of features $\gamma = \frac{1}{n_{features}} = 0.5$, the explored values were $\{0.05, 0.5, 1\}$. The default value, $\gamma = 0.5$ performed the best, so this value was maintained.

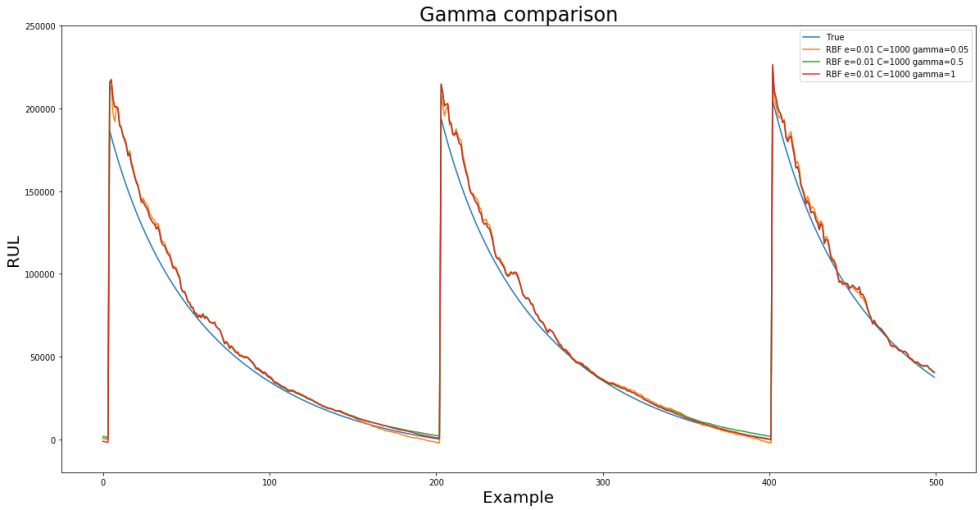


Figure 64: Gamma values comparison through predictions versus the true RUL values

The selected type of kernel, with the corresponding kernel and cost function parameters are displayed on Table 5.

Table 5: Selected values for the SVR parameters

Parameter	Selection
Kernel	RBF
γ	1/n_features (=1/2)
C	1000
ε	0.01

With the model to predict the mean of the confidence interval defined, the model to predict the variance was then designed. The chosen model, KNN, should capture the error of similar profiles to the one being studied in order to obtain the standard deviation. The procedure went as follows:

1. Predict the mean with the SVR model
2. Calculate the error (MSE) of the mean predictions versus the real values
3. Train the "Variance" model with the strain sequences as input and the error from the previous step as output
4. Make the error predictions (will be used as variance of the distribution)

The error from the SVR model was bigger at earlier stages of the lifecycle of each sequence studied, which is compatible with the intuition that the closer the sequence is to ending, the more precisely the model should predict the RUL. When designing the KNN the model to male these predictions, the only parameter to tune are the number of

neighbor point chosen to base the prediction on (interpolation). The model was trained on 4278 partial sequences, so values from 10 (0.24%) to 500 (11.68%) neighbors were analyzed. Through cross validation (K-Fold with 5 splits), the number of neighbors that provided the best R^2 were 60, 70, and very closely 120. 120 was choosing due to it favoring the robustness of the model. This delivered an $R^2 = 0.198$ in the validation set assigned for the variance model.

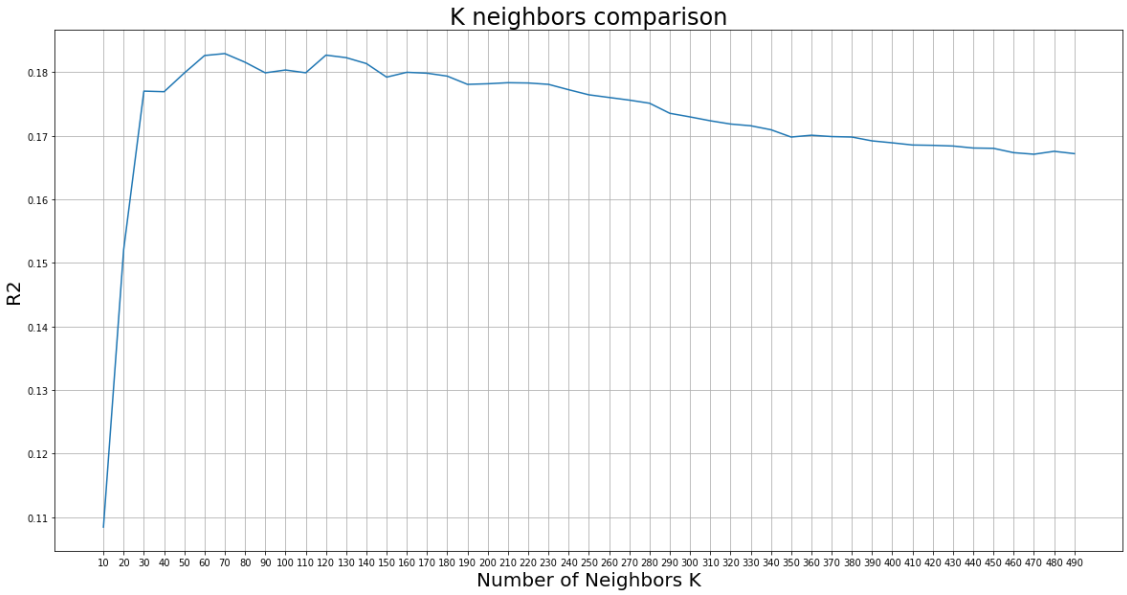


Figure 65: R2 results of the analyzed KNN through 5 splits in Cross Validation

To plot the RUL confidence interval predictions, the selected graph compares the number of cycles elapsed at the analyzed windows against the last cycle of the complete sequence it belongs to. The predictions made are for the RUL, so the last cycle predictions are calculated by adding the number of cycles elapsed in the windows being analyzed and the predicted RUL. A 99.7% confidence interval is shown (normal distribution up to 3 sigma) for the last cycle predictions. The predictions using both models were made on the variance validation set, obtaining profiles similar to the presented in Figure 66.

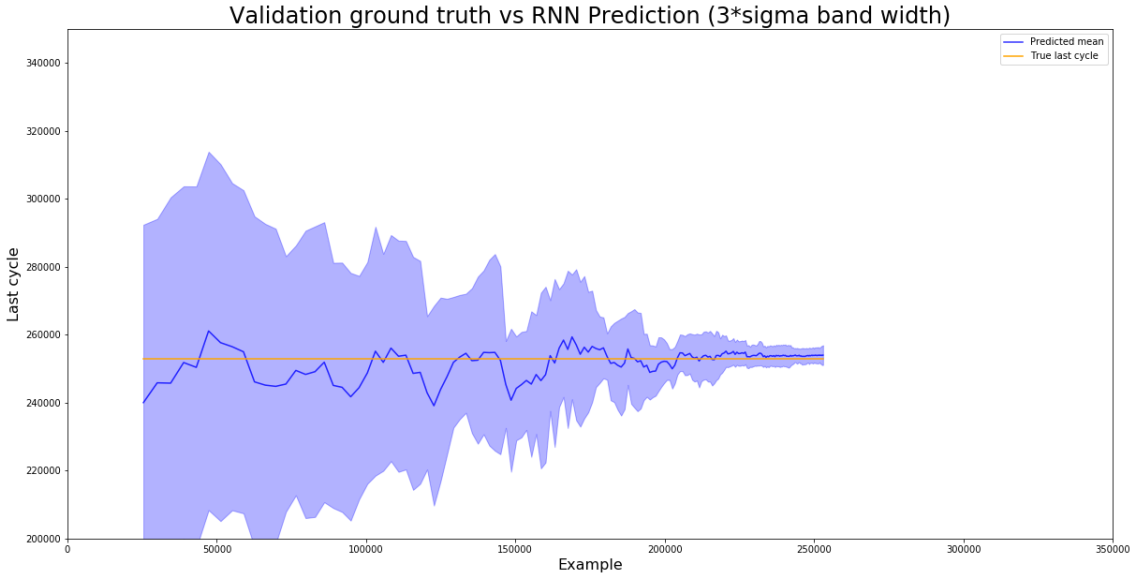


Figure 66: Sample of last cycle prediction by the SVR-KNN model on the variance validation set (K=120)

The behavior of the confidence interval was not smooth, adding too much complexity to the model. This suggests that a higher number of neighbors might be more appropriate. Higher number of neighbors were analyzed again on the training set to make a decision.

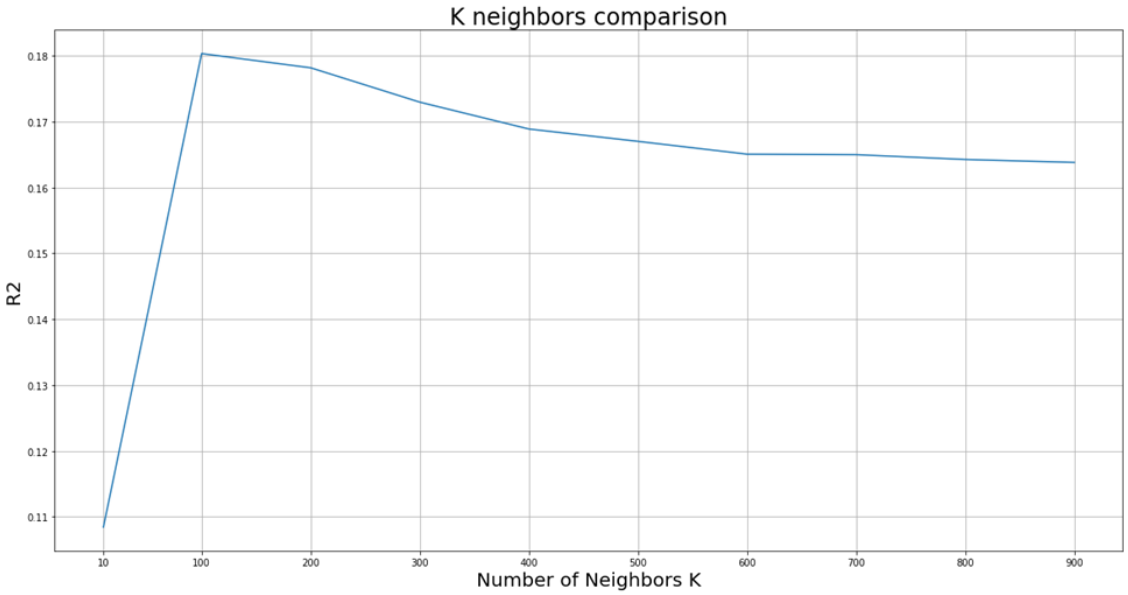


Figure 67: R2 results of the analyzed greater KNN through 5 splits in Cross Validation

There is no significant change in the performance when increasing the number of neighbors up to very high values. Thus, the selection of the number of neighbors was made by checking if an acceptable smoothness was reached. The selected number of neighbors with this criteria was 500.

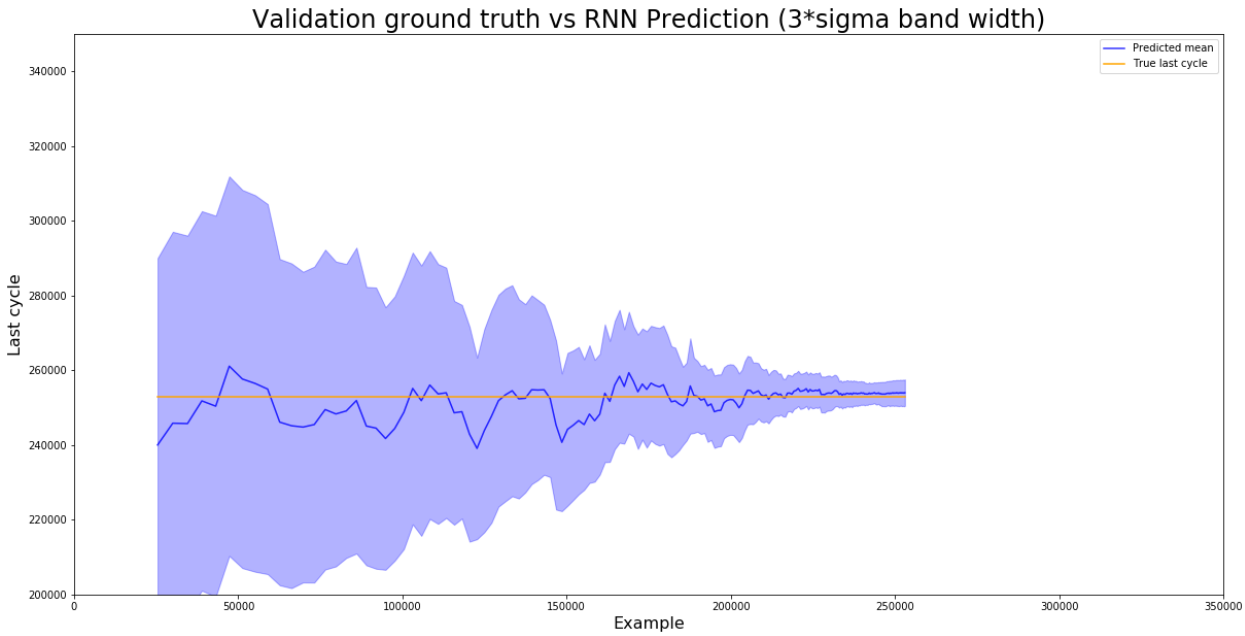


Figure 68: Sample of last cycle prediction by the SVR-KNN model on the variance validation set (K=500)

4.1.2. Results

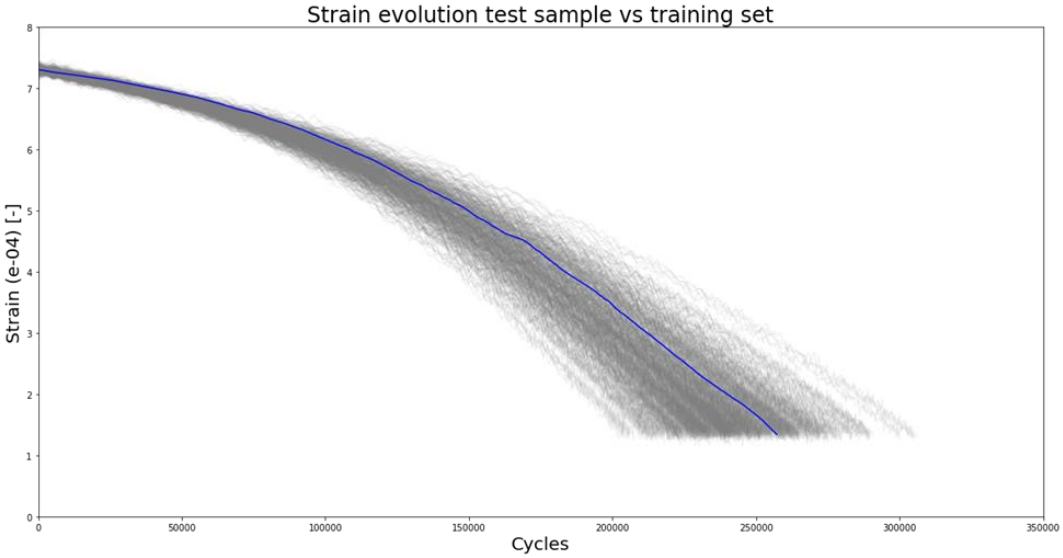
The results of the built model on the different sets are presented in Table 6. Unlike for the RNN model, the SVR model has two training sets and two validation sets. Each of the training sets was employed to train one of the two models (SVR-mean and KNN-variance). Due to the way the data was generated, the validation sets show a better performance than the training set. The validation sets were obtained with a sliding window for each point starting at the end of the sequences, and the sequences have more density of points towards the end (Figure 44). This results in more windows towards the end in the validation sets. The predictions at the end are better than at the beginning, and therefore the mean scores (for the three metrics) are better overall. The test set is obtained in the same way, so the comparisons will be made for the test set versus the validation sets. It should also be noted that there is a difference on the performance with respect to the validation sets, but this is inevitable given that the simulated values (training and validation sets) are an approximation of the real values, and the performance will be closer to the validation set the better the approximation is but the data will never be exactly the same.

