UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE CIENCIAS

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

Algoritmos Basados en Metaheurísticas para la Optimización de Búsqueda de Hiperparámetros en Algoritmos de Machine Learning

Autor: *Miguel Angel Oviedo Rodriguez*

Asesor: *José Manuel Castillo Cara*

Lima - Perú

2019

Resumen

Entrenar modelos requieren de configuraciones óptimas de hiperparámetros así como de algoritmos de machine learning adecuados para obtener resultados fiables que se puedan aplicar a casos nuevos. En el diseño de un modelo se siguen varios pasos bien definidos; en lo que compete a este trabajo se enfoca en la fase de *tuning* la cual se realiza el proceso de optimización. Para ello, se emplean tres técnicas: *grid search* que prueba todas las posibles combinaciones, *randomize search* que toma una muestra aleatoria del conjunto de hiperparámetros y retorna la mejor configuración de dicha muestra, y *evolutive search* que emplea algoritmos evolutivos. El conjunto de hiperparámetros que resulten serán evaluados en tres algoritmos supervisados: *Decision Tree*, *Support Vector Machine* y *k-Neighbors Nearst* cuyos resultados serán comparados en base al tiempo, CPU y memoria RAM.
Por último, se emplea la librería de python especializada en machine learning, *sklearn*, y otras asociadas a la ciencia de datos.

PALABRAS CLAVE: *Machine Learning, Algoritmos Evolutivos, Estimación de Algoritmos Distribuidos, Búsqueda de Hiperparámetros, Optimización de Hiperparámetros, python-sklearn, python-deap.*

Índice general

Índice de figuras

Índice de tablas

Índice de Código

Índice de Acrónimos

GPS	Sistema de Posicionamiento Global
IPS	Sistema de Posicionamiento de Interiores
SML	Aprendizaje de Máquinas Supervisado
EA	Algoritmos Evolutivos
RSSI	Indicador de Fuerza de Señal Recibida
MSE	Error Cuadrático Medio
SVM	Máquina de Soporte Vectorial
SVC	Máquina de Soporte Clasificador
SVR	Máquina de Soporte Regresor
k-NN	*k* Vecinos más cercanos
CART	Árboles de Clasificación y Regresión

Agradecimientos

Gracias a mi familia y amigos que me han apoyado a lo largo de mi etapa universitaria ayudándome a superar los obtáculos y mejorar como persona.

Un especial agradecimiento al profesor Manuel Castillo que siempre ha mostrado disposición de apoyarme en el desarrollo de este seminario, sus observaciones y críticas que me han ayudado a mejorar como estudiante y profesional.

Capítulo 1

Introducción

En este capítulo se presenta una visión general de la importancia del machine learning, sus aplicaciones en la actualidad y el porqué es tan importante encontrar métodos de optimización de hiperparámetros. También se presentan los objetivos y la estructura principal del mismo para una mejor orientación del lector.

1.1. Motivación

En la década pasada se dio el gran salto de las comunicaciones, el uso masivo del Internet y la telefonía móvil cambiaron la forma de ver e interactuar con mundo. Se desarrollaron tecnologías impensables que ayudaron a mejorar la calidad de vida de la gente (aunque también hayan aspectos negativos) por la cual muchas personas fascinadas se han dedicado a continuar desarrollando nuevas teorías, técnicas y metodologías que contribuyan al avance de la informática y las comunicaciones. Dichos avances han abierto la puerta a la teoría de la inteligencia artificial, que allá por las décadas del sesenta al ochenta se formularon como conceptos matemáticos sin posibilidad de aplicación, y que hace unos años atrás han dado, creo yo, el siguiente gran paso de la revolución tecnológica.

La inteligencia artificial es tan importante hoy en día que está presente en casi todas (por no decir en todas) las ramas de las ciencias: en medicina, ya se puede predecir enfermedades como la neumonía o el cáncer con mayor precisión [1]; en astronomía, obtener mejores imágenes del universo con redes

generativas [2]; en biología, reconstrucción del genoma humano con más eficiencia que los métodos tradicionales [3], solo por citar algunos.

El pilar fundamental en el campo del Machine Learning es la predicción con el mayor grado de exactitud posible. Para llegar a ello se deben pasar por varios procesos que empiezan por entender el conjunto de datos a analizar, el filtrado o tratamiento de los mismos para luego encontrar patrones o características que brinden información útil y con ello evaluar en algún modelo adecuado en la fase de entrenamiento. Por último se implementan una serie de técnicas para mejorar el desempeño del modelo optimizando sus hiperparámetros. Este último paso es muy importante porque hace que el modelo sea confiable y escalable, pero se torna una tarea difícil si se tienen muchas posibles combinaciones de éstas. Por ello, existen diversas técnicas que puedan buscar una configuración óptima del espacio de hiperparámetros el cual es tema de interés del presente trabajo.

1.2. Objetivos

1.2.1. Objetivo General

El principal objetivo de esta investigación es realizar un análisis comparativo entre técnicas de búsqueda convencionales y algoritmos evolutivos en la optimización de hiperparámetros, para posteriormente aplicarlo en resolver nuevos problemas.

1.2.2. Objetivos Específicos

- Estudiar las metaheurísticas como estrategias de optimización con un enfoque en los algoritmos evolutivos.
- Demostrar cómo los algoritmos evolutivos pueden optimizar hiperparámetros con menor costro computacional que con los métodos tradicionales.

- Evaluar el desempeño de algortimos supervisados *k-NN*, *SVM* y *DT* con el conjunto de hiperparámetros generados por los algoritmos evolutivos.
- Desarrollar un grado de conocimientos en técnicas y métodos de Machine Learning para solucionar problemas nuevos.

1.3. Estructura del Seminario

El seminario tiene la siguiente estructura, que se detalla a continuación para la mejor orientación del lector.

1. **Introducción:**
   En este capítulo introductorio se presenta una visión general acerca de Machine Learning y su importante contribución a diversas ciencias. También se comenta brevemente el problema de la optmización. Por último, se plantean los objetivos.
2. **Estado del Arte:**
   En este capítulo se da a conocer algunas investigaciones relacionadas al Machine Learning en general, y la optimización de hiperparámetros en específico.
3. **Hiperparámetros y Algoritmos:**
   En este capítulo se detalla las bases teóricas del funcionamiento de los hiperparámetros, como se desenvuelven los EA (Evolutive Algorithms); también se detallan los modelos de prueba con los posibles hiperparámetros. Finalmente, se incluye una especificación a detalle del entorno de desarrollo y pruebas del presente trabajo.
4. **Minería de Datos:**
   En este capítulo se realizará un análisis de los datos recolectados, y la experimentación con los modelos detallados en el capítulo anterior. También se expone en detalle las funciones y los algoritmos que se utilizaron en la experimentación.

5. **Optimización y Resultados:**

   En este capítulo se detallan los resultados de la optimización de hiperparámetros para los algoritmos DT, SVM y *k*-NN con cada una de las técnicas de búsqueda estudiadas; así como el análisis comparativo de su rendimiento computacional.

6. **Conclusiones y Trabajos a Futuro:**

   Finalmente, se exponen las conclusiones de este estudio y se plantean como posible enfoque la optimzación de hiperparámetros con metaheurísticas evolutivas como siguiente paso en la investigación del tema.

Capítulo 2

Estado del Arte

Actualmente el campo de la inteligencia artificial, en especial, del machine learning se ha expandido tanto que prácticamente abarca todas áreas profesionales como las finanzas, medicina, biología, ingenierías, música e incluso los videojuegos.
Motivados por los impresionantes resultados que se han logrado, se están desarrollando grandes investigaciones como las que se presentan a continuación.

2.1. Trabajos relacionados al machine learning

En esta sección se describen tres trabajos de investigación muy interesantes sobre machine learning orientados a mejorar los pronósticos en la detección de tumores cerebrales usando algoritmos de clasificación; predicción de reacciones adversas de medicamentos en pacientes; y en el área de deep learning una interesante investigación sobre transformaciones de imágenes con redes generativas adversarias.

2.1.1 **Unpaired Image-to-Image Translationusing Cycle-Consistent Adversarial Networks.** [4]

Cycle-GAN es una arquitectura que utilizan dos Redes Generativas Adversarias (GAN) con aprendizaje supervisado para la transformación de imágenes de un estilo a otro. El algoritmo realiza un mapeo a un conjunto de imágenes de entrada en la cual aprende y transforma

en su equivalente al estilo de un conjunto de salida. A su vez las imágenes obtenidas se someten a una GAN inversa de manera que ésta las trasforme a su estilo original y se compruebe que pertenece a ese conjunto. En pocas palabras el objetivo de esta Cycle-GAN es que aprenda hacer un mapero $G : X \rightarrow Y$ de tal manera que la que la distribución de imágenes desde $G(X)$ sea indistinguible para el conjunto Y y otra donde $F : Y \rightarrow X$ se obtenga $F(G(X)) \approx X$ que sea aceptable teniendo en cuenta que puede haber una pérdida de autenticidad. Los resultados muestran como se modifican imágenes de, por ejemplo, un paisaje de verano invierno a invierno, de un caballo a una zebra, cambiar estilos de cuadros de arte por otros como se muestra en la Figura 2.1.

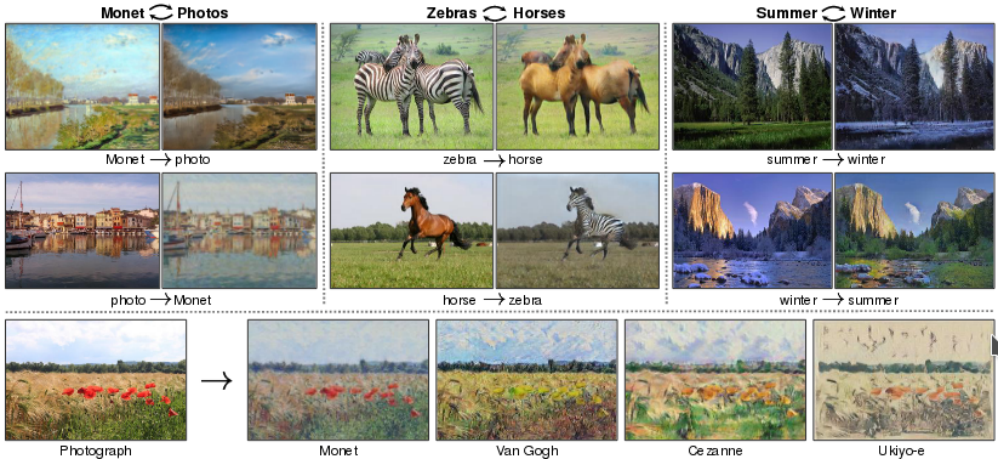

FIGURA 2.1: Las redes generativas adversarias pueden traducir imágenes de un tipo a otro y viceversa.