The first metric on the test set delivers a score of $R^2 = 0.966$, which suggests that the SVR mean model is approximating the observations of RUL well. As for the MAE, the model performs worse than the validation set, but the value of 5628.5 *cycles* in offset on average can still be acceptable. Lastly, the MPDF shows a lower result than the validation sets, which indicates that the model isn't capturing the true RUL as good as in the validation sets.

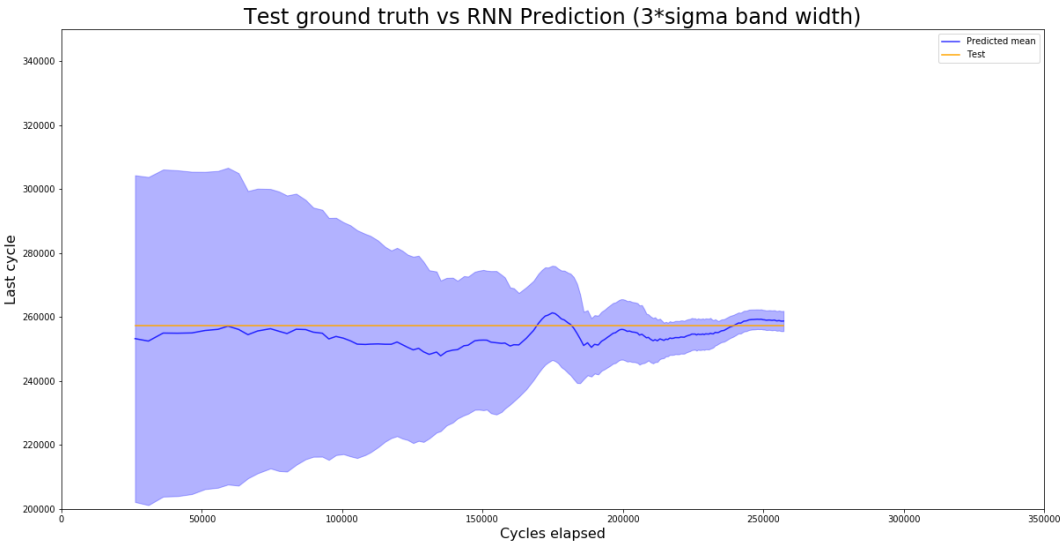
Table 6: Scores of the SVR model on the different sets (training 1&2, validation 1&2 and test)

Metric	Training 1	Training 2	Validation 1	Validation 2	Test
R^2	0.981	0.980	0.990	0.988	0.966
MAE [cycles]	6253.2	6441.4	3328.5	3555.3	5628.5
MPDF	18.7	18.5	33.4	33.3	17.1

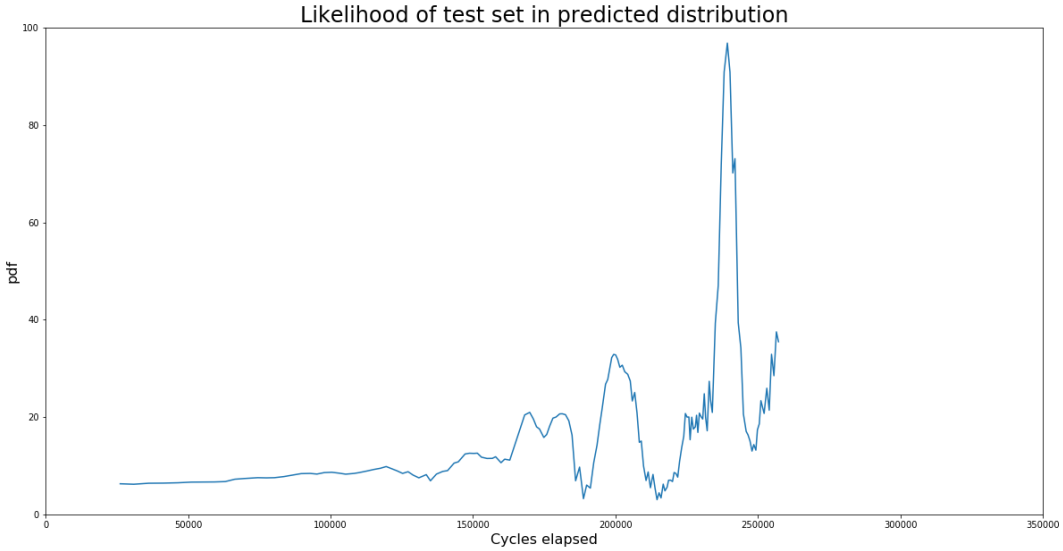
After seeing the scores on the three metrics the results are inconclusive, so the performance of the model had to be analyzed visually. Three graphs will be employed to check the performance of the model on the test data. The first graph shows in blue the sample from the test set being illustrated, with the training set behind in grey. This graph will show how similar that particular test sample is with respect to the sequences the models have been built on. The second graph has already been shown in the previous section for the validation set. This graph displays the number of cycles elapsed on a certain window versus the predicted last cycle, which is calculated as the number of cycles elapsed plus the predicted RUL. A 99.7% confidence interval is shown (normal distribution up to 3 sigma) for the prediction of the last cycle. The last graph shows the likelihood of the actual point from the test set in the predicted distribution. The likelihood is calculated with the probability density function through the predicted mean and variance. The best result obtained on the test set looks as following:



(a)



(b)

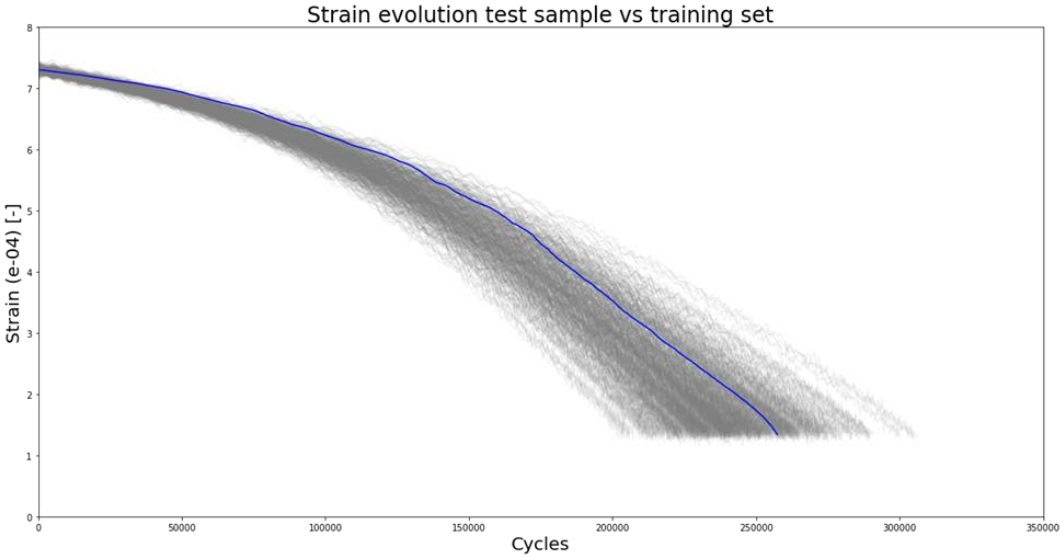


(c)

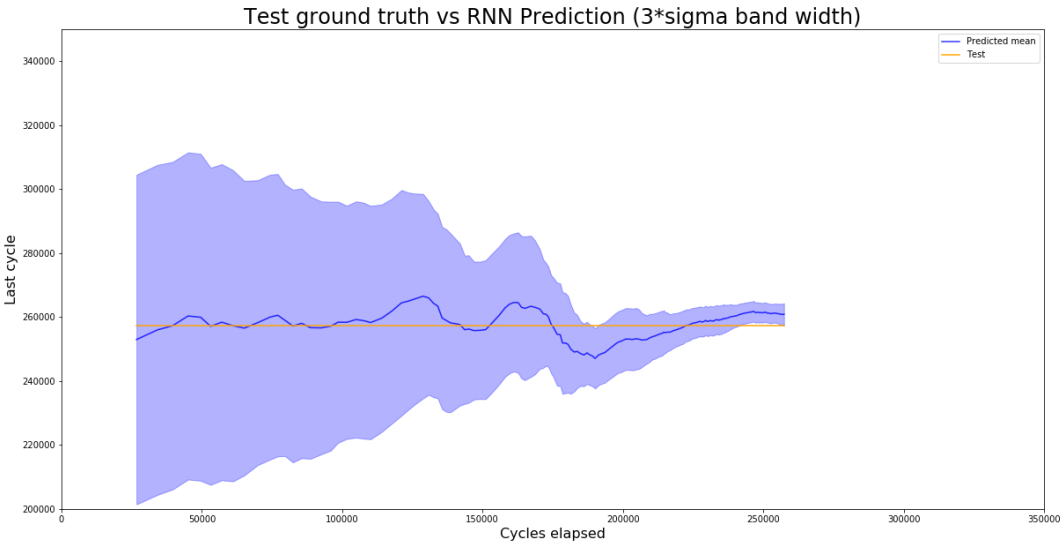
Figure 69: Evaluation graphs on best SVR prediction sample of test set

The result is considered positive for the following reasons. First, because the strain evolution profile is similar to those used in the training set (Figure 69 (a)). The degradation doesn't accelerate at exactly the same rates as the sequences in the training set (at the beginning the strain decreases more slowly, as well as towards the end at around 23000 cycles), but they are similar enough. Second, the real last cycle is always included in the predicted 99.7% confidence interval (Figure 69 (b)), which means that the model is not just making good predictions of this value, but also considering potential deviations from the predictions correctly. Lastly, the likelihood (calculated through probability density function) would ideally grow as the sequence advances, which would mean that the model is predicting the RUL more precisely as degradation advances and more theoretically relevant information is available. In the shown sample, the likelihood increases as more cycles pass (Figure 69 (c)), which is very good news.

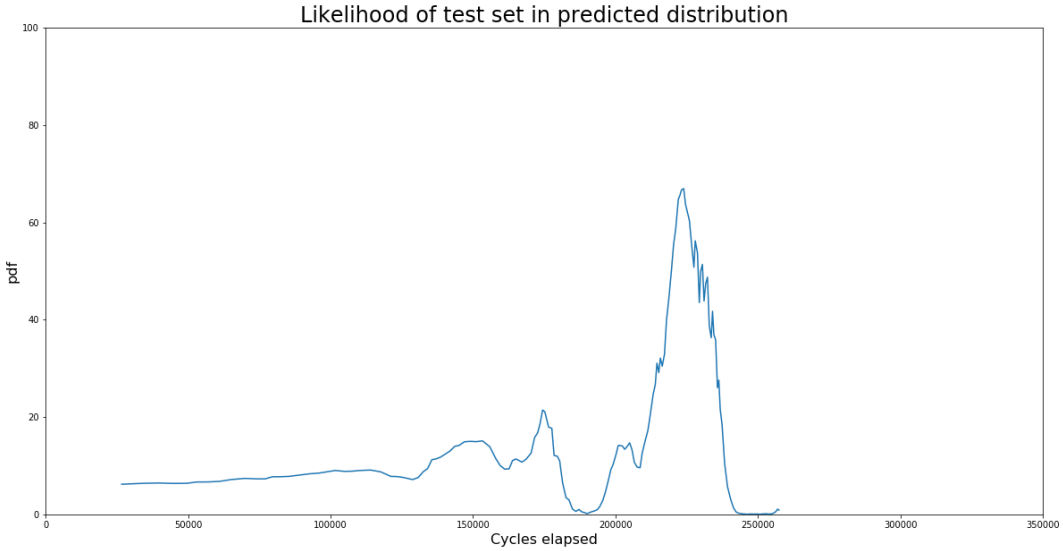
Unfortunately, not all the predictions are as good as the previously shown. A common result of the predictions can be seen in Figure 70. In this case, the strain decreases slower than the sequences considered at the beginning, then faster, then slower again, and at the very end it decreases more rapidly than the training set (Figure 70(a)). This seems to relate to the areas where the real value of the last cycle falls out of the confidence interval (Figure 70(b)). More specifically, the segments where the strain decreases more rapidly than usual (at around 190000 cycles and at the end). Consequently, in these sections the likelihood drops to very low values, truncating the increasing tendency.



(a)



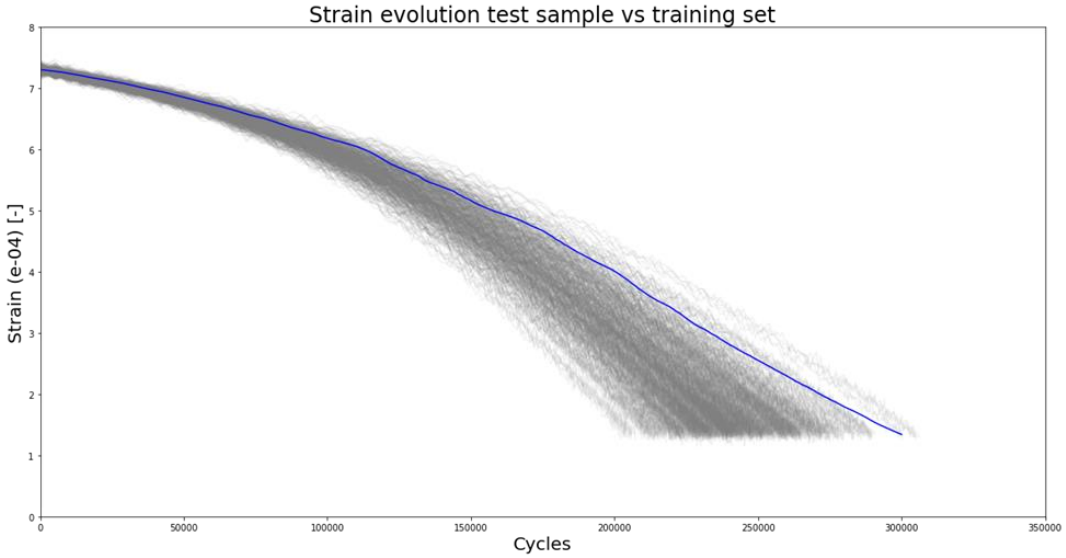
(b)