2.1.2 **Machine learning on adverse drug reactions for pharmacovigilance.** [5] En este trabajo se desarrolló una investigación sobre la predicción de las reacciones adversas a los medicamentos, siendo éste un problema que puede resultar en mortalidad, morbilidad y costos de salud sustanciales. Se han utilizado muchos métodos convencionales de machine learning como el SVM para predecir los efectos secundarios de los medicamentos posteriores a la comercialización. Sin embargo, debido a las complejas estructuras químicas de ciertos medicamentos y la naturaleza no lineal y desequilibrada de los datos biológicos, es posible que no se detecten

algunos efectos secundarios. Por otro lado, los estudios de investigación de descubrimiento de fármacos demostraron que el deep learning obtiene mejores predicciones que los métodos de machine learning, por ello en esta investigación aplicaron los enfoques de deep learning sin supervisión para predecir las ADR (Adverse Drugs Reaction) personalizadas y la reutilización de medicamentos. Los resultados muestran una predicción más precisa de los efectos secundarios y la reposición de medicamentos.

2.1.3 **Differentiation of glioblastoma from solitary brain metastases using radiomic machine-learning classifiers** [6]

El objetivo de este estudio fue identificar el clasificador de machine learning para diferenciar glioblastoma (GBM) de metástasis cerebrales solitarias (MET) antes de una operación. Cuatrocientos doce pacientes con tumores cerebrales solitarios (242 GBM y 170 MET cerebrales solitarios) se dividieron en conjuntos de entrenamiento (n = 227) y de prueba (n = 185). La extracción de características radiológicas se realizó con el software PyRadiomics. Con el conjunto de entrenamiento, se evaluaron doce métodos de selección de características y siete métodos de clasificación para construir clasificadores de machine learning de radiomías favorables. El rendimiento de los clasificadores se evaluó utilizando el área media bajo la curva (AUC) y la desviación estándar relativa en el percentil (RSD). En la fase de entrenamiento, trece clasificadores tuvieron resultados predictivos favorables (AUC $\leq$ 0.95 y RSD $\leq$ 6). En la fase de prueba, el análisis de la curva de características operativas del receptor (ROC) reveló que el algoritmo SVM más el Least Absolute Shrinkage and Selection Operator (LASSO, por sus siglas en inglés) tuvieron la eficacia de predicción más alta. Además, el rendimiento clínico del mejor clasificador fue superior a los neurorradiólogos en precisión, sensibilidad y especificidad. Según los resultados de esta investigación el empleo del algoritmo de SVM podría ayudar al neuroradiólogo a diferenciar el GBM del cerebro solitario MET preoperatorio.

En la Figura 2.2 se muestra los procedimientos seguidos. Después de la extracción de las características de los datos, se seleccionaron las que son relevantes para su posterior análisis. Se combinaron el esquema de selección de características múltiples y los clasificadores, y se seleccionaron modelos favorables con la ayuda de la validación cruzada en la fase de entrenamiento. En la fase de pruebas, el mejor modelo se identificó en comparación con la patología, y luego se comparó el rendimiento del mejor modelo con dos neurorradiólogos.

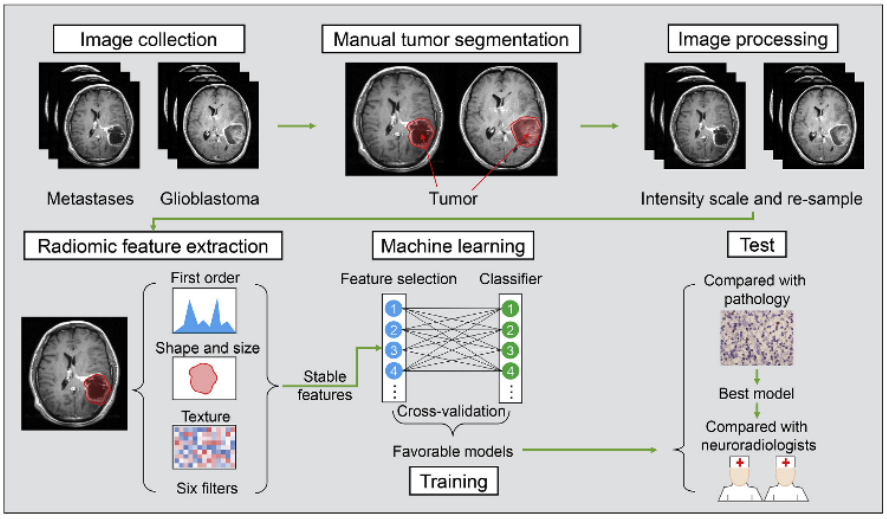

FIGURA 2.2: Flujo del análisis radiológico.

2.2. Trabajos relacionados con la Optimización de hiperparámetros

En esta sección se describen investigaciones de relativas a la optimización de hiperparámetros orientados a mejorar la calidad de software de sistemas; optimización en la predicción de procesos de negocios; y por último, una aplicación que mejora los resultados obtenidos a partir de sensores de gas.

2.2.1 **Optimización de Hiperparámetros de Algoritmos de machine learning Usados para Análisis de Calidad del Software Desarrollado en IBM RPG.** [7]

Esta investigación tuvo como objetivo encontrar métodos para ajustar

los algoritmos de machine learning para obtener el máximo desempeño que redunde en el análisis de datos de calidad del software desarrollado en IBM RPG en el departamento de sistemas, área de desarrollo del banco Agrobanco en Lima, Perú. Se emplearon algoritmos como Random Forest, *k*-NN , DT y SVM. Los resultados mostraron que para el algoritmo Random Forest los parámetros más importantes son: '*mtry*' (número de variables muestreadas aleatoriamente como candidatos en cada división.) y '*ntree*' (número de árboles) juegan un rol importante, se logró hallar valores óptimos fruto del proceso de afinamiento. También se pudo optimizar el mejor ajuste para la combinación los parámetros '*minsplit*' (división mínima) , '*maxdepth*' (máxima profundidad) , '*mtry*' y '*depth*' (profundidad del árbol).

2.2.2 **Genetic Algorithms for Hyperparameter Optimization in Predictive Business Process Monitoring.** [8]

Este trabajo de investigación presenta un framework para el monitoreo del proceso predictivo que reúne una gama de técnicas en la cual incluyen algoritmos genéticos, cada una con un conjunto asociado de hiperparámetros. Se presenta una mejora a un framework básico en el cual incorpora dos niveles de abstracción: uno con el objetivo de proporcionar una evaluación de cada configuración en un conjunto de seguimiento de validación y el segundo para desarrollar una población de configuraciones con el fin de identificar las mejores. El framework propuesto y los algoritmos de optimización de hiperparámetros se han evaluado en dos conjuntos de datos de la vida real y se han comparado con los enfoques de trabajos previos para el monitoreo predictivo de procesos de negocios. Los resultados demuestran la escalabilidad del enfoque y su capacidad para identificar configuraciones de framework precisas y confiables.

2.2.3 **On the optimization of the support vector machine regression**

hyperparameters setting for gas sensors array applications [9]

Se ha demostrado que la Support Vector Regression (SVR, por sus siglas en inglés) es más precisa en comparación con otras técnicas de machine learning que se utilizan comúnmente para aplicaciones de arrays de sensores químicos. Sin embargo, el rendimiento de la SVR depende en gran medida de la selección de sus hiperparámetros. La mayoría de las veces, los investigadores en este campo confían en los métodos de búsqueda exhaustiva para encontrar los valores adecuados de los hiperparámetros de la SVR al minimizar el error de predicción de validación cruzada. Este método no es una solución práctica debido al gran dominio de posibles valores de parámetros, que se ve agravado por la falta de conocimiento previo sobre los datos. Debido a ello en este trabajo se realizó una investigación para la optimización de los hiperparámetros de la SVR combinando el algoritmo SVR con el algoritmo de búsqueda de patrón generalizado (GPS) como una alternativa más rápida para determinar estos hiperparámetros. Los resultados demostraron que dicho algoritmo ofrece resultados similares a otros más complicados, como los algoritmos genéticos, la optimización bayesiana o la optimización de enjambres de partículas.

2.3. Conclusiones

Como comentamos líneas arriba, el campo de machine learning se ha extendido a muchas áreas y ha aportado valiosos resultados que han mejorado la calidad de vida.

La optimización de hiperparámetros ayudan a obtener mejores resultados con mayor rendimiento que métodos de fuerza bruta como la búsqueda exhaustiva.

El lo que compete mi trabajo es realizar un análisis comparativo del comportamiento de tres métodos de optimización de hiperparámetros sobre modelos de machine learning.

Capítulo 3

Hiperparámetros y Algoritmos

3.1. Introducción

En machine learning se distinguen dos tipos de parámetros, los que se fijan previamente y los que un modelo entrenado genera.
Los parámetros propiamente dichos se obtienen de acuerdo a los resultados del entrenamiento de un modelo. Por ejemplo, en las redes neuronales, es el mismo algoritmos el que determina los pesos de conexiones entre capas luego de un entrenamiento y según ello se obtienen resultados.