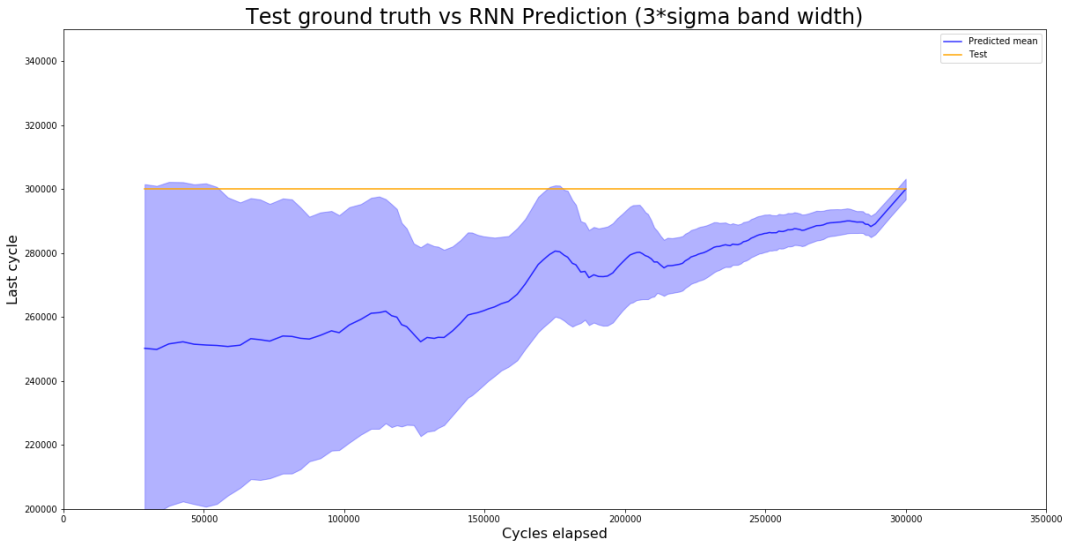
(c)

Figure 70: Evaluation graphs on standard SVR prediction sample of test set

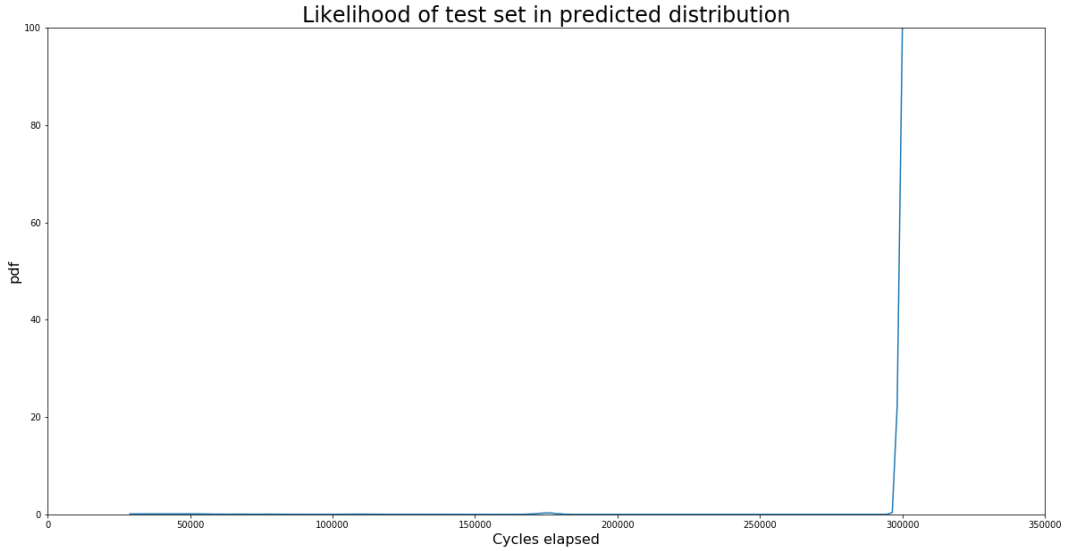
Finally, the worst result occurs when the sequence evolves completely different than the training set. The model predicts systematically values that are way off the real RUL (or last cycle) due to the sequence taking a path that hadn't been considered. This abnormal strain growth (derivates from the crack growth) seem to set the limit for the model and where it can make acceptable predictions. Even through the mean and slope are being considered for each window, values of the slope delivered by sequences like this one don't seem to have as much impact on the predictions as it should have.



(a)



(b)



(c)

Figure 71: Evaluation graphs on standard SVR prediction sample of test set

4.2. RNN

The obtention of the RNN's parameters through the tuning of the corresponding hyperparameters is presented. The used cost function, the optimization algorithm are also mentioned. After having the network defined, validation results are analyzed. The analysis compares the three metrics (R^2, MAE and MPDF), as well as a visual assessment of the results. Then, the performance on training, validation and test sets are compared in order to draw the conclusions of the model. The results were not as successful as in the SVR model, so the alternative of training on the real values from Virkler's dataset is presented with the corresponding results.

4.2.1. Hyperparameters

Before optimizing the parameters, a cost function had to be defined. The selected cost function was as the average of the negative log likelihood for the true point in the predicted distribution. The negative log likelihood of certain point can be computed as:

$$-\log(PDF) = -\log\left(\frac{1}{\sqrt{2\pi\sigma}} \cdot e^{-\frac{(y-\mu)^2}{2\sigma^2}}\right)$$

This way the RNN will try to predict a mean as close as possible to the real value, but also reducing the variance to increase the PDF. The optimization algorithm employed was Gradient Descent. More specifically, with the Adam optimizer, that has proven to provide more effective optimizations than the other alternatives. In the training, the RNN can be fed different amounts of values before Back Propagation (through Gradient Descent) takes place. If the whole training set is fed, the name it receives is Batch Gradient Descent. The result for Batch Gradient Descent for our model are shown in Figure 72(a). Another option is feeding the RNN mini-batches of the training set, receiving the name of Mini-Batch Gradient Descent. Each time a mini-batch is fed, Back Propagation (through Gradient Descent occurs) and the parameters are updated. This has the advantage of faster improvement of the algorithm. This can be seen in the much faster decrease in the cost function Figure 72 (b). The selected number of point for each batch were 256 (recommended to be multiple of 2)

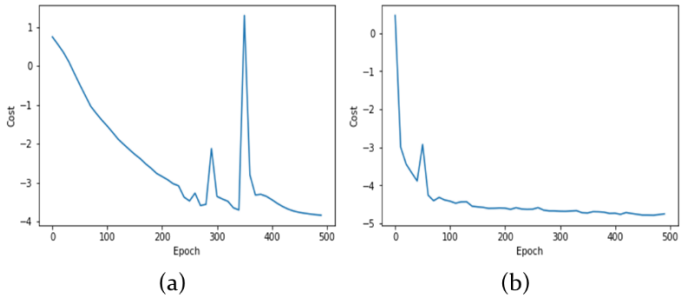


Figure 72: Batch Gradient Descent (a) versus Mini-Batch Gradient Descent (b) with Adam optimization on the studied RNN model, 500 epochs

The model was evaluated using a learning rate where the model increased the performance faster, but also trying to minimize the variation in the performance from one epoch to another (spikes). This was achieved through a learning rate $\alpha = 0.005$. As illustrated on Figure 72, the training was performed during 500 epochs. The other Adam optimization hyperparameters controlling the exponential decay rates of the gradients were set at the commonly used values of $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The only hidden layer was composed of LSTM cells with an output dimension of 4. This value presented good results on the predictions metrics (R^2, MAE and MPDF).

The distribution prediction of the described RNN model on the validation set showed very accurate results, approximating the mean to the true value the more of the original complete sequence it was provided. The variance (confidence interval shown of 99.7%) also decreased gradually and uniformly, providing high and growing likelihood values. A sample from the validation set is displayed on Figure 73. The confidence interval becomes very narrow, maximizing the likelihood, but this raises a warning for the test predictions (the model might not be able to generalize well).

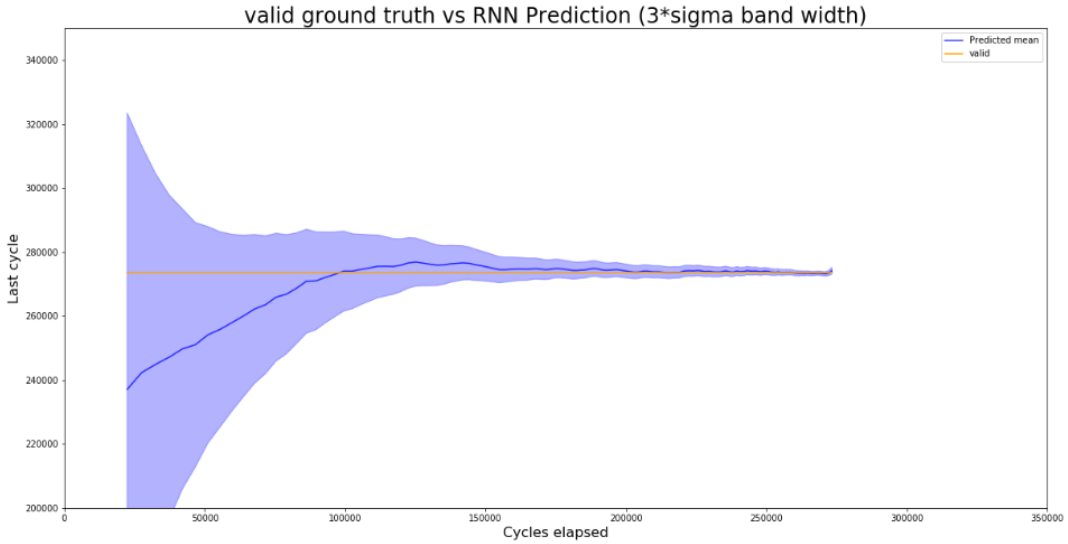


Figure 73: Sample of last cycle prediction by the RNN model on the validation set

4.2.2. Results

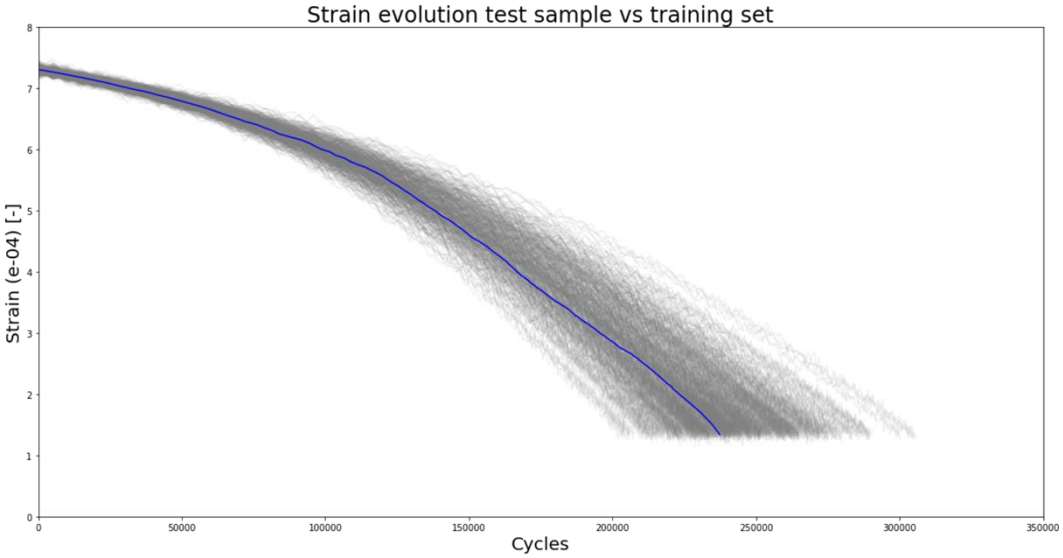
With the structure, learning process and hyperparameters defined, the results on the different sets (training, validation and test) are displayed on Table 7. The training and validation scores are very similar on the three metrics. As seen in Figure 73, the model approximates very accurately the RUL on the validation set with high and increasing values of the pdf per sequence, which is reflected on the three metrics being very similar for the training and validation set. As for the test set, each one of the three metrics gives a different point of view on the performance. The $R^2 = 0.945$ suggests that the model is doing a good job at estimating the mean for the RUL. As explained for the SVR model's results, there is a difference with respect to the training and validation set, but this is inevitable given that the simulated values (training and validation sets) are an approximation of the real values, and the performance will be closer to the validation set the better the approximation is but the data will never be exactly the same. Nevertheless, $R^2 = 0.945$ is a positive result. As for

the MAE, the validation's score is very different from the test's score. The test set' mean predictions present over four times as much offset as the validation set's mean predictions. This would suggest that the model is performing significantly worse on the test set than on the validation set (Figure 73). Lastly, the MPDF score points out that the test set's distribution predictions aren't capturing the true behavior of the test set. The cost function of the RNN was based on the PDF values, so differences in the MPDF scores will be increased, and indeed point out that the RNN isn't predicting the test set accurately enough.

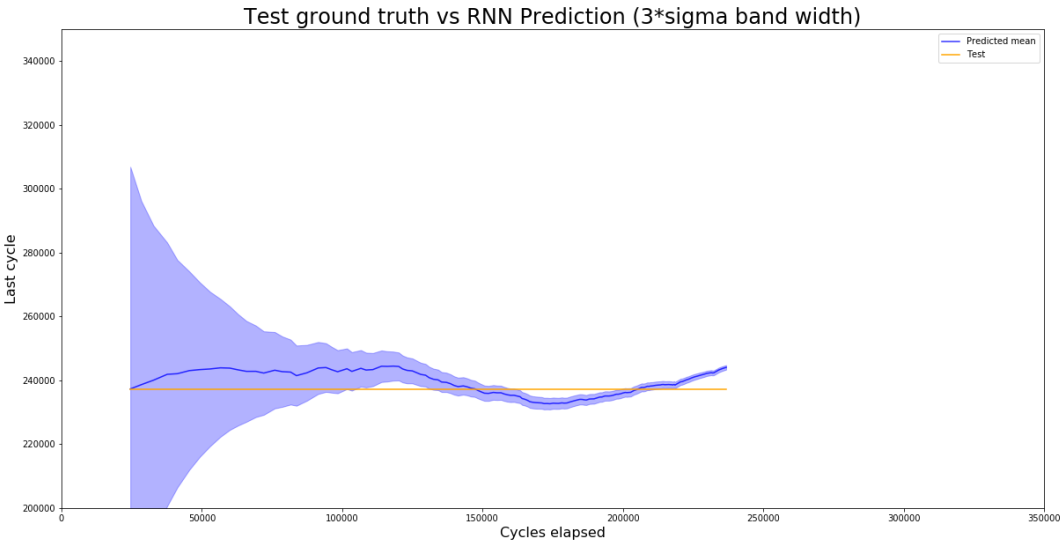
Table 7: Scores of the RNN model on the different sets (training, validation and test)

Metric	Training	Validation	Test
R^2	0.989	0.995	0.945
MAE [cycles]	1498.7	1285.3	8718.0
MPDF	197.1	191.6	15.4

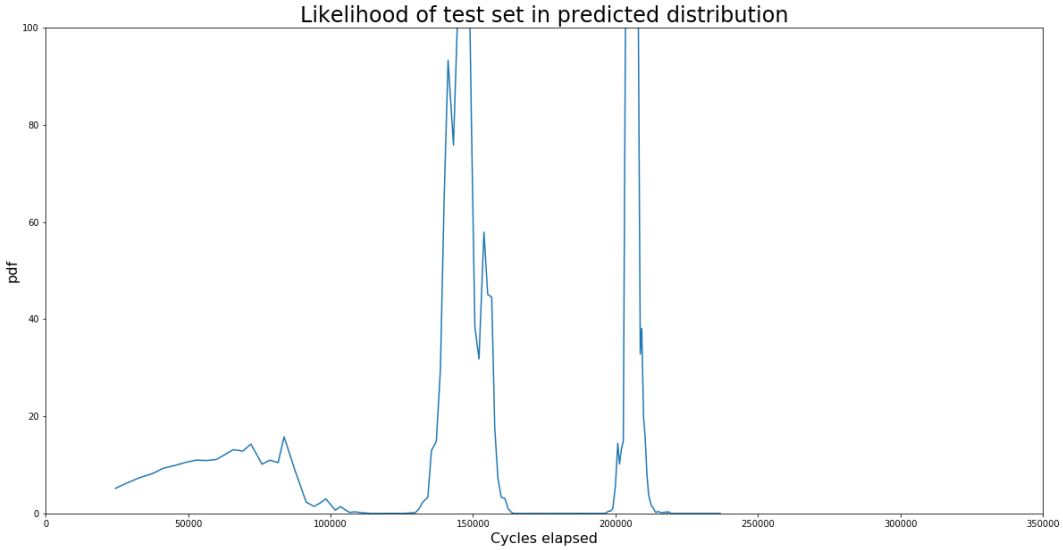
The intuition drawn from Table 7 were confirmed through the visual assessment of the performance on the test set visually. A sample of the test set is used to display three graphs (same as displayed on the SVR model's results: "4.1.2. Results"). The sample displayed reflects the most important conclusions, applicable to the results on the rest of the test set. The first graph shows in blue the sample from the test set being illustrated, with the training set behind in grey. This graph will show how similar that particular test sample is with respect to the sequences the models have been built on. The second graph has already been shown in the previous section for the validation set. This graph displays the number of cycles elapsed on a certain window versus the predicted last cycle, which is calculated as the number of cycles elapsed plus the predicted RUL. A 99.7% confidence interval is shown (normal distribution up to 3 sigma) for the prediction of the last cycle. The last graph shows the likelihood of the actual point from the test set in the predicted distribution. The likelihood is calculated with the probability density function through the predicted mean and variance.