3.2. Hiperparámetros

Los hiperparámetros son aquel conjunto de parámetros que va a configurar un entrenamiento de algún modelo. Como es sabido en este campo de la inteligencia artificial, existen muchos modelos supervisados y cada uno de ellos posee una cantidad distinta de hiperparámetros que se pueden fijar.
Por ejemplo, en el algoritmo de clasificación *k*-NN un hiperparámetro es *k*, que hace referencia a la cantidad de ejemplos más cercanos que determinarán el grupo al que pertenece un dato. Una mala elección de *k* puede hacer que el entrenamiento caíga en underfitting u overfitting. Otro conocido ejemplo orientado a redes neuronales es el descenso del gradiente, en el cual se fija como hiperparámetro el radio de aprendizaje α y se evalúa la velocidad de convergencia del algoritmo. En la figura 3.1 se observa dos configuraciones de dicho hiperparámetro.

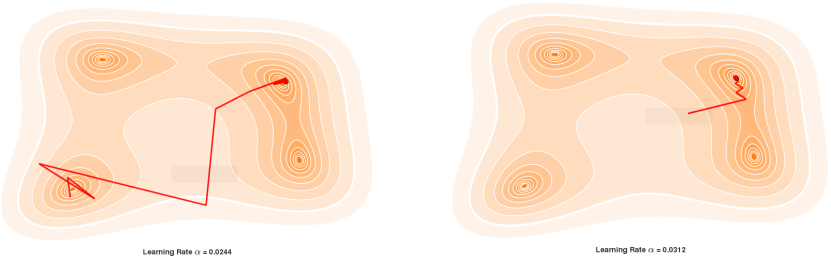

FIGURA 3.1: Configuración del radio de aprendizaje en el gradiente descentente. Para $\alpha = 0.0244$ el algoritmo se queda en un bucle infinito (diverge). Para $\alpha = 0.0312$ el algoritmo converge. Fuente: *www.benfrederickson.com/numerical-optimization/*

3.3. Tunning

Como comentamos en el capítulo uno, se realizan varios pasos para construir un modelo de machine learning. Después de haber entrenado un modelo puede que no hayamos obtenido buenas predicciones o que nuestra precisión no es la mínima deseada por lo que deberemos volver a entrenar nuestro modelo con una nueva configuración de hiperparámetros. Dicho proceso se realiza en la fase de *Tunning* o "sintonización". El objetivo es optimizar dicha configuración tal que nuestro modelo sea capaz de predecir resultados que estén acorde con el conjunto de datos de pruebas.

Sea por ejemplo el algoritmo **Alg1**, que tiene como espacio de hiperparámetros a **w**, **x**, **y**, **z**; con 3, 8, 10, 10 posibles valores respectivamente. Se tiene $3 \times 8 \times 10 \times 10 = 2400$ posíbles combinaciones.

3.3.1. Búsqueda Exhaustiva:

También conocido como *Grid Search*. Esta búsqueda prueba cada combinación de hiperparámetros y lo evalúa en el modelo. El patrón que se sigue aquí es similar a la de una rejilla, donde todos los valores se colocan en forma de una matriz. Cada conjunto de parámetros se toma en consideración y se anota la precisión. Una vez que se evalúan todas las combinaciones posibles, el modelo con el conjunto de parámetros que dan la mayor precisión se considera el mejor. En la Figura 3.2 se da una representación en la cual cada

punto (hiperparámetro) se toma en cuenta para la búsqueda.
La desventaja es que al ser una búsqueda del tipo fuerza bruta, puede consumir gran cantidad de tiempo y recursos, por lo que se debe emplear con criterio.

Para el algoritmo **Alg1**, GridSearch realizará 2400 validaciones cruzadas.

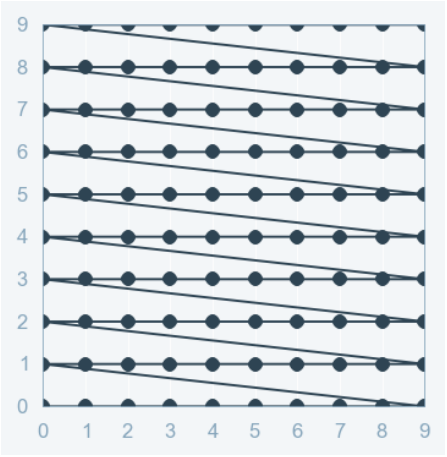

FIGURA 3.2: Representación visual de Grid Search
Fuente: *https://medium.com/*

3.3.2. Búsqueda Aleatoria:

La búsqueda aleatoria o *Random Search* es una técnica en la que se utilizan combinaciones aleatorias de hiperparámetros para encontrar una posible solución óptima para el modelo construido. Para optimizar con la búsqueda aleatoria, la función se evalúa en un número de configuraciones aleatorias en el espacio de parámetros. Las posibilidades de encontrar el parámetro óptimo son comparativamente más altas en la búsqueda aleatoria con respecto al *Grid Search*, debido al patrón de búsqueda aleatorio donde el modelo podría terminar entrenándose en los parámetros optimizados. La búsqueda aleatoria funciona mejor para datos de dimensiones inferiores, ya que el tiempo necesario para encontrar el conjunto correcto es menor con menos número de iteraciones.
Para el algoritmo **Alg1**, *Random Search* realizará menos de 40 validaciones cruzadas, pero no necesariamente encontrará la mejor configuración en

el espacio de hiperparámetros ya que depende del conjunto aleatorio de hiperparámetros que seleccione. En la Figura 3.3 se da una representación en la cual cada punto (hiperparámetro) es seleccionado de manera aleatoria.

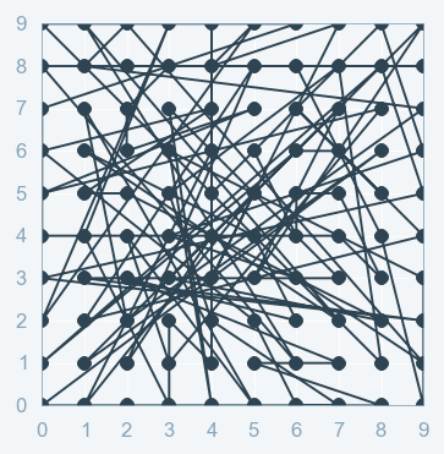

FIGURA 3.3: Representación visual de Random Search

Fuente: *https://medium.com/*

3.4. Metaheurística de Búsqueda Evolutiva

El término heurístico se refiere a procedimientos que tratan de aportar soluciones a un problema con un buen rendimiento, en lo referente a la calidad de las soluciones y a los recursos empleados. Por otro lado, las metaheurísticas son estrategias diseñadas para buscar, generar o seleccionar procedimientos heurísticos que puedan proporcionar una buena solución a un problema de optimización.

Dentro de las metaherurísticas de búsqueda están las evolutivas. Éstas establecen estrategias para conducir la evolución en el espacio de búsqueda de conjuntos de soluciones (usualmente llamados poblaciones) con la intención de acercarse a la solución óptima con sus elementos. El aspecto fundamental de las metaheurísticas evolutivas consiste en la interacción entre los miembros de la población frente a las búsqueda que se guían por la información de soluciones individuales.

Las diferentes metaheurísticas evolutivas se distinguen por la forma en que

combinan la información proporcionada por los elementos de la población para hacerla evolucionar mediante la obtención de nuevas soluciones.

3.4.1. Algoritmo Evolutivo

Los algoritmos evolutivos son métodos de optimización y búsqueda de soluciones basados en los postulados de la evolución biológica. En ellos se mantiene un conjunto de entidades que representan posibles soluciones, las cuales se mezclan, y compiten entre sí, de tal manera que las más aptas son capaces de prevalecer a lo largo del tiempo, evolucionando hacia mejores soluciones cada vez.
Se presentan las siguientes definiciones:

Individuo. Es la unidad representativa de una entidad que evolucionará al óptimo.
En el caso de **Alg1**, se define como un vector de tamaño cuatro, $i = [0, 2, 1, 4]$, que es la represntación de una de las posibilidades de **[w, x, y, z]**.

Gen. Es la mínima representación de una característica de un individuo.
Para **Alg1**, sería uno de los cuatro hiperparámetros, por ejemplo **y** $\in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$, que tiene diez posibles valores.

Población. Es el conjunto de individuos que serán evaluados.
Para **Alg1**, si se elige una población de tamaño tres, por ejemplo sería: $\{[0, 2, 3, 4]; [1, 5, 1, 8]; [0, 7, 9, 2]\}$. Que comprende un espacio muestral de tres individuos.

Cruce. Este proceso consiste en transmitir los genes de una generación a la siguiente.
Para **Alg1** es generar nuevos individuos que hereden algunos genes de sus progenitores.

Mutación. Este proceso consiste en la variación de algún gen en la cadena genética del individuo.
Para **Alg1** es el cambio de un determinado gen aleatoriamente en algún(os) individuo(s) de la población.

Evaluación. Este proceso consiste en calcular o estimar cuan óptimo (fitness) es cada individuo de la población.
Para **Alg1** es obtener el precisión del algoritmo con los parámetros del individuo.

Selección. Este proceso, como su nombre indica consiste en la separación del top de individuos más óptimos para la siguiente generación.
Para **Alg1** corresponde a la selección de individuos de la nueva generación que ofrezcan mejor precisión.

3.4.2. Modelo del Algoritmo Evolutivo

Definimos I para denotar a un espacio arbitrario de individuos, $a \in I$, y $F : I \to \Re$ para denotar a la función fitness de algún individuo; usando μ y λ como tamaños de población padre y heredera. Tenemos $P(t) = [a_1(t), ..., a_\mu(t)] \in I^\mu$, que caracteriza a una población en la generación t en el espacio muestral de individuos.
La *selección*, *mutación* y *cruce* se describen como los operadores $s : I^\lambda \to I^\mu$, $m : I^k \to I^\lambda$, y $x : I^\mu \to I^k$ respectivamente, que toman como conjunto de parámetros a Θ_s, Θ_m y Θ_x para la transformación de la población. Se tiene la implementación en Algoritmo 1.

Donde:

- $a*$ es el mejor individuo encontrado.
- $P*$ es la mejor población encontrada.
- l es el criterio de parada que tiene como conjunto de parámetros a Θ_l.