(a)



(b)



(c)

Figure 74: Evaluation graphs on RNN prediction sample of test set

The graph comparing the last cycle to the current cycle (Figure 74 (b)) reflects that the mean prediction doesn't approximate to the real values of RUL the more sequence the RNN is fed. In the same graph, the confidence interval doesn't capture the true values for a significant part of the sequence. Towards the end of the sequence, the true values aren't captured in the confidence interval, which is something that should be avoided by any means. The overall performance is unsuccessful. Therefore, in an attempt to help the model generalize better on the test set, regularization was explored through dropout of the input vectors. In this type of RNN dropout, a probability of excluding each input connection is defined. This way, during the training of the model, each input has a probability (50% in the displayed case) of being excluded from the calculations. This adds noise and can help overcoming overfitting problems. As seen in Table 8 the results reflect an improvement when using dropout, but the visual assessment reveals that the previous problems are still present (Figure 75). Therefore, this alternative was discarded.

Table 8: Scores of the RNN (with 50% input dropout) model on the different sets (training, validation and test)

Metric	Training	Validation	Test
R^2	0.992	0.995	0.966
MAE [cycles]	1435.9	1452.6	7145.2
MPDF	164.5	160.4	16.5

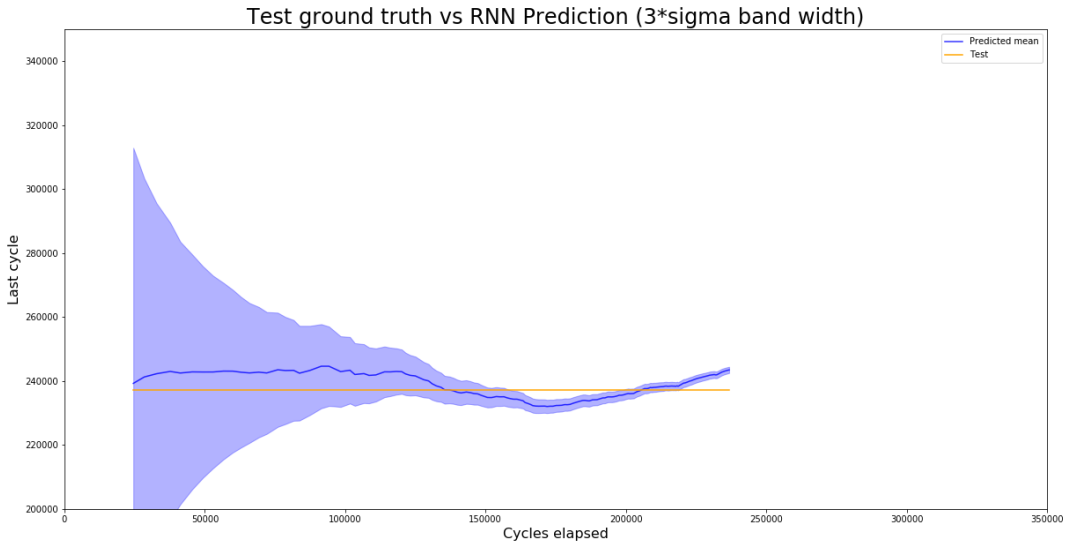


Figure 75: Evaluation graph on RNN (with 50% input dropout) prediction sample of test set

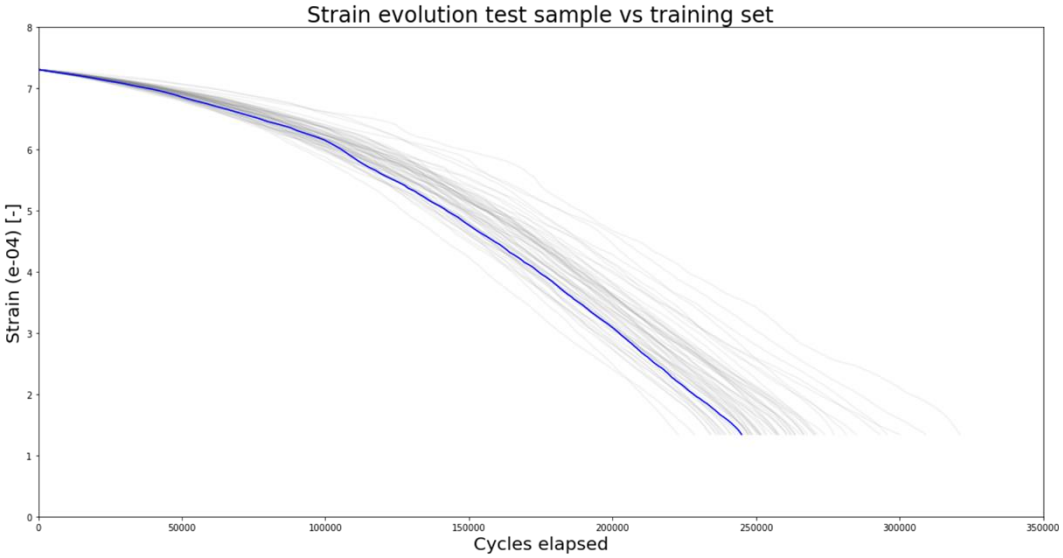
It is clear that the RNN didn't do a good job at translating what was learned from the simulated sequences to the real sequences on Virkler's dataset. In order to check if the same RNN would perform well if trained with real data, the Virkler's dataset of 68 sequences were divided in partial sequences the same way that the validation and test sets

for the RNN were generated. The first partial sequence for each complete sequence will include all the points with less than 245353 cycles (window length of the SVR model) elapsed. The following windows will include one more point each until the end of the sequence is reached. Then, the data was split in training and test set (90/10) and the exact same RNN model was trained on it. The model trained on real data (to predict also real data) delivered good results, as seen in Table 9. All the metrics improved significantly, but the most relevant is the MPDF. MPDF improved by 29.2% on the test set. The true values are now better predicted by the distribution prediction. This in confirmed by the visual assessment of the results on Figure 76. The confidence intervals are more cautious, always capturing the real values in them. Even though the variance prediction is bigger, the MPDF is higher, which can be interpreted as more reliable estimations. The PDF values are also always steadily increasing, meaning that the model is predicting better the more sequence it is fed.

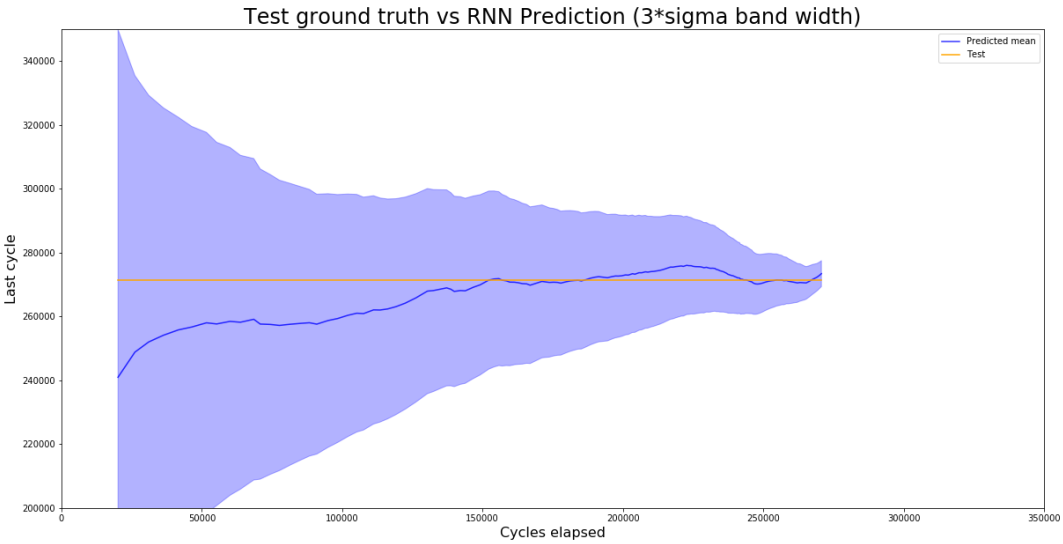
It is worth noting that at the very end of the sequences this model, as well as the original RNN model and the SVR model, predict a slightly higher life than the real value. This can be seen in the Figure 76 (b) as an overestimation of the last cycle at the very end, as well as in Figure 76 (c) as a drop in the value of the PDF. The Virkler's sequences show an increase in the degradation at the very end that could explain this behavior. These accelerations in the degradation on the Virkler's sequences could be the result of the crack reaching the third region of crack growth, where the crack grows much more rapidly than before.

Table 9: Scores of the RNN model trained with real data on the different sets (training and test)

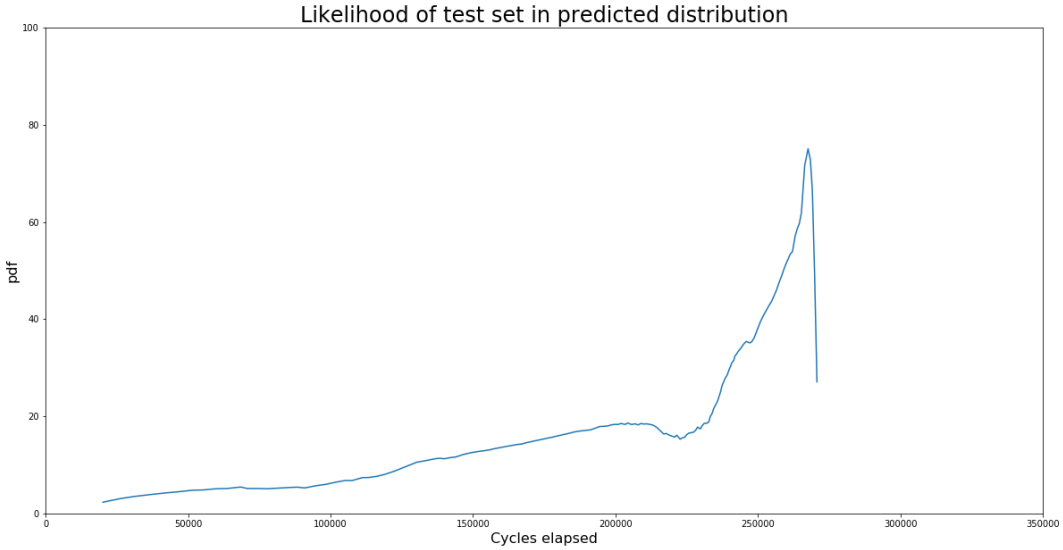
Metric	**Training**	**Test**
R^2	0.982	0.983
MAE [cycles]	5056	5246
MPDF	19.2	19.9



(a)



(b)



(c)

Figure 76: Evaluation graphs on RNN (trained with real data) prediction sample of test set

5. Conclusions

The models designed show the possibilities for the prediction of RUL with two models from the machine learning field. The design, results and analysis were different from one to another, but there are common conclusions than can be drawn from the whole work. The conclusions regarding how the problem was tackled, the techniques employed and the work planification will be presented first as "Methodology conclusions". Then, the conclusions regarding the results from the models are exposed. Finally, the recommendations for future work derived from this research are exhibited.

5.1. Methodology conclusions

The main question to answer in the research was if it was possible to estimate real degradation behavior by using simulated (or analytical) sequences. To develop a reliable model the simulation data should be as close to reality as possible. Thankfully, Virkler's dataset has been deeply studied and analyzed and simulations of this kind were possible. But even with all this previous information, the simulated sequences missed some behaviors of the true sequences, like the irregular acceleration of the degradation (crack growth rate). This could have been modeled with more advanced approaches that, for example, considered a variation of the Paris law's parameters (C and m) during the crack growth sequence. The other option would be obtaining more real data. Nowadays the tendency is to implement more sensors to obtain more data, which would greatly improve the performance on real data of the models. Another alternative that wasn't explored is the training of the models with mixed data from simulations and real values. The models might get the advantages from both sides: bigger training set with the general common behavior in the simulations and real data, and the variations of the real data not explained by the analytical model.

It is also worth mentioning that the strain measurements were generated from the crack growth sequences by transforming each point's crack growth into strain through XFEM, and adding noise to emulate the measurements of strain gauges. The availability of real strain gauges measurements would have allowed more realistic results, as well as a more reasonable simulation of the strain measurement error. The results from the work revealed that it can be possible to model real crack growth behaviors through simulations if the phenomenon is known, at least to some degree. If the real data is not available, this would be the recommended solution.

The research on fatigue crack growth has focused on crack lengths, and not on other relevant features that might make more sense in reality, like strain measurements. Nevertheless, it would also be interesting to model the phenomenon with crack length and compare it to previous work.

There was also a technical aspect worth remembering that had a great impact on the performance: the normalization of the inputs and outputs. The inputs are recommended to be normalized or standardized (they were normalized due to the data not having outliers and also not having a Gaussian distribution) in order to accelerate the learning and facilitate the convergence of the algorithm. This was proven to be true, but the normalization of the output was also crucial. With the outputs bounded between [0,1] both algorithms learned better and much more quickly.

Finally, another technical aspect worth noting is the importance of the visual assessment of the results. The metrics employed might suggest that, for example, certain parameter selection of the model are performing better. When visually checked, it turned out that it was not capturing the sections of the sequences as good as expected. The solution was assessing visually if the score from the selected metric reflected an overall better performance. The alternative is employing a more advanced metric that is able to capture all the behaviors that are wanted in the predictions.

5.2. Results conclusions

The results (the three selected metrics) of both models are displayed on Table 10. The SVR model consisted on two training and two validation sets, so the average of them was calculated in order to show just one training and one validation score and compare it more easily with the RNN model's results.

Table 10: Scores of the SVR and RNN models on the different sets (training, validation and test)

SVR

Metric	Training	Validation	Test
R^2	0.981	0.989	0.966
MAE [cycles]	6347.3	3441.9	5628.5
MPDF	18.6	33.4	17.1

RNN

Metric	Training	Validation	Test
R^2	0.989	0.995	0.945
MAE [cycles]	1498.7	1285.3	8718.0
MPDF	197.1	191.6	15.4

The results show that the SVR model was clearly more fitting for the task than the RNN. There were two factors that could have negatively affected the performance of the SVR model with respect to RNN, but they didn't end up being a problem. First, due to the nature of the model, the SVR only accepts a fixed number if inputs and doesn't capture information from points outside of the selected window, while the RNN considers all the points from the beginning of each sequence until the current studied point. Even with less information, and not considering the time dependency, SVR obtained better results in the test set. Secondly, the SVR model consisted on two sub-models (SVR and KNN) that were not trained to optimize the MPDF, yet they performed better on this metric on the test set than the RNN. The results in training and validation were much better for the RNN model, not only in the MPDF, which is self-evident because RNN was trained to optimize this metric, but also in the R^2 and MAE metrics. This suggests that if the simulations were more similar to the real data, the RNN would have performed better than the SVR.