Algorithm 1 Modelo del algoritmo Evolutivo

Require: μ, λ, Θ_l, Θ_x, Θ_m, Θ_s
Ensure: a*, P*
1: $t \rightarrow 1$
2: $P(t) \rightarrow\ initialize(\mu)$
3: $F(t) \rightarrow\ evaluate(\mu)$
4: **while** $l(P(t), \Theta_l)$ **do**
5: $\quad P'(t) \leftarrow\ mate(P(t), \Theta_x)$
6: $\quad P''(t) \leftarrow\ mutate(P'(t), \Theta_m)$
7: $\quad F(t) \leftarrow\ evaluate(P''(t), \lambda)$
8: $\quad P(t+1) \leftarrow\ select(P''(t), F(t), \mu, \Theta_s)$
9: $\quad t \leftarrow\ t+1$
10: **end while**

Se entiende que $P(t)$, $P'(t)$ y $P''(t)$ tienen como población a μ, λ y k individuos respectivamente tras los procesos de las líneas 5,6,7 del Algoritmo 1.

Este proceso se visualiza también en la Figura 3.4 se tiene el proceso secuencial que rige el algoritmo evolutivo.

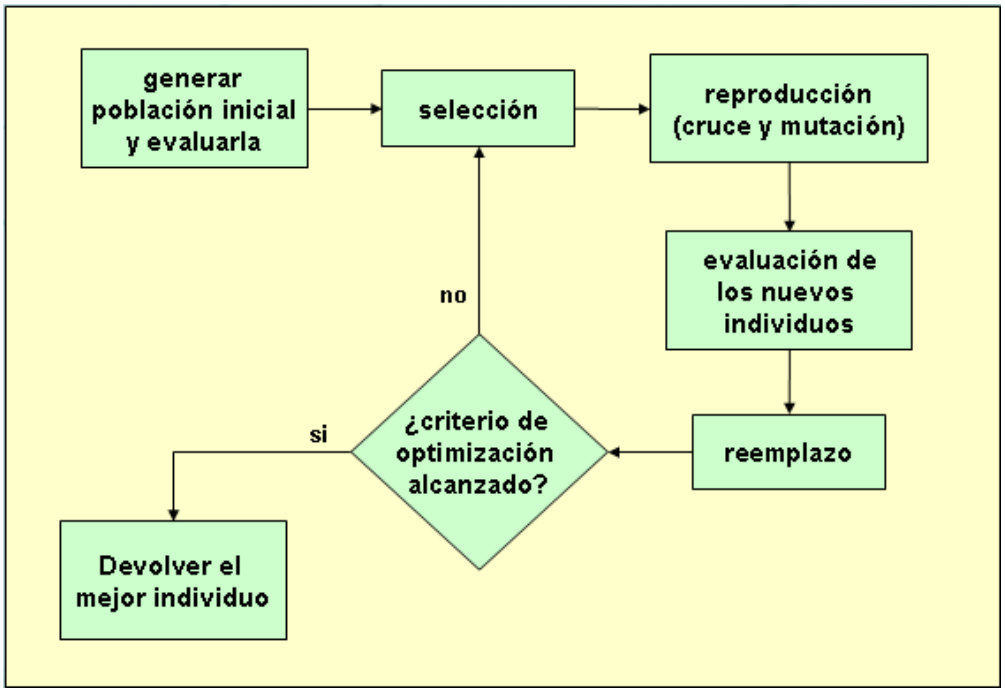

FIGURA 3.4: Diagrama representativo del Algoritmo Evolutivo.

Fuente: *https://sites.google.com/site/inteligenciacompyredes*

La Figura 3.4 es una representación ejemplar al caso de **Alg1** expuesto anteriormente.

3.5. Modelos de Prueba de Machine Learning

Este trabajo de investigación se emplean tres algoritmos supervisados de clasificación: Decision Tree, Support Vector Machine y *k*-Nearset Neighbors. Cada uno tiene un espacio de hiperparámetros que serán evaluados en el capítulo cinco. A continuación se exponen brevemente sus características.

3.5.1. Decision Tree

Decision Tree o árboles de decisión son un modelo predictivo para pasar de las observaciones sobre un elemento (ramas) a conclusiones sobre el valor objetivo del elemento (hojas).
El algoritmo básico para la construcción de árboles de decisión llamado Iterative Dichotomiser 3 (ID3), que emplea una búsqueda de arriba hacia abajo a través del espacio de posibles ramas sin retroceso. ID3 fue sucedido por una serie de algoritmos, para ser hoy el algoritmo optimizado CART. En la Figura 3.5 cada nodo interno corresponde a cada una de las variables de entrada; hay ramificaciones hacia los nodos hijos para cada uno de los posibles valores de cada variable de entrada, así cada hoja representa un valor de la variable de destino dado los valores de las variables de entrada representadas por el camino de la raíz a la hoja [10].

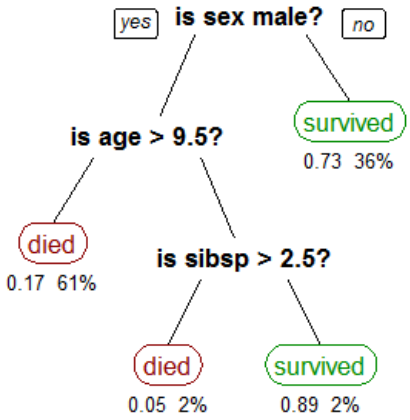

FIGURA 3.5: Ejemplo de árbol de decisión que muestra la supervivencia de los pasajeros del Titanic.
Fuente: *https://en.wikipedia.org*

Formalmente, $x_i \in \Re^n$ y $y \in \Re^1$, el árbol de decisión se divide recursivamente de manera que se agrupan muestran con las mismas etiquetas, así sea Q la data en el nodo m; para cada división $\theta = (j, t_m)$, j es una característica y t_m el umbral y se tiene las ramas $Q_l(\theta)$ y $Q_r(\theta)$ [10].

$$Q_l(\theta) = (x, y) \mid x_j <= t_m \ \& \ Q_r(\theta) = Q \setminus Q_l(\theta)$$

Classification Tree

Estos modelos de árbol de decisión tienen como variable objetivo a un conjunto discreto de valores. En estas estructuras de árbol, las hojas representan las etiquetas de clase y las ramas representan conjunciones de características que conducen a esas etiquetas de clase.
Si un objetivo es un resultado de clasificación tomando valores $0, 1, \ldots, k-1$, para el nodo m, R_m representa una región con N_m observaciones, se tiene la proporción de las observaciones de la clase k en el nodo m [10, 11]:

$$P_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$$

Hiperparámetros de Decision Tree

criterion. Cadena de texto, es el criterio de calidad de split, *gini* como criterio de impuridad y *entropy* como ganancia de información.

splitter. Estrategia de particionado, puede ser *random*, *best random split* y *best split*.

max_depth. Número entero, es la máxima profundidad del árbol.

min_samples_split. Número entero, mínimo de ejemplares requeridos para hacer un split en un nodo terminal.

min_samples_leaf. Número entero, mínimo de ejemplares requeridos para abrir una nodo hoja.

max_features. Número de características para la búsqueda del mejor split. puede ser $sqrt(n\_features)$, $log2(n\_features)$, un número entero o un real como porcentaje de $|max\_features \times n\_features|$.

class_weight. Vector de pesos de cada clase, se puede representar como un diccionario de forma clase:peso. También balanceado (en función del porcentaje de data) o None para pesos equitativos.

presort. Valor booleano, puede ser Verdadero para preordenar la data.

3.5.2. Support Vector Machine

Support Vector Machine son un conjunto de algoritmos de aprendizaje supervisado desarrollados por Vladimir Vapnik, están estrechamente relacionados con problemas de Clasificación, Support Vector Classifier (SVC), y de Regresión Support Vector Regressor (SVR).
SVM es un modelo que representa los puntos en el espacio, es decir construye

en conjunto de hiperplanos en un espacio de dimensionalidad muy alta e incluso infinita cuyo objetivo es separar las salidas lo máximo posible de forma óptima.
La manera más simple de realizar la separación es mediante una línea recta, un plano recto o un hiperplano N-dimensional, pero en la realidad éste es un caso idóneo y muy poco probable. SVM trata con más de dos variables predictoras, curvas no lineales de separación, casos donde los conjuntos de datos no pueden ser completamente separados, cuya solución es el uso de una función Kernel [12].

Support Vector Classifier

El modelo SVC maneja la separación de características usando las funciones no lineales kernel para mapear la data en un espacio donde un hiperplano no será suficiente. Estas funciones kernel no dependen de la dimensión del espacio por ello son usadas para transformar la data a un espacio de características de más dimensiones para hacer posible la separación lineal más amplio posible mediante el uso de hiperplanos de separación como vectores de dos puntos más cercanos entre ellos, como se observa en la Figura 3.6 [13].

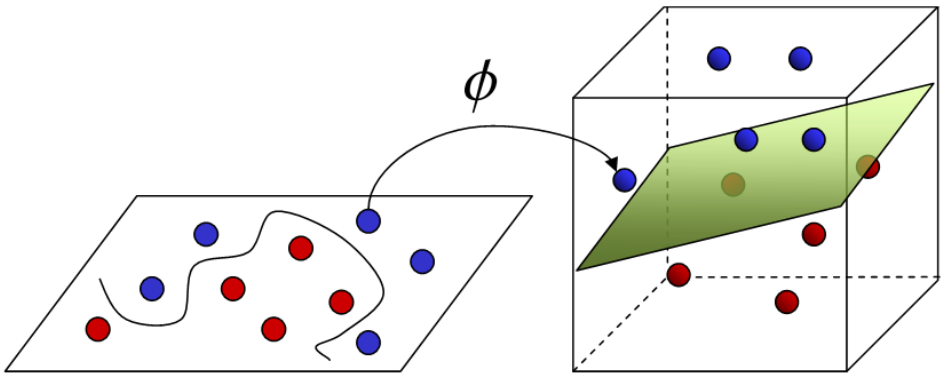

FIGURA 3.6: Transformación del espacio con la función Kernel ϕ.
Fuente: *https://www.jeremyjordan.me*

Hiperparámetros del SVM

kernel. La función kernel proyecta las características de entrada de mayor dimensión, para mapear el espacio de entradas X a un nuevo espacio de características de mayor dimensionalidad (Hilbert) [12]:

$$F = \{\varphi(x) \mid x \in X\}$$

$$Si\ x = x_1, x_2, \ldots, x_n \rightarrow \varphi(x) = \{\varphi_1(x), \varphi_2(x), \ldots, \varphi_n(x)\}$$

Se tienen los siguientes tipos de función kernel:

- Función polinomial homogénea (poly): $ker(x_i, x_j) = (x_i \cdot x_j)$
- Función gaussiana de base radial (rbf): $\exp(-\gamma \parallel x - x' \parallel^2)$. γ is specified by keyword gamma, must be greater than 0.
- Función sigmoid (sigmoid): $ker(x_i, x_j) = tanh(x_i \cdot x_j - \theta)$

gamma. Número real, usado en función kernel poly, sigmoid y poly. Por defecto $1/num_{caractersticas}$

degree. Número entero, usado en la función kernel poly.

coef0. Número real, usado en función kernel sigmoid y poly como término independiente.

C. Número Real, es el parámetro de penalización del término de error.

probability Valor booleano, para habilitar probabilidad de estimación.

shrinking. Valor booleano, para habilitar heurística shrinking.

tol. Número Real, es la tolerancia para la detención del criterio usado.

class weight. Es un diccionario o lista de pesos, puede ser balanceado (según frecuencia), equitativo (todos peso uno), o personalizado.

3.5.3. *k*- Nearest Neighbors

En el reconocimiento de patrones, el método *k*-Nearest Neighbors (*k*-NN) es un modelo no paramétrico utilizado en clasificación y regresión. La entrada consta de los k ejemplos de entrenamiento más cercanos en el espacio de características de entrada. La salida depende de si *k*-NN se utiliza para la clasificación o la regresión. En la Figura 3.7 se observa que el color a asignar al elemento x dependerá de sus vecinos más cercanos.

Tanto para la clasificación como para la regresión, puede ser útil asignar peso a los vecinos, de modo que los vecinos más cercanos contribuyan más al promedio que los más lejanos (funciones de distancia). Por ejemplo, un esquema de ponderación común consiste en dar a cada vecino un peso de $1/d$, donde d es la distancia al vecino [14]. Para éste modelo se requiere que las características estén estandarizadas para que su valor real no influencie en los pesos o distancias, además es necesario elegir el valor óptimo para k, que aumentará considerablemente la precisión en la predicción.

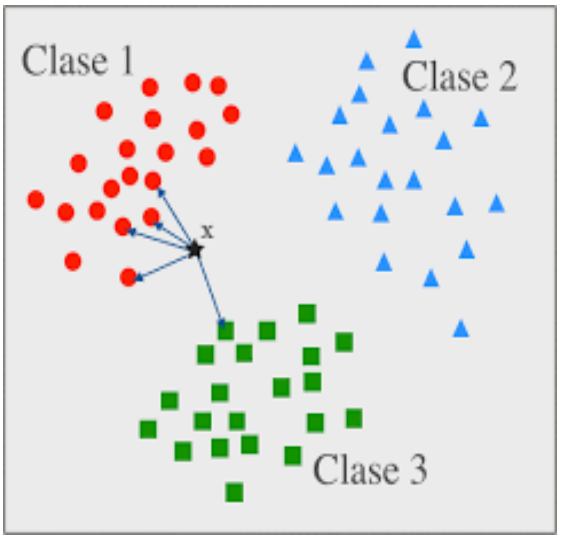

FIGURA 3.7: Selección de salida en función de *k*-Nearest Neighbors. Fuente: *https://www.tamps.cinvestav.mx*

El valor de k se puede presumir inspeccionando los datos, un valor de k grande es más preciso, ya que reduce el ruido general, pero no hay certeza; la validación cruzada es otra forma de determinar retrospectivamente un buen valor k usando subconjuntos independientes [15].

En la clasificación k-NN

Este algoritmo de clasificación almacena todos los casos disponibles y clasifica nuevos casos basados en una medida de similitud, funciones de distancia.
Un objeto es clasificado por voto mayoritario de sus *k* vecinos, siendo el que más se repite la salida donde k es un entero positivo pequeño [16].

Hiperparámetros de *k*-NN

Métricas de Distancia. Para determinar la relevancia de unas características respecto a otras se utilizan las funciones de distancia, que pueden ser para válidas para variables categóricas y continuas.

- Distancia Euclidiana (v. continua) $\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$
- Distancia Manhatan (v. continua) $\sum_{i=1}^{k} \mid x_i - y_i \mid$
- Distancia Minkowski (v. continua) $(\sum_{i=1}^{k}(\mid x_i - y_i \mid)^q)^{\frac{1}{q}}$
- Distancia Hamming (v. categórica) $\sum_{i=1}^{k}\lceil x_i - y_i \rceil$

n_neighbors. Número entero, indica el número de vecinos que se deben tomar en cuenta.

weights. Vector de números reales, que indiquen la influencia de cada característica, puede también ser uniforme (todos peso uno), o por distancia (cuán cerca se encuentran).

algorithm. Es el posible algoritmo búsqueda usado para encontrar los vecinos más cercanos, pueden ser ball_tree, kd_tree o brute.

leaf_size. Número entero, es la cantidad de hojas que se pasan como datos a los algoritmos kd_tree o ball_tree. El valor óptimo depende de la naturaleza del problema

p. Número entero, es el parámetro de potencia para la métrica de distancia de Minkowski, $p = 1$ implica Manhatan y $p = 2$ implica Euclidean.

3.6. Entorno de desarrollo y pruebas

Para modelar los algoritmos de este capítulo se usa el lenguaje python. Además, para computación científica y el modelado de aplicaciones, se necesitan librerías adicionales (packages) fuera del estandar que ofrece python; se usaron los siguientes:

- **Librería numpy 1.13.1:** NUMeric Python es un paquete para computación científica, necesario para el manejo de vectores, matrices y diferentes operaciones de álgebra lineal, más en página oficial (numpy.org).
- **Librería scipy 0.19.1:** SCIentific Python es un paquete para computación científica, maneja una colección de algoritmos numéricos y toolboxes específicos para procesamiento de señales, optimización, estadísticas entre otros, más en página oficial (scipy.org).
- **Librería pandas 0.20.3:** Pandas es un paquete de python necesario para el manejo de estructuras de datos con alto rendimiento. Este paquete es fundamental para la extracción y manipulación de datos, más en página oficial (pandas.pydata.org).
- **Librería matplotlib 2.0.2:** PLOTting LIBrary es un paquete de publicación de imágenes 2D en diferentes formatos y con soporte para 3D, es una

poderosa herramienta que facilita la visualización de datos y resultados para el presente trabajo, más en página oficial (matplotlib.org).

- **Librería seaborn 0.8.2:** Seaborn es una librería para realizar gráficos estadísticos en python, construido sobre matplotlib e integrado con PyData, nos pérmite generar gráficos muy atractivos., más en página oficial (seaborn.pydata.org).

- **Librería scikit-learn 0.19.0:** El paquete sklearn contien todos los modelos descritos con anterioridad, permite un manejo simple y eficiente para la minería y análisis de datos. Es fundamental para la extracción de características, preparación de datos, modelado y visualización de resultados, más en página oficial (scikit-learn.org).

- **Librería deap 1.0:** Es un framework para programar y elaborar modelos genéticos evolucionarios, además cuenta con la implementación de mecanísmos multiprocesos y beanchmark, más en página oficial (deap.readthedocs.io).

La implementación y desarrollo de pruebas se realizó en jupyter notebook con python3, en una laptop Intel Core i5-4200 con memoria RAM de 8GB en un entorno debian 9.9 Stretch de 64 bits.

Capítulo 4

Minería de Datos

En este capítulo se define el problema de localización de interiores utilizando dispositivos de Bluetooth; se presenta el conjunto de datos obtenido así como el procesos de extracción y preparación. Por último se seleccionan y entrenan en tres algoritmos supervisados de machine learning donde se analiza la precisión obtenida.

4.1. Definición del Problema

Hoy en día, existe un gran interés en desarrollar mecanismos precisos de localización en interiores inalámbricos que permitan la implementación de muchos servicios orientados al consumidor. Los mecanismos inalámbricos de localización en interiores basados en la intensidad de señal (RSSI) recibida se están explorando ampliamente.
El presente trabajo extrae información de localización de interiores por señales RSSI por medio de receptores bluetooth.

Características del interior

El interior experimental mide $9.3\ m.$ de largo y $6.3\ m.$ de ancho, en el cual se distribuyen quince sectores de $1m^2$, con separación de $0.5m$ vertical y horizontalmente, además se dejó una franja de $1.15\ m.$ al borde del interior, como se muestra en Figura 4.1.

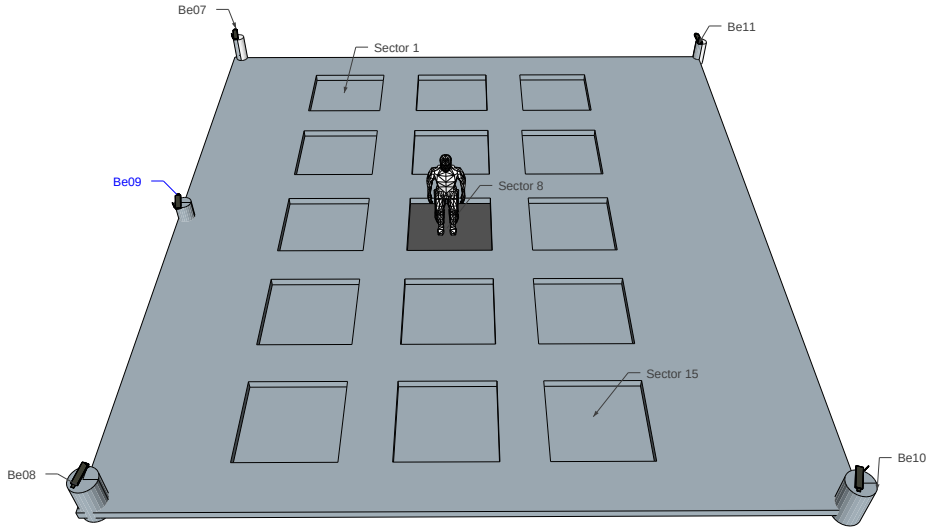

FIGURA 4.1: Localización de beacons en interior del ambiente de prueba. Fuente: *Empirical Study of the Transmission Power.*

Lo que se busca es aplicar los modelos de aprendizaje automático de clasificación sobre el volumen de información obtenida de la fase de experimentación, con el objetivo de medir el nivel precisión de los algoritmos y poder aumentarlo con la configuración más óptima de hiperparámetros.