The main conclusion to draw from the results is that, when trying to predict real data by training a model with simulations, the model's complexity must be low and it must have a great prior generalization potential, like the SVR has. Structural Risk Minimization principles that SVR is based on clearly favor developing a more solution that can be generalized for the real data in this case. As for the RNN, it is an algorithm prone to overfitting, so it might not be the best solution when the training and the test data come from different distributions (sources). Even with the addition of regularization through dropout of the inputs, the model did not generalize well, and didn't translate in a good performance on the test set. Although, as explained in the previous paragraph, the RNN model would perform well when trained on real data. This idea was tested, delivering the results shown in Table 9 and Figure 76. If the real data is available, RNN would be a very valid model to use in order to predict RUL.

Finally, there is an important aspect to comment about the Virkler's dataset. Every sequence accelerates towards the end of it, hindering the predictions of the models. One of the reasons for this acceleration in the crack growth (degradation process) could be that the crack growth is reaching the third region, displayed on Figure 77. In this region, the crack growth accelerates much more rapidly and approaches like Paris' law aren't applicable. This issue could be tackled by establishing a slightly more restrictive threshold.

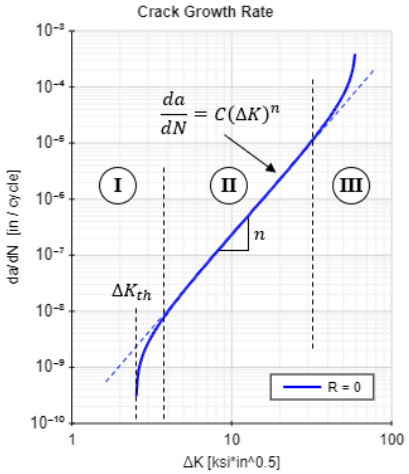


Figure 77: Fatigue crack growth rate [MECH]

5.3. Recommendations for future work

The work shown on this dissertation belongs to a research on RUL in fatigue crack growth that will be continued. The models developed are desired to be compared with previous approaches, so the work will continue by focusing on the estimation of RUL by employing the crack length sequences, instead of the strain evolution. Other models will potentially be explored, which would add a more general idea on the potential of approximating the RUL with these kind of data-driven models.

It would also be interesting to explore more deeply on real data. Real data of either the strain measurements of the crack length. The RNN model showed that it could provide good estimations of the RUL when trained with the real data from Virkler's dataset. More of these data-driven models can also perform well and could be very useful for situations

where real data is available. If the study is performed on the strain evolution, a study on the performance of the strain gauges employed for fatigue crack growth scenarios would be very valuable.

Additionally, the RUL was directly estimated, not worrying about the whole crack propagation process. It would be interesting to compare the performance of models directly estimating the value of RUL versus the models that forecast the remaining crack propagation and determine the RUL based on it. Some already employed models for this approach, based on time series analysis, and are ARIMA [SOLO91], Gaussian Process [MOHA09] or ANN [MOHN09].

Finally, the scatter of the data still has room for improvement. In this work the Paris law's C parameter was scattered for each sequence, but the real data's crack growth accelerated differently for different regions in a same sequence. This can happen due to the inhomogeneity inside a specimen, where the grains will crack in different ways and at different rates for the same loading and conditions. Therefore, the variation of the crack growth rate should be considered on each sequence. If the Paris' law is utilized, a study on the variation of the parameters inside each crack growth sequence would be very useful. Furthermore, an appropriate scatter should also be added to the loading, the geometrical dimensions, environmental conditions, material properties, etc.

6. Literature

[MRAZ14] Mraz, Stephen. “Basics of Aerospace Materials: Aluminum and Composites.” Machine Design, 19 June 2014, https://www.machinedesign.com/materials/basics-aerospace-materials-aluminum-and-composites.

[DOOR15] Doornbos, Peter. “Boeing 787 Dreamliner Specs.” Modern Airliners, 2015, http://www.modernairliners.com/boeing-787-dreamliner/boeing-787-dreamliner-specs/.

[VIRK78] Virkler, Dennis Andrew, Brnm Hillberry, and P. K. Goel. "The statistical nature of fatigue crack propagation." Journal of Engineering Materials and Technology 101.2 (1979): 148-153.

[IVAN14] Ivan, Morgun. “How-to Simulate Support Vector Machine (SVM) in R.” Proft.me, 22 Apr. 2014, https://en.proft.me/2014/04/22/how-simulate-support-vector-machine-svm-r/.

[XPER18] Xpertup. “Understand Recurrent Neural Network (RNN).” XpertUp, 9 June 2018, http://www.xpertup.com/2018/06/09/understand-recurrent-neural-network-rnn/.

[BARU18] Baru, Aditya. “Three Ways to Estimate Remaining Useful Life for Predictive Maintenance.” MATLAB & Simulink, MathWorks, 2018, https://www.mathworks.com/company/newsletters/articles/three-ways-to-estimate-remaining-useful-life-for-predictive-maintenance.html.

[KHEL16] Khelif, Racha, et al. "Direct remaining useful life estimation based on support vector regression." IEEE Transactions on industrial electronics 64.3 (2016): 2276-2285.

[TOTA04] “Aircraft and Aerospace Applications: Part Two.” Total Materia, Apr. 2004, http://www.totalmateria.com/Article96.htm.

[TURK16] Turkoz, Emre. “Materials Science.” Princeton University, The Trustees of Princeton University, 2016, https://www.princeton.edu/~eturkoz/materials.html.

[MECH] “Fatigue Crack Growth.” MechaniCalc, https://mechanicalc.com/reference/fatigue-crack-growth.

[ELBE70] Elber, Wolf. "The significance of fatigue crack closure." Damage tolerance in aircraft structures. ASTM International, 1971.

[AN__13] An, Dawn, Joo-Ho Choi, and Nam Ho Kim. "Prognostics 101: A tutorial for particle filter-based prognostics algorithm using Matlab." Reliability Engineering and System Safety 115 (2013): 161-169.

[WANG17] Wang, Wenyi, Weiping Hu, and Nicholas Armstrong. "Fatigue crack prognosis using Bayesian probabilistic modelling." Mechanical Engineering Journal 4.5 (2017): 16-00702.

[BELY99] Belytschko, Ted, and Tom Black. "Elastic crack growth in finite elements with minimal remeshing." International journal for numerical methods in engineering 45.5 (1999): 601-620.

[MELE96] Melenk, Jens M., and Ivo Babuška. "The partition of unity finite element method: basic theory and applications." Computer methods in applied mechanics and engineering 139.1-4 (1996): 289-314.

[DASS09] Dassault Systèmes. “Abaqus Analysis User's Manual.” Abaqus 6.9-EF Online Documentation, 6 Oct. 2009, http://130.149.89.49:2080/v6.9ef/books/usb/default.htm?startat=pt04ch10s06at33.html.

[SONG06] Song, Jeong - Hoon, Pedro MA Areias, and Ted Belytschko. "A method for dynamic crack and shear band propagation with phantom nodes." International Journal for Numerical Methods in Engineering 67.6 (2006): 868-893.

[MOHA14] Mohammadi, Tayyebeh. "Failure mechanisms and key parameters of FRP debonding from cracked concrete beams." (2014).

[SOLO91] Solomos, G. P., and V. C. Moussas. "A time series approach to fatigue crack propagation." Structural Safety 9.3 (1991): 211-226.

[SPEN89] Spencer, B. F., J. Tang, and M. E. Artley. "Stochastic approach to modeling fatigue crack growth." AIAA journal 27.11 (1989): 1628-1635.

[BOWE08] Bower, Alan F. “Chapter 9: Modeling Material Failure.” Applied Mechanics of Solids (A.F. Bower), 2008, solidmechanics.org/text/Chapter9_4/Chapter9_4.htm.

[ZEHN12] Zehnder, Alan T. "Linear elastic stress analysis of 2d cracks." Fracture Mechanics. Springer, Dordrecht, 2012. 7-32.

[MOHA09] Mohanty, Subhasish, et al. "Gaussian process time series model for life prognosis of metallic structures." Journal of Intelligent Material Systems and Structures 20.8 (2009): 887-896.

[MOHN09] Mohanty, J. R., et al. "Application of artificial neural network for predicting fatigue crack propagation life of aluminum alloys." (2009).

[VAPN95] Vapnik, Vladimir. The nature of statistical learning theory. Springer science & business media, 2013.

[GUNN98] Gunn, Steve R. "Support vector machines for classification and regression." ISIS technical report 14.1 (1998): 5-16.

[GHAN12] Ghanbari, Abdolreza Asadi, et al. "Brain Computer Interface with Wavelets and Genetic Algorithms." Wavelet Transforms and Their Recent Applications in Biology and Geoscience (2012): 119.

[AMID18] Amidi, Afshine, and Shervine Amidi. “Supervised Learning Cheatsheet Star.” CS 229 - Supervised Learning Cheatsheet, Sept. 2018, https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-supervised-learning.

[MOUS18] Moustapha, Maliki, et al. "Comparative study of Kriging and support vector regression for structural engineering applications." ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part A: Civil Engineering 4.2 (2018): 04018005.

[QUCI19] Qucit. “A Simple Technique to Estimate Prediction Intervals for Any Regression Model.” Qucit, 20 May 2019, https://qucit.com/a-simple-technique-to-estimate-prediction-intervals-for-any-regression-model_en/.

[CHE_18] Che, Zhengping, et al. "Recurrent neural networks for multivariate time series with missing values." Scientific reports 8.1 (2018): 6085.

[HOCH97] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[OLAH15] Olah, Christopher. “Understanding LSTM Networks.” Understanding LSTM Networks -- Colah's Blog, colah.github.io/posts/2015-08-Understanding-LSTMs/.

[ZYCH18] Zychlinski, Shaked. “Predicting Probability Distributions Using Neural Networks.” Taboola Engineering, 13 Nov. 2018, https://engineering.taboola.com/predicting-probability-distributions/.

[BENG04] Bengio, Samy. “An Introduction to Statistical Machine Learning - Theoretical Aspects -.” Dalle Molle Institute for Perceptual Artificial Intelligence (IDIAP), 21 Oct. 2004, https://bengio.abracadoudou.com/lectures/old/tex_theory.pdf.

[SCIK19] “RBF SVM Parameters.” *RBF SVM Parameters*, Scikit-Learn, scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html.

7. Appendix I – Crack length data generation

```
import numpy as np
import matplotlib.pyplot as plt
import csv
import scipy.stats as stats
import math
import ast
import copy
```

Virkler data generation from Wang's paper (2017)

```
np.random.seed(101)
```

1. Dimensions and properties definition

```
# Initial half-crack length [mm]
A0 = 18.0/2

# b: semi-width of the specimen [mm]
B = 152.4/2

# Parameter 'm' from Paris' law [-]
M = 2.9

# Parameter 'C' from Paris' law, normal distribution [MPa^(-M) * m^(1-M/2)]
MU_C = 8.586 * 10**-11
SIGMA_C = 0.619 * 10**-11

# Maximum crack length studied in Virkler's test [mm]
A_CRIT = 99.6/2

# Delta stress at the ends of the specimen [MPa]
SIG_MAX = 60.34
SIG_MIN = 12.08
DELTA_SIG = SIG_MAX - SIG_MIN

# Convergence of the crack growth at da = 0.2 mm
STEP_SIZE = 0.2
```

2. Generate examples

```
# Geometry factor to calculate delta_K [-]
def geometry_factor(a,b):
    return 1/np.sqrt(np.cos(np.pi/2*a/b))

EXAMPLES = 1000

C = []

a = []
N = []
```

```
for i in range(EXAMPLES):

    C.append(np.random.normal(MU_C, SIGMA_C))

    N.append([0])
    a.append([A0])

    while a[i][-1] < A_CRIT:
        # delta_K [MPa * sqrt(m)]
        delta_K = geometry_factor(a[i][-1],B) * DELTA_SIG * np.sqrt(np.pi*a[i
][-1])/np.sqrt(1000)

        # dN_da [1/mm]
        dN_da = 1/(C[-1] * delta_K**M * 1000)

        a_new = round(a[i][-1] + STEP_SIZE, ndigits=2)
        N_new = N[i][-1] + STEP_SIZE * dN_da

        a[i].append(a_new)
        N[i].append(N_new)

a_2 = [[crack * 2 for crack in seq] for seq in a]
```

3. Relevant plots

```
plt.figure(figsize = (20,10))
for i in range(100):
    plt.plot(N[i], a_2[i])
    plt.title("Crack length evolution from generated examples", fontsize=24)
    plt.xlabel('Cycles', fontsize = 20)
    plt.ylabel('Crack length (2*a) [mm]', fontsize = 20)
plt.show()
```

png

png

```
count, bins, ignored = plt.hist(C, 30, density = False)
x = np.linspace(MU_C - 3 * SIGMA_C, MU_C + 3 * SIGMA_C, 1000)
bin_width = bins[1]-bins[0]
scaler = np.sum(count * bin_width)
pdf = 1/(SIGMA_C * np.sqrt(2 * np.pi)) * np.exp( - (x - MU_C)**2 / (2 * SIGMA
_C**2) )
plt.plot(x, pdf * scaler, linewidth=2, color='r')
plt.title("Paris' law C parameter sampled values", fontsize = 16)
plt.xlabel('C [-]', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.show()
```

png

png

```
N_last = [max(N[i]) for i in range(len(N))]
```

```
plt.hist(N_last)
plt.title("Last cycle values", fontsize=20)
plt.xlabel('Last cycle', fontsize=16)
plt.ylabel('Count', fontsize=16)
plt.show()
```

png

png

```
RUL = np.zeros(np.shape(N))
for i in range(len(N)):
    RUL[i] = [N_last[i]-val for val in N[i]]
```

4. Save crack growth as csv files

```
with open('N_last_gen.csv', mode='w', newline='') as f:
    writer=csv.writer(f)
    writer.writerows(map(lambda x: [x], N_last))
```

```
aN = []
for i in range(len(a_2)):
    aN.append(list([a_2[i][j], N[i][j]] for j in range(len(a_2[i]))))
```

```
with open('aN_gen.csv', mode='w', newline='') as f:
    writer=csv.writer(f)
    writer.writerows(aN)
```

```
with open('RUL_gen.csv', mode='w', newline='') as f:
    writer=csv.writer(f)
    writer.writerows(RUL)
```

8. Appendix II – Strain data generation

```
import numpy as np
import matplotlib.pyplot as plt
import csv
import scipy.stats as stats
import math
import ast
import copy
```

Crack length to strain conversion of generated data

1. Import generated crack length sequences

```
# Analytical data generated from Virkler's dataset
with open('aN_gen.csv', 'r') as f:
  reader = csv.reader(f)
  aN = list(reader)

# Imported data has points as strings. Here they are evaluated and converted
to lists
for i in range(len(aN)):
     for j in range(len(aN[i])):
            aN[i][j] = ast.literal_eval(aN[i][j])

# Array linking crack length with the corresponding strain [crack_length, str
ain]
with open('crack_to_strain.csv', 'r') as f:
  reader = csv.reader(f)
  crack_strain = list(reader)
```