Sensores bluetooth Emisores y Receptores

En el proceso de experimentación se tiene un dispositivo emisor de frecuencias RSSI, baliza BLE 4.0, y un dispositivo receptor, Raspberry PI 2; con los cuales se generan todo el volumen de datos necesario para las pruebas.
La baliza de la Figura 4.3, que se mide bajo siete niveles de potencia Tx y para este trabajo se utiliza la potencia intermedia.

Beacons Bluetooth (Emisor) Se tienen 5 balizas bluetooth BLE 4.0, que cuenta con siete niveles de potencia; siendo Tx= {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07} en orden descendente de potencia. El beacon de la Figura 4.3 se sintonizará con el Receptor a una frecuencia intermedia Tx=0x04 para generar los datos RSSI.

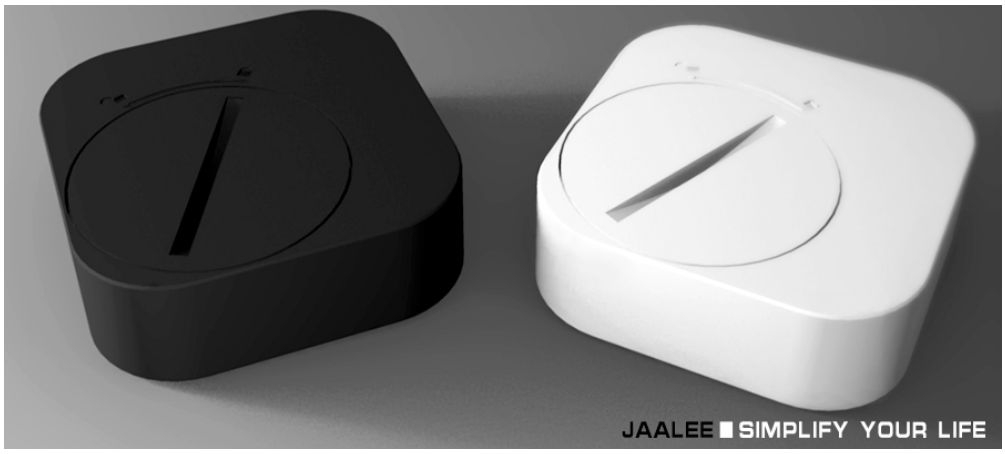

FIGURA 4.2: Beacon Bluetooth. Fuente: *Jaalee inc.*

Raspberry Bluetooth (Receptor): El dispositivo receptor es un Raspberry PI 2, equipado con una antena usb BLE 4.0, con el cual se generan quince archivos (uno por cada sector en el interior), cada uno con las señales RSSI emitidas por los cinco beacons, identificados por sus direcciones MAC ahora nombrados como Be07, Be08, Be09, Be10, Be11 [17].

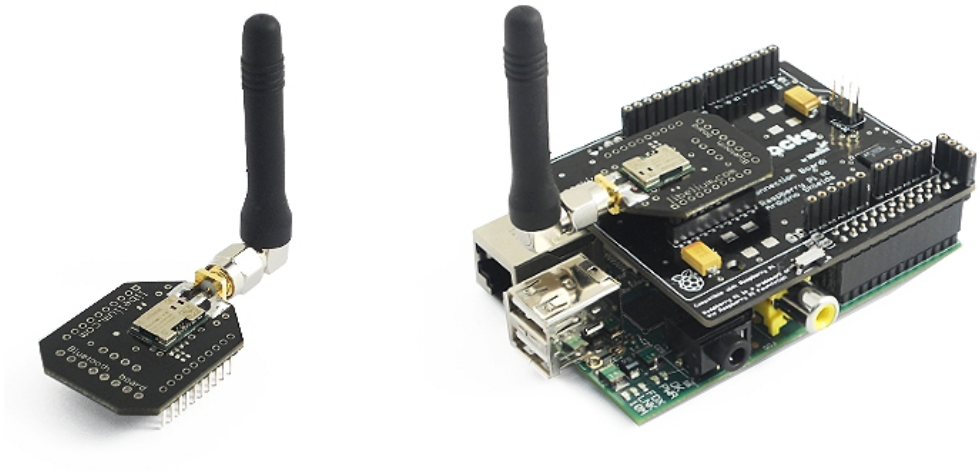

FIGURA 4.3: Raspberry Bluetooth. Fuente: *www.cooking-hacks.com*

4.2. Preparación de los Datos

Se concentró la data de los quince sectores formando uno sólo que contiene todos los vectores de RSSI de cada posición, no todos los beacons hacen el mismo número de muestras, por lo que se completó con la moda de cada

beacon. Obteniendo la distribución de datos de la Tabla 4.1, donde μ representa la media de valores RSSI por Beacon por cada Sector y σ su correspondiente desviación estandar.

		Be07		**Be08**		**Be09**		**Be10**		**Be11**	
Sector	**Data**	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
1	453	-62.7	4.2	-65.2	3.3	-59.7	0.8	-72.9	3.8	-69.2	3.7
2	449	-63.5	2.8	-60.2	0.7	-62.5	3.6	-73.9	6.1	-67.9	1.0
3	438	-64.9	3.4	-68.9	1.9	-73.3	3.6	-74.5	4.3	-73.4	4.8
4	436	-67.0	3.2	-66.0	3.9	-59.8	4.5	-71.7	3.5	-66.9	4.2
5	456	-63.7	2.0	-66.8	3.3	-66.1	5.4	-70.1	1.9	-70.0	4.0
6	480	-58.8	2.9	-68.7	4.5	-74.5	4.7	-72.4	2.9	-70.5	3.9
7	478	-74.7	2.4	-67.3	1.9	-66.7	4.4	-54.3	1.7	-67.8	3.5
8	439	-68.7	2.0	-59.7	5.0	-65.0	4.1	-68.1	3.1	-67.5	6.7
9	429	-71.7	4.9	-66.0	2.0	-71.1	1.9	-69.3	2.7	-74.4	4.3
10	644	-69.3	4.8	-64.3	3.5	-72.1	2.2	-70.0	4.2	-72.6	3.5
11	465	-70.6	4.2	-58.7	1.8	-67.8	2.5	-68.1	5.0	-67.6	3.8
12	437	-73.6	2.6	-64.7	4.6	-74.8	4.5	-75.8	2.0	-63.7	1.8
13	467	-68.2	3.3	-58.0	1.5	-69.7	2.4	-76.4	4.0	-67.3	3.0
14	536	-72.9	5.2	-58.4	4.1	-70.5	4.9	-72.7	3.0	-69.1	2.4
15	567	-70.7	2.0	-64.6	4.6	-66.9	3.8	-74.2	3.1	-63.8	6.2
Todos	7174	-68.2	5.6	-63.8	5.0	-68.2	5.9	-70.9	6.2	-68.8	5.0

TABLA 4.1: Distribución de datos por Sectores.

Para hacer pruebas de entrenamiento y validación se reordena la data aleatoriamente, separando el 80 % para entrenamiento de datos y 20 % para la validación.

4.3. Exploración de Datos

Coeficiente de Correlación: Determina si hay una relación lineal (1), inversamente lineal (-1) o no la hay (0). Los valores intermedios se interpretan como el nivel de dependencia lineal o inversamenta lineal, es considerable si es menor a -0.5 o mayor a 0.5.

FIGURA 4.4: Matriz de Correlación de Pearson de Frecuencias RRSI a frecuencia Tx=0x04. Fuente: *Elaboración Propia.*

Densidad de Distribución: Determina el comportamiento de las variables y su probabilidad de ocurrencia $N(\mu, \lambda^2)$.

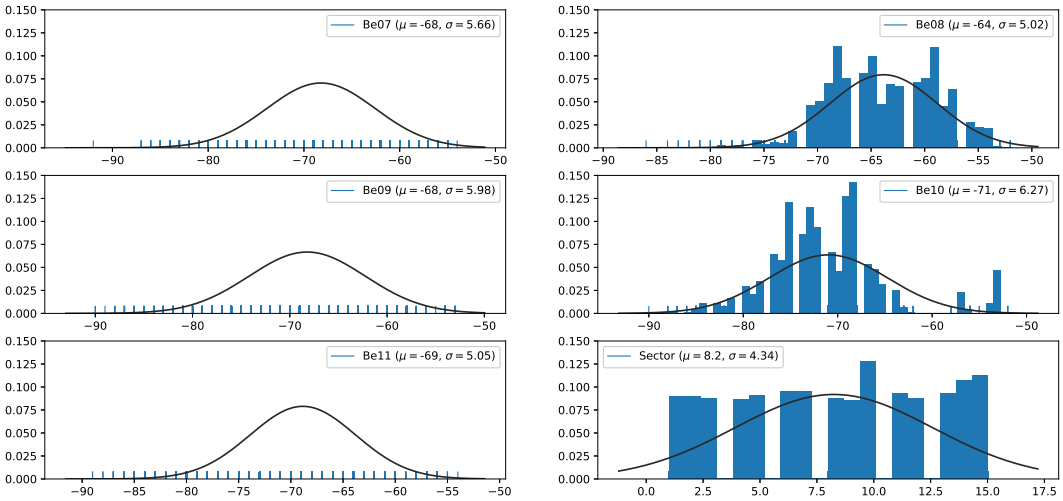

FIGURA 4.5: Densidad de distribución de datos por característica y sector. Fuente: *Elaboración Propia.*

En la Figura 4.4 se representa el tipo de relación entre las características (beacons) y el sector. Como es de notar, entre ellos tienen bajo coeficiente de correlación, lo que implica que no hay dependencia lineal ni inversamente lineal entre las características y el sector. En la Figura 4.5 se visualiza la distribución de frecuencias RSSI por cada Beacon (Be07, Be08, Be09, Be10, Be11) y la distribución de datos de los sectores.

4.4. Modelado de Algoritmos

La data una vez preparada y visualizada, podemos aplicar los modelos detallados anteriormente, siguiendo los siguientes pasos:

1. **La división de la data en Train & Test:**

   En Código 4.1 se tiene la implementación de la función prepareData, que recibe como parámetro el dataset, que es formato dataframe y el porcentaje de data para validación (20 % por defecto).

   Primero se separa en X todas las columnas menos la última (data de entrada), y en Y la última (data de salida). Se usa la función **train_test_split** para retornar dos pares de datos, [X, Y] para entrenar y para validar.

```
1 def prepareData(dataset, val_size=0.20):
2   vector = dataset.values
3   X = vector[:, 0:dataset.shape[1] - 1].astype(float)
4   Y = vector[:, dataset.shape[1] - 1]
5   return train_test_split(X, Y, test_size=val_size)
6 X_train, X_test, Y_train, Y_test = prepareData(dataset, 0.1)
```

CÓDIGO 4.1: Separación de data para entrenamiento y validación

2. **Selección de Algoritmos:**

   En Código 4.2 se agregan los modelos de Clasificación de prueba, con score de determinación de precisión $accuracy$.

```
models = []
models.append(('Árbol de Decisión', DecisionTreeClassifier()))
models.append(('k Vecinos + Cercanos', KNeighborsClassifier()))
models.append(('Máquina de Soporte Vectorial', SVC()))
```

CÓDIGO 4.2: Selección de los modelos de prueba

3. **Evaluación de Algoritmos:**

   En Código 4.3 se tiene la implementación de la función de evaluación de modelos por score. Esta función recibe como parámetros *models*, un arreglo de modelos de la forma ('$nombre$', $modelo$), *numfolds*, indíca el número de splits en la data para la validación cruzada, *scoring* y los datos X input y Y output.

   La función **cross_val_score** obtiene los resultados de las $numfolds$ evaluaciones del modelo, cada una con un kfold diferente de entre los datos.

```
def evaluateModels(models, num_folds, scoring, X_train, Y_train):
  results = {}
  for name, model in models:
    kfold = KFold(n_splits=num_folds)
    cv_results = cross_val_score
      (model, X_train, Y_train, cv=kfold, scoring=scoring)
    results[name] = cv_results
  return pd.DataFrame(data=results)
resultados = evaluateModels(models, 10, 'accuracy', X_train, Y_train)
```

CÓDIGO 4.3: Función de evaluación de modelos de machine learning

   El retorno de esta función es un dataframe de los $numfolds$ resultados obtenidos para el score seleccionado como estimador para los modelos de prueba.

4. **Comparación de Algoritmos:**

   En Código 4.4 se tiene la función de comparación que recibe como parámetro el dataframe de resultados obtenidos de la evaluación de modelos.

```
1 def compareResults(results):
2   new_result = pd.concat([results.mean(), results.std()], axis=1)
3   new_result.columns = ["Precisión media", "Desviación Estandar"]
4   sns.boxplot(data=results, orient='h')
5   pyplot.xlim(xmin=0.7, xmax=1)
6   pyplot.show()
7   return new_result
```

CÓDIGO 4.4: Función visual para comparación de resultados

En Figura 4.6 se muestra el boxplot con los resultados obtenidos en la evaluación de algoritmos en el rango [$ymin = 0.7$, $ymax = 1$].

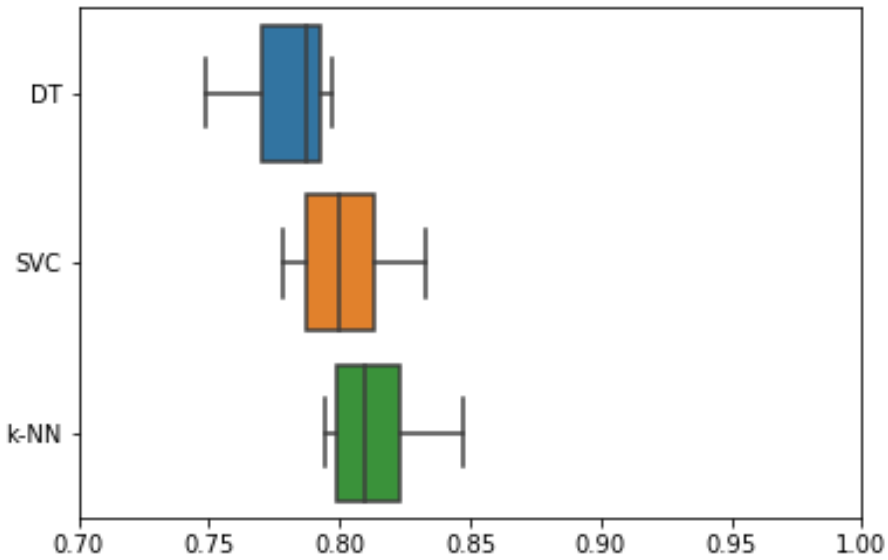

FIGURA 4.6: Comparación de resultados de precisión de los modelos evaluados. Fuente: *Elaboración Propia.*

4. **Validación de Resultados:**

	Precisión	Desviación
DT	0.781196	0.015881
SVC	0.801210	0.017264
kNN	0.813509	0.018294

TABLA 4.2: Precisión media y desviación estándar para los tres algoritmos evaluados

En Código 4.5 se tiene la función de validación, que debe arrojar resultados de score muy próximos a los obtenidos en evaluación. Esto indica que no hay overfiting. El retorno de la función es un dataframe con los score: accuracy, MSE y matriz de confución.

```
1  def validateModels(models, X_train, Y_train, X_validation, Y_validation):
2    results = {'Modelo':[], 'Precisión':[], 'Error':[], 'cf_matrix':[]}
3    for name, model in models:
4      model.fit(X_train, Y_train)
5      Y_predicted = model.predict(X_validation)
6      results['Modelo'].append(name)
7      results['cf_matrix'].append(
8        confusion_matrix(y_pred=Y_predicted, y_true=Y_validation))
9      results['Precisión'].append(
10       accuracy_score(y_pred=Y_predicted, y_true=Y_validation))
11     results['Error'].append(
12       mean_squared_error(y_pred=Y_predicted, y_true=Y_validation))
13   return pd.DataFrame(data=results)
```

CÓDIGO 4.5: Función de validación de resultados

6. **Optimización de Data:**

   Para que los algoritmos tengan mejor precisión y rendimiento, siempre se puede mejorar la data de entrada, así por ejemplo si se tienen valores muy grandes y poco influenciables; afectará en el desempeño.

   Normalización: Cuando la data no tiene una ditribución normal (forma de campana). En Código 4.6 se visualiza la función de normalización de modelos.

```
def normalizeData(models):
  pipelines = []
  for name, model in models:
    pipelines.append(('Norm' + name, Pipeline
      ([('Normalizer', Normalizer()), (name, model)])))
  return pipelines
```

CÓDIGO 4.6: Función de Normalización de Algoritmos de ML

Estandarización: Cuando la data si tiene una ditribución normal (forma de campana).En Código 4.7 se visualiza la función de estandarización de modelos.

```
def standarizeData(models):
  pipelines = []
  for name, model in models:
    pipelines.append(('Scaled' + name, Pipeline
      ([('Scaler', StandardScaler()), (name, model)])))
  return pipelines
```

CÓDIGO 4.7: Función de Estandarización de Algoritmos de ML

Gracias a la visualización en Figura 4.5 sabemos que toda la data está distribuida normalmente, aunque con sesgos a los lados en algunos casos, por consiguiente valoraremos más la estandarización para nuestro volumen de datos.

Capítulo 5

Optimización y Resultados

En este capítulo se expone el punto principal de este seminario. Se presenta los resultados obtenidos de evaluar cada algoritmo supervisado con un conjunto de hiperparámetros generados con cada técnica de búsqueda estudiada; según ello se analizó y comparó la precisión, tiempo de ejecución y rendimiento computacional.

5.1. Introducción

Cada modelo de aprendizaje no siempre logra obtener una buena predicción para casos nuevos , pues para cada tipo de datos o distribuciones existen configuraciones de hiperparámetros que se ajustan más a cada problema.

Para hacerlo de forma óptima se implementa una clase de estimador de selección que recibe como entrada dos diccionarios, uno de los modelos y el otro de hiperparámetros con sus posibles valores. Esta clase incluye dos funciones, la función de búsqueda **fitSearch**, se muetra en Código 5.1; donde *Method_SearchCV* es el método para obtener la mejor configuración y ***kwargs* son los argumentos de dicho método.

```
1 def fitSearch(self, X, y, cv=KFold(n_splits=5), n_jobs=1, verbose=1,
2     scoring=None, **kwargs):
3   for key in self.keys:
4     print("Running Search for %s." % key)
5     start = time.time()
6     gs = Method_SearchCV(self.models[key], self.params[key], cv=cv,
7         n_jobs=n_jobs, verbose=verbose, scoring=scoring,
```

```
8            refit=refit, **kwargs)
9        self.grid_searches[key] = gs.fit(X, y)
10       end = time.time()
11       self.time_model[key] = [end-start]
```

CÓDIGO 5.1: Función de entrenamiento de búsqueda de hiperparámetros óptimos

La segunda función es **score**, se muestra en Código 5.2; donde se evalúan y muestran los resultados obtenidos en el la búsqueda.

```
1  def score(self):
2    for estimator in self.keys:
3      d = {}
4      d['.Accuracy'] =
5      self.grid_searches[estimator].cv_results_['mean_test_score']
6      d['.Error'] =
7      self.grid_searches[estimator].cv_results_['std_test_score']
8      df1 = pd.DataFrame(d)
9      d = self.grid_searches[estimator].cv_results_['params']
10     df2 = pd.DataFrame(list(d))
11     self.result[estimator] = pd.concat([df1, df2], axis=1).fillna(' ')
12   return pd.concat(self.result).fillna(' ')
```

CÓDIGO 5.2: Función de visualización de resultados de fitSearch

El número de candidatos a mejor configuración de hiperparámetros es el producto de las cantidades de posibles valores para cada hiperparámetro. Sea h_i un hiperparámetro en el espacio de hiperparámetros H, que puede tomar n_i posibles valores; el número de posibles candidatos es $\prod(n_i) \forall h_i \in H$.