2. Conversion from crack length to strain

```
cracks_ref = [float(vec[0]) for vec in crack_strain]

cracks_ref = np.asarray(cracks_ref)

eN = copy.deepcopy(aN)
for i in range(len(aN)):
    for j in range(len(aN[i])):
        idx = np.where(cracks_ref == aN[i][j][0]/2)[0].tolist()[0]
        eN[i][j][0] = float(crack_strain[idx][1])
```

3. Virtual strain measurement error

```
strains = [float(point[1]) for point in crack_strain]

gauge_tolerance = round(np.mean(strains) * 0.05,3)
```

To emulate real measurements, a random walk is introduced through a Gaussian distribution with mean 0 and std dev (tolerance/3)

```
std_dev = gauge_tolerance/3
```

```
for i in range(len(eN)):
    for j in range(len(eN[i])):
        eN[i][j][0] = eN[i][j][0] + np.random.normal(0, std_dev)
```

4. Save strain evolution as csv file

```
with open('eN_gen.csv', mode='w', newline='') as f:
    writer=csv.writer(f)
    writer.writerows(eN)
```

5. Plot the strain evolution examples

```
eN = np.asarray(eN)

eN_display = eN.swapaxes(1,2)

plt.figure(figsize = (20,10))
for seq in eN_display:
    plt.plot(seq[1], seq[0])
    plt.title("Strain evolution in generated examples", {'fontsize':24})
    plt.xlabel('Cycles',{'fontsize':20})
    plt.ylabel('Strain (e-04) [-]',{'fontsize':20})
plt.show()
```

png

png

9. Appendix III – Strain data from Virkler's tests

```
import matplotlib.pyplot as plt
import csv
import ast
import copy
import numpy as np
import pandas as pd
```

Crack length to strain (real Virkler tests)

1. Importing Virkler's dataset

```
data = pd.read_csv('../../04 - Virkler Data/Virkler.csv')
```

1.1. Obtaining the RUL

```
# Selecting the columns with the 68 crack growth sequences
columns_names = []
for i in range(68):
    columns_names.append('CycleCount' + str(i+1))

# Converting the cycles elapsed to remaining useful life (RUL)
data_RUL = data[columns_names].iloc[-1]-data[columns_names]

RUL=data_RUL.T.values.tolist()

# Save the RUL at each point as csv file
with open('RUL_real.csv', mode='w', newline='') as f:
    writer=csv.writer(f)
    writer.writerows(RUL)
```

1.2. Obtaining the last sysle of each sequence

```
N_last = data_RUL.iloc[0].values.tolist()

plt.hist(N_last)
plt.show()
```

png

png

```
with open('N_last_real.csv', mode='w', newline='') as f:
    writer=csv.writer(f)
    writer.writerows([[val] for val in N_last])
```

1.3. Obtaining the crack growth sequences

```
aN = []
for i in range(68):
    columns = ['CrackLength','CycleCount' + str(i+1)]
    aN.append(data[columns].values.tolist())

aN = np.asarray(aN)

aN_display = aN.swapaxes(1,2)
```

```
plt.figure(figsize = (20,10))
flag = 1
for seq in aN_display:
    plt.plot(seq[1], seq[0])
    plt.title("Virkler's dataset", {'fontsize':28})
    plt.xlabel('Cycles',{'fontsize':20})
    plt.ylabel('Crack length (2a) [mm]',{'fontsize':20})
    flag=0
plt.show()
```

png

png

2. Convert the crack length to strain

```
# Array linking crack length with the corresponding strain [crack_length, str
ain]
with open('crack_to_strain.csv', 'r') as f:
  reader = csv.reader(f)
  crack_strain = list(reader)

cracks_ref = [float(vec[0]) for vec in crack_strain]

cracks_ref = np.asarray(cracks_ref)

# Each crack growth value is mapped to the corresponding strain
eN = copy.deepcopy(aN)
for i in range(len(aN)):
    for j in range(len(aN[i])):
        idx = np.where(cracks_ref == aN[i][j][0])[0].tolist()[0]
        eN[i][j][0] = float(crack_strain[idx][1])
```

3. Save strain evolution as csv file

```
with open('eN_real.csv', mode='w', newline='') as f:
    writer=csv.writer(f)
    writer.writerows(eN.tolist())
```

4. Plot the strain evolution examples

```
eN_display = eN.swapaxes(1,2)

plt.figure(figsize = (20,10))
for seq in eN_display:
    plt.plot(seq[1], seq[0])
    plt.title("Strain evolution in Virkler's dataset", {'fontsize':24})
    plt.xlabel('Cycles',{'fontsize':20})
    plt.ylabel('Strain (e-04) [-]',{'fontsize':20})
plt.show()
```

png

png

10.Appendix IV – SVR model

```
import csv
import ast
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
import matplotlib.pyplot as plt
from tabulate import tabulate
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
import pickle
```

1. Importing the data

```
# Analytical data generated from Vgen's test
with open('eN_gen.csv', 'r') as f:
  reader = csv.reader(f)
  eN_gen = list(reader)
with open('RUL_gen.csv', 'r') as f:
  reader = csv.reader(f)
  RUL_gen = list(reader)

with open('N_last_gen.csv', 'r') as f:
  reader = csv.reader(f)
  N_last_gen = list(reader)

# Real data from Virkler's reals
with open('eN_real.csv', 'r') as f:
  reader = csv.reader(f)
  eN_real = list(reader)
with open('RUL_real.csv', 'r') as f:
  reader = csv.reader(f)
  RUL_real = list(reader)

with open('N_last_real.csv', 'r') as f:
  reader = csv.reader(f)
  N_last_real = list(reader)
```

Imported data has points as strings. Here they are evaluated and converted to lists

```
for i in range(len(eN_gen)):
     for j in range(len(eN_gen[i])):
            eN_gen[i][j] = ast.literal_eval(eN_gen[i][j])

for i in range(len(RUL_gen)):
     for j in range(len(RUL_gen[i])):
            RUL_gen[i][j] = ast.literal_eval(RUL_gen[i][j])

for i in range(len(N_last_gen)):
     for j in range(len(N_last_gen[i])):
            N_last_gen[i][j] = ast.literal_eval(N_last_gen[i][j])
```

```
for i in range(len(eN_real)):
     for j in range(len(eN_real[i])):
            eN_real[i][j] = ast.literal_eval(eN_real[i][j])

for i in range(len(RUL_real)):
     for j in range(len(RUL_real[i])):
            RUL_real[i][j] = ast.literal_eval(RUL_real[i][j])

for i in range(len(N_last_real)):
     for j in range(len(N_last_real[i])):
            N_last_real[i][j] = ast.literal_eval(N_last_real[i][j])
```

- **Input**: strain-cycle partial sequences
- **Output**: RUL

```
X_gen_complete = eN_gen
y_gen_complete = RUL_gen

X_real_complete = eN_real
y_real_complete = RUL_real
```

2. Training, validation and test sets

In the SVR model, the training, validation and test sets are obtained differently.

- **Training set** is obtained from the generated data (90%) through non-overlapping windows

- **Validation set** is obtained from the generated data (10%) through sliding window

- **Test set** is gathered from the real Virkler data through sliding window

```
X_train_complete, X_valid_complete, y_train_complete, y_valid_complete = trai
n_test_split(X_gen_complete, y_gen_complete, test_size=0.1, shuffle = False)

X_test_complete, y_test_complete = X_real_complete, y_real_complete
```

Saving the last values of each sequence in the training, validation and test set

```
N_last_train_complete, N_last_valid_complete = train_test_split(N_last_gen, t
est_size=0.1, shuffle = False, random_state = 101)

N_last_test_complete = N_last_real
```

2.1. Training data

Obtained from the generated data through non-overlapping windows

The strain data isn't collected each fixed increment of time or strain, but with fixed increment of crack length.

We will employ a fixed window length and use as RUL the one from last measurement included in the window.

The window length will be such that it divides the sequences in around 10 windows.

```
mean_N_last_gen = np.mean(N_last_gen)
print('The mean number of cycles of each complete sequence is:', mean_N_last_
gen)
```

```
The mean number of cycles of each complete sequence is: 245530.6710224977
```

```
#There are around 245,000 cycles per sequence, so in order to have around 10
partial sequences per complete sequence
# the chosen window length will be:
WINDOW = round(mean_N_last_gen/10)
print('Window length:',WINDOW)
```

```
Window length: 24553.0
```

Separating each complete series in around 10 partial series (window width = 24535 cycles)

```
X_train_partial=[]
y_train_partial = []

# Last cycle of the complete sequence, assigned to each partial sequence
N_last_train_partial = []
# Last cycle of the partial sequence
N_last_train_partial_window = []

# List saving the upper bounds of the windows
upper_bounds_train = []

for i in range(len(X_train_complete)):
    # Each sequence mapped starting from the last index
    j = len(X_train_complete[i])-1
    # The first window's upper limit is at the end of the sequence
    N_max = X_train_complete[i][-1][1]
    # The first point is at the end of the sequence
    N = N_max

    # Each window limits (N_max-window; N_max) are generated until the lower
limit is < 0
    while (N_max-WINDOW)>=0:
        # This list will gather the values within the window
        window_list=[]

        # Save the N_last corresponding to the complete sequence
        N_last_train_partial.append(N_last_train_complete[i])

        # Save the N_last corresponding to the partial sequence in this windo
w
        N_last_train_partial_window.append(N_max)

        # Before mapping in the range of the window, save the RUL of the firs
t value (end of the window to be mapped)
        y_train_partial.append(y_train_complete[i][j])

        # Points falling in each window (N_max-window; N_max) are gathered
        while N >= (N_max-WINDOW):
            window_list.append(X_train_complete[i][j])
            j-=1
            N = X_train_complete[i][j][1]
```

```
        # Saving the strain-cycle points within the window
        X_train_partial.append(window_list)

        upper_bounds_train.append(N_max)

        # Change the window limits for the following window
        N_max-=WINDOW
print('Number of complete sequences:', len(X_train_complete))
```

```
Number of complete sequences: 900
```

```
print('Number of partial sequences:', len(X_train_partial))
print(round(len(X_train_partial)/len(X_train_complete),2), 'partial sequences
 per complete sequence')
```

```
Number of partial sequences: 8556
9.51 partial sequences per complete sequence
```

2.2. Validation data

Obtained from the generated data through sliding window

```
X_valid_partial=[]
y_valid_partial = []

# Last cycle of the complete sequence, assigned to each partial sequence
N_last_valid_partial = []
# Last cycle of the partial sequence
N_last_valid_partial_window = []

# List saving the upper bounds of the windows
upper_bounds_valid = []

for i in range(len(X_valid_complete)):
    # Each sequence mapped starting from the last index
    last_idx = len(X_valid_complete[i])-1
    j = last_idx
    # The first window's upper limit is at the end of the sequence
    N_max = X_valid_complete[i][-1][1]
    # The first point is at the end of the sequence
    N = N_max

    # Each window limits (N_max-window; N_max) are generated until the lower
limit is < 0
    while (N_max-WINDOW)>=0:

        # This list will gather the values within the window
        window_list=[]

        # Save the N_last corresponding to the complete sequence
        N_last_valid_partial.append(N_last_valid_complete[i])

        # Save the N_last corresponding to the partial sequence in this windo
w
        N_last_valid_partial_window.append(N_max)
```

```
        j = last_idx

        # Points falling in each window (N_max-window; N_max) are gathered
        while N >= (N_max-WINDOW):
            window_list.append(X_valid_complete[i][j])
            j-=1
            N = X_valid_complete[i][j][1]

        # Saving the strain-cycle points within the window
        X_valid_partial.append(window_list)
        # Saving the RUL of the first value (end of the mapped window)
        y_valid_partial.append(y_valid_complete[i][last_idx])

        upper_bounds_valid.append(N_max)

        #The fist point will be one position before the last window's
        last_idx-=1
        # Change the window limits for the following window
        N_max = X_valid_complete[i][last_idx][1]
        N = N_max
print('Number of complete sequences:', len(X_valid_complete))
```

```
Number of complete sequences: 100
```

```
print('Number of partial sequences:', len(X_valid_partial))
print(len(X_valid_partial)/len(X_valid_complete), 'partial sequences per comp
lete sequence')
```

```
Number of partial sequences: 19946
199.46 partial sequences per complete sequence
```

2.3. Test data

Gathered from the real Virkler data through sliding window

```
X_test_partial=[]
y_test_partial = []

# Last cycle of the complete sequence, assigned to each partial sequence
N_last_test_partial = []
# Last cycle of the partial sequence
N_last_test_partial_window = []

upper_bounds_test = []

for i in range(len(X_test_complete)):
    # Each sequence mapped starting from the last index
    last_idx = len(X_test_complete[i])-1
    j = last_idx
    # The first window's upper limit is at the end of the sequence
    N_max = X_test_complete[i][-1][1]
    # The first point is at the end of the sequence
    N = N_max
```

```
    # Each window limits (N_max-window; N_max) are generated until the lower
limit is < 0
    while (N_max-WINDOW)>=0:

        # This list will gather the values within the window
        window_list=[]

        # Save the N_last corresponding to the complete sequence
        N_last_test_partial.append(N_last_test_complete[i])

        # Save the N_last corresponding to the partial sequence in this windo
w
        N_last_test_partial_window.append(N_max)

        j = last_idx

        # Points falling in each window (N_max-window; N_max) are gathered
        while N >= (N_max-WINDOW):
            window_list.append(X_test_complete[i][j])
            j-=1
            N = X_test_complete[i][j][1]

        # Saving the strain-cycle points within the window
        X_test_partial.append(window_list)
        # Saving the RUL of the first value (end of the mapped window)
        y_test_partial.append(y_test_complete[i][last_idx])

        upper_bounds_test.append(N_max)

        #The fist point will be one position before the last window's
        last_idx-=1
        # Change the window limits for the following window
        N_max = X_test_complete[i][last_idx][1]
        N = N_max

print('Number of complete sequences:', len(X_test_complete))
```

```
Number of complete sequences: 68
```

```
print('Number of partial sequences:', len(X_test_partial))
print(round(len(X_test_partial)/len(X_test_complete),2), 'partial sequences p
er complete sequence')
```

```
Number of partial sequences: 10860
159.71 partial sequences per complete sequence
```

3. Data preprocessing

3.1. Extract mean and slope for each window

```
reg = LinearRegression()

def mean_slope(sequences):
    """ Extracts the mean and slope from a sequence
```

```
    Parameters:
    X_windows (# examples, # points , 2): input set of sequences

    Returns:
    mean_slope (# examples, 2): mean and slope of each example"""

    mean_slope = []
    for seq in sequences:
        cycles = [[point[1]] for point in seq]
        strain = [[point[0]] for point in seq]
        reg.fit(cycles,strain)
        mean_slope.append([np.mean([point[0] for point in seq]), reg.coef_[0]
[0]])
    return mean_slope

X_train = mean_slope(X_train_partial)
y_train = y_train_partial

N_last_train = N_last_train_partial
N_last_train_window = N_last_train_partial_window

X_valid = mean_slope(X_valid_partial)
y_valid = y_valid_partial

N_last_valid = N_last_valid_partial
N_last_valid_window = N_last_valid_partial_window

X_test = mean_slope(X_test_partial)
y_test = y_test_partial

N_last_test = N_last_test_partial
N_last_test_window = N_last_test_partial_window
```

3.2. Normalization of the mean and slope

```
scaler = MinMaxScaler()

scaler.fit(X_train)

MinMaxScaler(copy=True, feature_range=(0, 1))

X_train_norm = scaler.transform(X_train)

X_valid_norm = scaler.transform(X_valid)

X_test_norm = scaler.transform(X_test)
```

3.3. Normalization of the RUL (output)

```
max_y = np.max(y_train)
y_train_norm = y_train/max_y

y_valid_norm = y_valid/max_y

y_test_norm = y_test/max_y
```

3.4. Splitting data for the Mean model (1) and the Variance model (2)

Training sets

```
X1_train_norm, X2_train_norm, y1_train_norm, y2_train_norm = train_test_split
(X_train_norm, y_train_norm, test_size=0.5, shuffle = False, random_state = 1
01)

N_last1_train, N_last2_train = train_test_split(N_last_train, test_size=0.5,
shuffle = False, random_state = 101)

N_last1_train_window, N_last2_train_window = train_test_split(N_last_train_wi
ndow, test_size=0.5, shuffle = False, random_state = 101)
```

Validation sets

```
X1_valid_norm, X2_valid_norm, y1_valid_norm, y2_valid_norm = train_test_split
(X_valid_norm, y_valid_norm, test_size=0.5, shuffle = False, random_state = 1
01)

N_last1_valid, N_last2_valid = train_test_split(N_last_valid, test_size=0.5,
shuffle = False, random_state = 101)

N_last1_valid_window, N_last2_valid_window = train_test_split(N_last_valid_wi
ndow, test_size=0.5, shuffle = False, random_state = 101)
```

Test set

```
N_last_test = N_last_test_partial

N_last_test_window = N_last_test_partial_window
```

4. Building and training the SVR mean model

```
epsilon, C, gamma = 0.01, 1000, 0.5

svr_mean = SVR(kernel='rbf', C = C, epsilon = epsilon, gamma = gamma)

svr_mean.fit(X1_train_norm, y1_train_norm)

SVR(C=1000, cache_size=200, coef0=0.0, degree=3, epsilon=0.01, gamma=0.5,
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

5. Building and training the KNN variance model

```
knn_variance = KNeighborsRegressor(n_neighbors=500)

# Predict the mean of the variance training set
mu2_pred_train = []
for pred in svr_mean.predict(X2_train_norm):
    mu2_pred_train.append(max(0, pred))

# Compute the prediction error vector on the variance training set
error2_train = (mu2_pred_train - y2_train_norm) ** 2

# Fit the model of variance prediction to the training data
knn_variance.fit(X2_train_norm, error2_train)

KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
          metric_params=None, n_jobs=None, n_neighbors=500, p=2,
          weights='uniform')
```

6. Results

```
def pdf(x, mu, sigma):
    """ Returns the Probablility Density Function value of a point in the giv
en Gaussian distribution

    Parameters:
    x: point to assess in the Gaussian distribution
    mu: mean of the Gaussian distribution
    sigma: standard deviation of the Gaussian distribution

    Returns:
    pdf: PDF value"""

    pdf = []
    for i in range(len(x)):
        num = np.exp(-(x[i]-mu[i])**2/(2*sigma[i]**2))
        den = sigma[i] * np.sqrt(2*np.pi)
        pdf.append(num/den)
    return pdf
```

6.1. Scores

```
mu1_pred_train = []
for pred in svr_mean.predict(X1_train_norm):
    mu1_pred_train.append(max(0, pred))
sigma1_pred_train = np.sqrt(knn_variance.predict(X1_train_norm))

mu1_pred_valid = []
for pred in svr_mean.predict(X1_valid_norm):
    mu1_pred_valid.append(max(0, pred))
sigma1_pred_valid = np.sqrt(knn_variance.predict(X1_valid_norm))

mu2_pred_train = []
for pred in svr_mean.predict(X2_train_norm):
    mu2_pred_train.append(max(0, pred))
sigma2_pred_train = np.sqrt(knn_variance.predict(X2_train_norm))

mu2_pred_valid = []
for pred in svr_mean.predict(X2_valid_norm):
    mu2_pred_valid.append(max(0, pred))
sigma2_pred_valid = np.sqrt(knn_variance.predict(X2_valid_norm))

mu_pred_test = []
for pred in svr_mean.predict(X_test_norm):
    mu_pred_test.append(max(0, pred))
sigma_pred_test = np.sqrt(knn_variance.predict(X_test_norm))

scores={}

scores[''] = ['Train 1', 'Train 2', 'Valid 1', 'Valid 2', 'Test']

mpdf1_train=np.mean(pdf(y1_train_norm,mu1_pred_train,sigma1_pred_train))

mpdf2_train=np.mean(pdf(y2_train_norm,mu2_pred_train,sigma2_pred_train))

mpdf1_valid=np.mean(pdf(y1_valid_norm,mu1_pred_valid,sigma1_pred_valid))
```

```
mpdf2_valid=np.mean(pdf(y2_valid_norm,mu2_pred_valid,sigma2_pred_valid))

mpdf_test=np.mean(pdf(y_test_norm,mu_pred_test,sigma_pred_test))

scores['MPDF'] = [mpdf1_train, mpdf2_train, mpdf1_valid, mpdf2_valid, mpdf_te
st]

mae1_train=mean_absolute_error(y1_train_norm, mu1_pred_train) * max_y

mae2_train=mean_absolute_error(y2_train_norm, mu2_pred_train) * max_y

mae1_valid=mean_absolute_error(y1_valid_norm, mu1_pred_valid) * max_y

mae2_valid=mean_absolute_error(y2_valid_norm, mu2_pred_valid) * max_y

mae_test=mean_absolute_error(y_test_norm, mu_pred_test) * max_y

scores['MAE'] = [mae1_train, mae2_train, mae1_valid, mae2_valid, mae_test]

r2_1_train=r2_score(y1_train_norm, mu1_pred_train)

r2_2_train=r2_score(y2_train_norm, mu2_pred_train)

r2_1_valid=r2_score(y1_valid_norm, mu1_pred_valid)

r2_2_valid=r2_score(y2_valid_norm, mu2_pred_valid)

r2_test=r2_score(y_test_norm, mu_pred_test)

scores['R2'] = [r2_1_train, r2_2_train, r2_1_valid, r2_2_valid, r2_test]

print(tabulate(scores, headers='keys'))
```

```
            MPDF      MAE        R2
-------  -------  -------  --------
Train 1  18.7219  6253.2   0.980884
Train 2  18.4905  6441.42  0.980162
Valid 1  33.417   3328.51  0.989927
Valid 2  33.3359  3555.34  0.987517
Test     17.1216  5628.52  0.965651
```

6.2. Variance validation set

```
# Identifying the partial sequences that belong to the same complete sequence
n2_valid = []
for i in range(len(N_last2_valid)):
    if N_last2_valid[i] != N_last2_valid[i-1]:
        n2_valid.append(i)

pdf2_valid = pdf(y2_valid_norm,mu2_pred_valid,sigma2_pred_valid)

j=0

for i in range(len(n2_valid)-1):

    # Predictions plot
```

```
    n1 = n2_valid[i]
    n2 = n2_valid[i+1]
    plt.figure(figsize = (20,10))
    plt.plot(N_last2_valid_window[n1:n2], np.asarray(mu2_pred_valid[n1:n2], d
type=np.float32) * max_y + N_last2_valid_window[n1:n2]
             , label = 'Predicted mean', color = 'b', alpha=0.8)
    plt.plot(N_last2_valid_window[n1:n2], N_last2_valid[n1:n2]
             , label = 'True last cycle', color = '#ffa500')
    plt.fill_between(N_last2_valid_window[n1:n2]
                     , (mu2_pred_valid[n1:n2] + 3 * sigma2_pred_valid[n1:n2])
 * max_y + N_last2_valid_window[n1:n2]
                     , (mu2_pred_valid[n1:n2] - 3 * sigma2_pred_valid[n1:n2])
 * max_y + N_last2_valid_window[n1:n2]
                     , color = 'B', alpha=0.3)
    plt.title('Validation ground truth vs RNN Prediction (3*sigma band width)
', {'fontsize':24})
    plt.ylabel('Last cycle', {'fontsize':16})
    plt.xlabel('Example', {'fontsize':16})
    plt.xlim(0,3.5e5)
    plt.ylim(2e5,3.5e5)
    plt.legend(loc='best')
    plt.show()

    plt.figure(figsize = (20,10))
    plt.title('Likelihood of validation set in predicted distribution', {'fon
tsize':24})
    plt.ylabel('PDF', {'fontsize':16})
    plt.xlabel('Cycles elapsed', {'fontsize':16})
    plt.plot(N_last2_valid_window[n1:n2],pdf2_valid[n1:n2])
    plt.xlim(0,3.5e5)
    plt.show()
```

png

png

…

6.3. Test set

```
pdf_test = pdf(y_test_norm,mu_pred_test,sigma_pred_test)
```

```
# Identifying the partial sequences that belong to the same complete sequence
n_test = []
for i in range(len(N_last_test)):
    if N_last_test_partial[i] != N_last_test_partial[i-1]:
        n_test.append(i)

j=0

for i in range(len(n_test)-1):
    # Input sequence plot
    plt.figure(figsize = (20,10))
    for seq in X_train_complete[:500]:
        seq = np.asarray(seq).swapaxes(1,0)
```

```
        plt.plot(seq[1], seq[0], color='grey',alpha=0.1)

    seq_test = np.asarray(X_test_complete[j]).swapaxes(1,0)
    plt.plot(seq_test[1], seq_test[0], color='b')
    plt.title("Strain evolution test sample vs training set", {'fontsize':24}
)
    plt.xlabel('Cycles elapsed',{'fontsize':20})
    plt.ylabel('Strain (e-04) [-]',{'fontsize':20})
    plt.xlim(0,3.5e5)
    plt.ylim(0,8)
    plt.show()

    # Predictions plot
    n1 = n_test[i]
    n2 = n_test[i+1]
    plt.figure(figsize = (20,10))
    plt.plot(N_last_test_window[n1:n2], np.asarray(mu_pred_test[n1:n2], dtype
=np.float32) * max_y + N_last_test_window[n1:n2]
             , label = 'Predicted mean', color = 'b', alpha=0.8)
    plt.plot(N_last_test_window[n1:n2], N_last_test[n1:n2]
             , label = 'Test', color = '#ffa500')
    plt.fill_between(N_last_test_window[n1:n2]
                     , (mu_pred_test[n1:n2] + 3 * sigma_pred_test[n1:n2]) * m
ax_y + N_last_test_window[n1:n2]
                     , (mu_pred_test[n1:n2] - 3 * sigma_pred_test[n1:n2]) * m
ax_y + N_last_test_window[n1:n2]
                     , color = 'B', alpha=0.3)
    plt.title('Test ground truth vs RNN Prediction (3*sigma band width)', {'f
ontsize':24})
    plt.ylabel('Last cycle', {'fontsize':16})
    plt.xlabel('Cycles elapsed', {'fontsize':16})
    plt.xlim(0,3.5e5)
    plt.ylim(2e5,3.5e5)
    plt.legend(loc='best')
    plt.show()

    plt.figure(figsize = (20,10))
    plt.title('Likelihood of test set in predicted distribution', {'fontsize'
:24})
    plt.ylabel('PDF', {'fontsize':16})
    plt.xlabel('Cycles elapsed', {'fontsize':16})
    plt.plot(N_last_test_window[n1:n2],pdf_test[n1:n2])
    plt.xlim(0,3.5e5)
    plt.ylim(0,100)
    plt.show()

    j+=1
```

png

png

…

7. Saving the models

```
# Save the SVR mean model to disk
filename = 'SVR_model/svr_mean_model.sav'
pickle.dump(svr_mean, open(filename, 'wb'))

# Save the KNN variance model to disk
filename = 'SVR_model/svr_variance_model.sav'
pickle.dump(knn_variance, open(filename, 'wb'))
```

11. Appendix V – RNN model

```
import csv
import ast
from sklearn.model_selection import train_test_split
import numpy as np
import copy
import tensorflow as tf
from tensorflow.keras.layers import Masking, LSTM
import tensorflow_probability as tfp
import time
from tensorflow.keras.preprocessing.sequence import pad_sequences
import matplotlib.pyplot as plt
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from tabulate import tabulate
```

1. Importing the data

```
# Analytical data generated from Vgen's test
with open('eN_gen.csv', 'r') as f:
  reader = csv.reader(f)
  eN_gen = list(reader)
with open('RUL_gen.csv', 'r') as f:
  reader = csv.reader(f)
  RUL_gen = list(reader)

with open('N_last_gen.csv', 'r') as f:
  reader = csv.reader(f)
  N_last_gen = list(reader)

# Real data from Virkler's reals
with open('eN_real.csv', 'r') as f:
  reader = csv.reader(f)
  eN_real = list(reader)
with open('RUL_real.csv', 'r') as f:
  reader = csv.reader(f)
  RUL_real = list(reader)

with open('N_last_real.csv', 'r') as f:
  reader = csv.reader(f)
  N_last_real = list(reader)
```

Imported data has points as strings. Here they are evaluated and converted to lists

```
for i in range(len(eN_gen)):
     for j in range(len(eN_gen[i])):
            eN_gen[i][j] = ast.literal_eval(eN_gen[i][j])

for i in range(len(RUL_gen)):
     for j in range(len(RUL_gen[i])):
            RUL_gen[i][j] = ast.literal_eval(RUL_gen[i][j])
```

```
for i in range(len(N_last_gen)):
     for j in range(len(N_last_gen[i])):
            N_last_gen[i][j] = ast.literal_eval(N_last_gen[i][j])

for i in range(len(eN_real)):
     for j in range(len(eN_real[i])):
            eN_real[i][j] = ast.literal_eval(eN_real[i][j])

for i in range(len(RUL_real)):
     for j in range(len(RUL_real[i])):
            RUL_real[i][j] = ast.literal_eval(RUL_real[i][j])

for i in range(len(N_last_real)):
     for j in range(len(N_last_real[i])):
            N_last_real[i][j] = ast.literal_eval(N_last_real[i][j])
```

- **Input**: strain-cycle partial sequences
- **Output**: RUL

```
X_gen_complete = eN_gen
y_gen_complete = RUL_gen

X_real_complete = eN_real
y_real_complete = RUL_real
```

2. Train, validation and test sets

In the RNN model, the training, validation and test sets are obtained differently.

- **Training set** is obtained from the generated data (90%) through non-overlapping windows

- **Validation set** is obtained from the generated data (10%) through sliding window

- **Test set** is gathered from the real Virkler data through sliding window

```
X_train_complete, X_valid_complete, y_train_complete, y_valid_complete = trai
n_test_split(X_gen_complete, y_gen_complete, test_size=0.1, shuffle = False)

X_test_complete, y_test_complete = X_real_complete, y_real_complete
```

Saving the last values of each sequence in the training, validation and test set

```
N_last_train_complete, N_last_valid_complete = train_test_split(N_last_gen, t
est_size=0.1, shuffle = False)

N_last_test_complete = N_last_real
```

2.1. Training data

The complete sequences (x) are divided into partial sequences (x_partial) by taking progressively more values form the complete sequence.

For each complete sequence: 1. Take values from the beginning of the complete sequence to 'np.random.randint(1, 40)' 2. Take values from the beginning of the complete sequence to the previous last value + 'np.random.randint(1, 40)' 3. Repeat step 3 until the length of the complete sequence is reached

For each partial sequence, the value of RUL (y_partial) will be chosen as the RUL of the last point in the partial sequence

```
X_train_partial = []
y_train_partial = []

# Last cycle of the complete sequence, assigned to each partial sequence
N_last_train = []
# Last cycle of the partial sequence
N_last_train_window = []

for i in range(len(X_train_complete)):
    n = int(np.random.randint(1, 40))
    while n < len(X_train_complete[i]):
        X_train_partial.append(X_train_complete[i][:n])

        y_train_partial.append(y_train_complete[i][n-1])

        N_last_train.append(N_last_train_complete[i])

        N_last_train_window.append(X_train_complete[i][n-1][1])

        n += int(np.random.randint(1, 40))

print('Number of complete sequences:', len(X_train_complete))
```

```
Number of complete sequences: 900
```

```
print('Number of partial sequences:', len(X_train_partial))
print(round(len(X_train_partial)/len(X_train_complete),2), 'partial sequences
 per complete sequence')
```

```
Number of partial sequences: 8895
9.88 partial sequences per complete sequence
```

2.2. Validation data

Obtained from the generated data by adding one more point for each window

```
WINDOW = 24553

X_valid_partial = []
y_valid_partial = []

# Last cycle of the complete sequence, assigned to each partial sequence
N_last_valid = []
# Last cycle of the partial sequence
N_last_valid_window = []

for i in range(len(X_valid_complete)):

    n=0
    j=0
    # Find the index of the last point before WINDOW cycles
    while X_valid_complete[i][n][1] < WINDOW:
        n+=1
```

```
    while n < len(X_valid_complete[i]):
        X_valid_partial.append(X_valid_complete[i][:n])

        y_valid_partial.append(y_valid_complete[i][n-1])

        N_last_valid.append(N_last_valid_complete[i])

        N_last_valid_window.append(X_valid_complete[i][n-1][1])

        n += 1

print('Number of complete sequences:', len(X_valid_complete))
```

```
Number of complete sequences: 100
```

```
print('Number of partial sequences:', len(X_valid_partial))
print(len(X_valid_partial)/len(X_valid_complete), 'partial sequences per comp
lete sequence')
```

```
Number of partial sequences: 19946
199.46 partial sequences per complete sequence
```

2.3. Test data

```
X_test_partial = []
y_test_partial = []

# Last cycle of the complete sequence, assigned to each partial sequence
N_last_test = []
# Last cycle of the partial sequence
N_last_test_window = []

for i in range(len(X_test_complete)):

    n=0
    j=0
    # Find the index of the last point before window_length cycles
    while X_test_complete[i][n][1] < WINDOW:
        n+=1

    while n < len(X_test_complete[i]):
        X_test_partial.append(X_test_complete[i][:n])

        y_test_partial.append(y_test_complete[i][n-1])

        N_last_test.append(N_last_test_complete[i])

        N_last_test_window.append(X_test_complete[i][n-1][1])

        n += 1

print('Number of complete sequences:', len(X_test_complete))
```

```
Number of complete sequences: 68
```

```
print('Number of partial sequences:', len(X_test_partial))
print(round(len(X_test_partial)/len(X_test_complete),2), 'partial sequences p
er complete sequence')
```

```
Number of partial sequences: 10860
159.71 partial sequences per complete sequence
```

3. Time series preprocessing

3.1. Input normalization

```
max_e = 0; max_N = 0; min_e = 10e8; min_N = 10e8
for sequence in X_train_partial:
    for point in sequence:
        max_e = max(point[0],max_e)
        max_N = max(point[1],max_N)
        min_e = min(point[0],min_e)
        min_N = min(point[1],min_N)

print('Strains in training set between', (min_e,max_e))
print('Number of cycle in training set between', (min_N,max_N))
```

```
Strains in training set between (1.2159674551962765, 7.4735252045890785)
Number of cycle in training set between (0, 303800.13156262465)
```

```
def normalize_strain_and_cycle(X):
    """ Normalizes strain and cycle from the sequences: x_norm = (x-min)/(max
-min)

    Parameters:
    X (# examples, # points , 2): input set of sequences

    Returns:
    X_norm (# examples, # points , 2): normalized strains and cycles"""

    X_norm = copy.deepcopy(X)
    for i in range(len(X)):
        for j in range(len(X[i])):
            X_norm[i][j] = [(X[i][j][0] - min_e)/(max_e-min_e), (X[i][j][1] -
 min_N)/(max_N - min_N)]
    return X_norm

X_train_norm = normalize_strain_and_cycle(X_train_partial)

X_valid_norm = normalize_strain_and_cycle(X_valid_partial)

X_test_norm = normalize_strain_and_cycle(X_test_partial)
```

3.2. Output normalization

```
max_y = np.max(y_train_partial)

y_train_norm = y_train_partial / max_y
y_valid_norm = y_valid_partial / max_y
y_test_norm = y_test_partial / max_y
```

3.3. Division of training data in batches

```
batch_size = 256
batch_num = int(len(X_train_norm) / batch_size)

X_batches = np.array_split(X_train_norm, batch_num)
y_batches = np.array_split(y_train_norm, batch_num)
```

4. Building RNN

```
tf.reset_default_graph()
X = tf.placeholder(name='X',shape=(None,None,2),dtype=tf.float32)
y = tf.placeholder(name='y',shape=(None,1),dtype=tf.float32)

layer = X
layer = Masking(mask_value = 0)(layer)
layer = LSTM(4, return_sequences = False, activation = 'tanh')(layer)

mu = tf.layers.dense(inputs=layer, units=1, activation = 'sigmoid')
sigma = tf.layers.dense(inputs=layer, units=1, activation=lambda x: tf.nn.elu
(x) + 1)

def mdn_cost(mu, sigma, y):
    dist = tfp.distributions.Normal(loc=mu, scale=sigma)
    return tf.reduce_mean(-dist.log_prob(y))

cost = mdn_cost(mu, sigma, y)

epochs = 500
learning_rate = 0.005
display_step = 10

optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

5. Training and saving the RNN

```
# Will be used to plot costs vs epoch
epoch_list = []
costs = []

# Instance of tf.train.Saver() class, will be used to save the trained model
saver = tf.train.Saver()

# Start counting the time elapsed in the optimization
start_time = time.time()

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for epoch in range(epochs):
        avg_cost = 0.0
        for i in range(batch_num):
            X_batch = pad_sequences(X_batches[i], dtype=np.float32)
            y_batch = np.asarray(y_batches[i], dtype=np.float32)[:,np.newaxis
]
            _, c = sess.run([optimizer,cost], feed_dict={X:X_batch, y:y_batch
})
```

```
            avg_cost += c/batch_num
        if epoch % display_step == 0:
            print('Time elapsed (mins): ', (time.time() - start_time)/60)
            print('Epoch {0} | cost = {1:.4f}'.format(epoch,avg_cost))
            epoch_list.append(epoch)
            costs.append(avg_cost)

            plt.plot(epoch_list, costs)
            plt.xlabel('Epoch')
            plt.ylabel('Cost')
            plt.show()

    print('Final cost: {0:.4f}'.format(avg_cost))

    mytime = time.time() - start_time
    print("Time elapsed in optimization (mins): ", mytime/60)

    # Saving the RNN model
    saver.save(sess, 'RNN_model/rnn_model')

    # Making the predictions with the trained model

    mu_pred_train=[]
    sigma_pred_train=[]
    for i in range(len(X_train_norm)):
        mu_pred_train.append(sess.run(mu, feed_dict={X:np.expand_dims(X_train
_norm[i], axis = 0)}).item())
        sigma_pred_train.append(sess.run(sigma, feed_dict={X:np.expand_dims(X
_train_norm[i], axis = 0)}).item())

    mu_pred_valid=[]
    sigma_pred_valid=[]
    for i in range(len(X_valid_norm)):
        mu_pred_valid.append(sess.run(mu, feed_dict={X:np.expand_dims(X_valid
_norm[i], axis = 0)}).item())
        sigma_pred_valid.append(sess.run(sigma, feed_dict={X:np.expand_dims(X
_valid_norm[i], axis = 0)}).item())

    mu_pred_test=[]
    sigma_pred_test=[]
    for i in range(len(X_test_norm)):
        mu_pred_test.append(sess.run(mu, feed_dict={X:np.expand_dims(X_test_n
orm[i], axis = 0)}).item())
        sigma_pred_test.append(sess.run(sigma, feed_dict={X:np.expand_dims(X_
test_norm[i], axis = 0)}).item())
```

6. Results

```
def pdf(x, mu, sigma):
    """ Returns the Probablility Density Function value of a point in the giv
en Gaussian distribution

    Parameters:
    x: point to assess in the Gaussian distribution
```

```
    mu: mean of the Gaussian distribution
    sigma: standard deviation of the Gaussian distribution

    Returns:
    pdf: PDF value"""

    pdf = []
    for i in range(len(x)):
        num = np.exp(-(x[i]-mu[i])**2/(2*sigma[i]**2))
        den = sigma[i] * np.sqrt(2*np.pi)
        pdf.append(num/den)
    return pdf
```

6.1. Scores

```
scores={}

scores[''] = ['Train', 'Valid', 'Test']

mpdf_train=np.mean(pdf(y_train_norm,mu_pred_train,sigma_pred_train))

mpdf_valid=np.mean(pdf(y_valid_norm,mu_pred_valid,sigma_pred_valid))

mpdf_test=np.mean(pdf(y_test_norm,mu_pred_test,sigma_pred_test))

scores['MPDF'] = [mpdf_train, mpdf_valid, mpdf_test]

mae_train=mean_absolute_error(y_train_norm, mu_pred_train) * max_y

mae_valid=mean_absolute_error(y_valid_norm, mu_pred_valid) * max_y

mae_test=mean_absolute_error(y_test_norm, mu_pred_test) * max_y

scores['MAE'] = [mae_train, mae_valid, mae_test]

r2_train=r2_score(y_train_norm, mu_pred_train)

r2_valid=r2_score(y_valid_norm, mu_pred_valid)

r2_test=r2_score(y_test_norm, mu_pred_test)

scores['R2'] = [r2_train, r2_valid, r2_test]

print(tabulate(scores, headers='keys'))
```

```
           MPDF      MAE        R2
-----  --------  -------  --------
Train  197.073   1498.74  0.989485
Valid  191.629   1285.31  0.995462
Test    15.3544  8717.98  0.9451
```

6.2. Validation set

```
# Identifying the partial sequences that belong to the same complete sequence
n_valid = []
for i in range(len(N_last_valid)):
```

```
    if N_last_valid[i] != N_last_valid[i-1]:
        n_valid.append(i)

mu_pred_valid = np.asanyarray(mu_pred_valid)

sigma_pred_valid = np.asarray(sigma_pred_valid)

N_last_valid_window = np.asarray(N_last_valid_window)

pdf_valid = pdf(y_valid_norm,mu_pred_valid,sigma_pred_valid)

j=0

for i in range(len(n_valid)-1):
    # Input sequence plot
    plt.figure(figsize = (20,10))
    for seq in X_gen_complete[:500]:
        seq = np.asarray(seq).swapaxes(1,0)
        plt.plot(seq[1], seq[0], color='grey',alpha=0.1)

    seq_valid = np.asarray(X_valid_complete[j]).swapaxes(1,0)
    plt.plot(seq_valid[1], seq_valid[0], color='b')
    plt.title("Strain evolution valid sample vs training set", {'fontsize':24
})
    plt.xlabel('Cycles elapsed',{'fontsize':20})
    plt.ylabel('Strain (e-04) [-]',{'fontsize':20})
    plt.xlim(0,3.5e5)
    plt.ylim(0,8)
    plt.show()

    # Predictions plot
    n1 = n_valid[i]
    n2 = n_valid[i+1]
    plt.figure(figsize = (20,10))
    plt.plot(N_last_valid_window[n1:n2], mu_pred_valid[n1:n2] * max_y + N_las
t_valid_window[n1:n2]
             , label = 'Predicted mean', color = 'b', alpha=0.8)
    plt.plot(N_last_valid_window[n1:n2], N_last_valid[n1:n2]
             , label = 'valid', color = '#ffa500')
    plt.fill_between(N_last_valid_window[n1:n2]
                     , (mu_pred_valid[n1:n2] + 3 * sigma_pred_valid[n1:n2]) *
 max_y + N_last_valid_window[n1:n2]
                     , (mu_pred_valid[n1:n2] - 3 * sigma_pred_valid[n1:n2]) *
 max_y + N_last_valid_window[n1:n2]
                     , color = 'B', alpha=0.3)
    plt.title('valid ground truth vs RNN Prediction (3*sigma band width)', {'
fontsize':24})
    plt.ylabel('Last cycle', {'fontsize':16})
    plt.xlabel('Cycles elapsed', {'fontsize':16})
    plt.xlim(0,3.5e5)
    plt.ylim(2e5,3.5e5)
    plt.legend(loc='best')
    plt.show()

    plt.figure(figsize = (20,10))
```

```
    plt.title('Likelihood of valid set in predicted distribution', {'fontsize
':24})
    plt.ylabel('PDF', {'fontsize':16})
    plt.xlabel('Cycles elapsed', {'fontsize':16})
    plt.plot(N_last_valid_window[n1:n2],pdf_valid[n1:n2])
    plt.xlim(0,3.5e5)
    plt.ylim(0,100)
    plt.show()

    j+=1
```

png

png

...

6.3. Test set

```
# Identifying the partial sequences that belong to the same complete sequence
n_test = []
for i in range(len(N_last_test)):
    if N_last_test[i] != N_last_test[i-1]:
        n_test.append(i)

mu_pred_test = np.asanyarray(mu_pred_test)

sigma_pred_test = np.asarray(sigma_pred_test)

N_last_test_window = np.asarray(N_last_test_window)

pdf_test = pdf(y_test_norm,mu_pred_test,sigma_pred_test)

j=0

for i in range(len(n_test)-1):
    # Input sequence plot
    plt.figure(figsize = (20,10))
    for seq in X_gen_complete[:500]:
        seq = np.asarray(seq).swapaxes(1,0)
        plt.plot(seq[1], seq[0], color='grey',alpha=0.1)

    seq_test = np.asarray(X_real_complete[j]).swapaxes(1,0)
    plt.plot(seq_test[1], seq_test[0], color='b')
    plt.title("Strain evolution test sample vs training set", {'fontsize':24}
)
    plt.xlabel('Cycles elapsed',{'fontsize':20})
    plt.ylabel('Strain (e-04) [-]',{'fontsize':20})
    plt.xlim(0,3.5e5)
    plt.ylim(0,8)
    plt.show()

    # Predictions plot
    n1 = n_test[i]
    n2 = n_test[i+1]
    plt.figure(figsize = (20,10))
```

```
    plt.plot(N_last_test_window[n1:n2], mu_pred_test[n1:n2] * max_y + N_last_
test_window[n1:n2]
             , label = 'Predicted mean', color = 'b', alpha=0.8)
    plt.plot(N_last_test_window[n1:n2], N_last_test[n1:n2]
             , label = 'Test', color = '#ffa500')
    plt.fill_between(N_last_test_window[n1:n2]
                     , (mu_pred_test[n1:n2] + 3 * sigma_pred_test[n1:n2]) * m
ax_y + N_last_test_window[n1:n2]
                     , (mu_pred_test[n1:n2] - 3 * sigma_pred_test[n1:n2]) * m
ax_y + N_last_test_window[n1:n2]
                     , color = 'B', alpha=0.3)
    plt.title('Test ground truth vs RNN Prediction (3*sigma band width)', {'f
ontsize':24})
    plt.ylabel('Last cycle', {'fontsize':16})
    plt.xlabel('Cycles elapsed', {'fontsize':16})
    plt.xlim(0,3.5e5)
    plt.ylim(2e5,3.5e5)
    plt.legend(loc='best')
    plt.show()

    plt.figure(figsize = (20,10))
    plt.title('Likelihood of test set in predicted distribution', {'fontsize'
:24})
    plt.ylabel('PDF', {'fontsize':16})
    plt.xlabel('Cycles elapsed', {'fontsize':16})
    plt.plot(N_last_test_window[n1:n2],pdf_test[n1:n2])
    plt.xlim(0,3.5e5)
    plt.ylim(0,100)
    plt.show()

    j+=1
```

png

png

...