5.2. Tuning de Decision Tree

Para la realización de las pruebas de tunning y búsqueda de la configuración más óptima de hiperparámetros para DT; se busca entre los siguientes posibles

valores permitidos:

(**dt1**) **class_weight:** balanced, None.

(**dt2**) **criterion:** entropy, gini.

(**dt3**) **max_features:** sqrt, log2, None.

(**dt4**) **min_samples_split:** 2, 4, 6, 8.

(**dt5**) **splitter:** best, random.

Este modelo tiene en total 96 candidatos a mejor configuración de hiperparámetros. A cinco splits para verificar eficiencia, hacen 480 validaciones cruzadas.

5.2.1. Búsqueda Exhaustiva

Tras la ejecución de la búsqueda exhaustiva se tiene:

Running Exhaustive Search for DT.

Fitting 5 folds for each of 96 candidates, totalling 480 fits

[Parallel(n_jobs=8)]: Done 480 out of 480 | elapsed: 2.7s finished

Incluyendo el tiempo de comparación y validación, le tomó a esta búsqueda 2.6660077571868896 segundos; los 5 mejores resultados se exponen en la Tabla 5.1.

		Hiperparámetros				
Precisión	**Desviación**	**dt1**	**dt2**	**dt3**	**dt4**	**dt5**
78.19	1.33	balanced	entropy		2	best
77.99	2.04	balanced	gini		2	best
77.76	0.70	balanced	entropy	sqrt	2	best
77.74	2.16	balanced	entropy		2	best
77.55	1.88		gini		2	best

TABLA 5.1: Top 5 configuraciones óptimas para DT con búsqueda exhaustiva.

5.2.2. Búsqueda Aleatoria

Tras la ejecución de la búsqueda aleatoria se tiene:

Running Randomized Search for DT.

Fitting 5 folds for each of 10 candidates, totalling 50 fits

[Parallel(n_jobs=8)]: Done 50 out of 50 | elapsed: 0.5s finished

Incluyendo el tiempo de comparación y validación, le tomó a esta búsqueda 0.5133950710296631 segundos; los 5 mejores resultados se exponen en la Tabla 5.2.

		Hiperparámetros				
Precisión	**Desviación**	**dt1**	**dt2**	**dt3**	**dt4**	**dt5**
77.66	0.88		gini	sqrt	2	best
77.53	0.85		entropy		4	best
77.17	1.40	balanced	gini		8	best
76.74	0.49	balanced	entropy		6	best
76.28	0.99	balanced	gini	log2	6	best

TABLA 5.2: Top 5 configuraciones óptimas para DT con búsqueda aleatoria.

5.2.3. Búsqueda Evolutiva

Tras la ejecución de la búsqueda evolutiva con población inicial de 10 individuos, probabilidad de cruce de 50 %, probabilidad de mutación de 20 % durante 9 generaciones se tiene:

Running Evolutive Search for DecisionTree.

Generación	Población	avg	min	max	std
0	10	0.719616	0.690011	0.75775	0.0261117
1	6	0.742168	0.693948	0.774315	0.0220669
2	5	0.748352	0.697884	0.774315	0.0269338
3	4	0.76121	0.70018	0.77858	0.0227953
4	6	0.774151	0.749221	0.78186	0.00901128
5	4	0.778858	0.774315	0.78186	0.00307462
6	6	0.779351	0.774807	0.78186	0.00219198
7	6	0.780695	0.778088	0.78186	0.00108659
8	6	0.781286	0.780384	0.78186	0.000592501
9	6	0.781204	0.77858	0.78186	0.000956364

El tiempo de búsqueda obtenido es 2.2804245948791504 segundos; donde se tiene el incremento del score a lo largo de las generaciones, en la generación 0 se evalúa la población inicial.

Sumando todas las poblaciones tenemos 49 individuos evaluados, empezando con una población inicial de 10, haciendo un total de 59 de las 96 evaluaciones posibles. Se puede observar que se obtiene mejores resultados conforme se crean nuevas generaciones llegando a 0.781204 en la última generación. Los resultados de la presición se muestran en la Tabla 5.3.

	Hiperparámetros				
Precisión	**dt1**	**dt2**	**dt3**	**dt4**	**dt5**
77.19		entropy		2	best
77.19		entropy		2	best
77.19	balanced	entropy		2	best
77.19		entropy		2	best

TABLA 5.3: Top 4 configuraciones óptimas para DT con búsqueda evolutiva.

Por otra parte, en la Figura 5.1 se muestran tres gráficas de caja y bigote en la que se describen el consumo computacional de cada búsqueda. Se

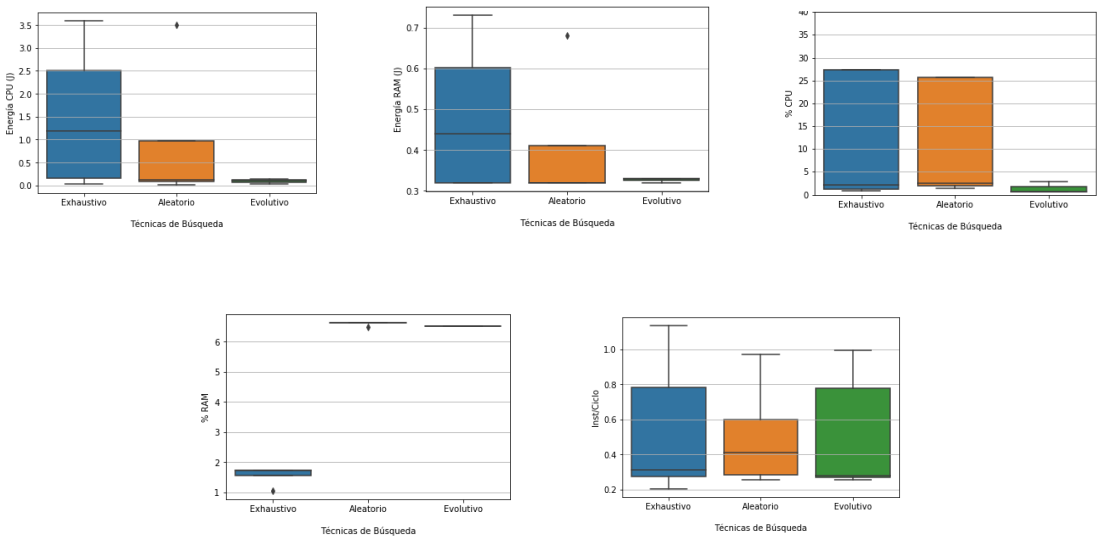

FIGURA 5.1: Comparación de energía consumida por el CPU, memoria RAM, %CPU, %RAM e instrucciones por ciclo para DT. Fuente: *Elaboración propia*

deduce rápidamente que con la búsqueda evolutiva el consumo de energía del procesador y la memoria son muy bajos, con valores menores a 0.5 J en cada muestra. Los dos restantes requieren de más energía llegando a valores extremos de 3.5 J en el caso del procesador con la búsqueda exhaustiva.

El porcentaje del consumo de procesador también sigue un patrón similar, pero en el caso de la memoria es opuesto.

Las instrucciones por ciclo muestran resultados comparables, concentrados entre 0.2 y 0.8.

5.3. Tuning de Support Vector Machine

Para la realización de las pruebas de sintonización y búsqueda de la configuración más óptima de hiperparámetros para support vector machine; se búsca entre los siguientes posibles valores permitidos:

(**svc1**) **C:** 0.5, 1, 1.5, 3.

(**svc2**) **decision_function_shape:** ovo, ovr.

(**svc3**) **degree:** 2, 3, 4, 5.

(**svc4**) **kernel:** linear, rbf, poly, sigmoid.

(**svc5**) **probability:** True, False.

(**svc6**) **shrinking:** True, False.

(**svc7**) **max_iter:** 300.

Este modelo tiene en total 512 candidatos a mejor configuración de hiperparámetros. A cinco splits, hacen 2560 validaciones cruzadas.

5.3.1. Búsqueda Exhaustiva

Tras la ejecución de la búsqueda exhaustiva se tiene:

Running Exhaustive Search for SVM.

Fitting 5 folds for each of 96 candidates, totalling 480 fits
[Parallel(n_jobs=8)]: Done 2560 out of 2560 | elapsed: 114.3min finished

Incluyendo el tiempo de comparación y validación, le tomó a esta búsqueda 6855.664493322372 segundos; los 5 mejores resultados se exponen en la Tabla 5.4.

		Hiperparámetros					
Precisión	**Desviación**	**svc1**	**svc2**	**svc3**	**svc4**	**svc5**	**svc6**
79.2	1.8	1.0	ovr	5	rbf	True	False
79.2	1.8	1.0	ovo	2	rbf	False	False
79.2	1.8	1.0	ovo	2	rbf	True	False
79.2	1.8	1.0	ovo	5	rbf	True	True
79.2	1.8	1.0	ovo	3	rbf	False	True

TABLA 5.4: Top 5 configuraciones óptimas para SVM con búsqueda exhaustiva.

5.3.2. Búsqueda Aleatoria

Tras la ejecución de la búsqueda aleatoria se tiene:

Running Randomized Search for SVM.

Fitting 5 folds for each of 10 candidates, totalling 50 fits

[Parallel(n_jobs=8)]: Done 50 out of 50 | elapsed: 1.4min finished

Incluyendo el tiempo de comparación y validación, le tomó a esta búsqueda 85.41693353652954 segundos; los 5 mejores resultados se exponen en la Tabla 5.5.

		Hiperparámetros					
Precisión	**Desviación**	**svc1**	**svc2**	**svc3**	**svc4**	**svc5**	**svc6**
78.60	1.69	1.0	ovr	5	rbf	False	True
78.55	1.59	1.0	ovr	2	rbf	False	False
78.12	1.64	1.0	ovr	2	rbf	True	True
72.04	1.62	1.0	ovo	4	rbf	False	False
72.04	1.62	1.0	ovo	5	rbf	True	False

TABLA 5.5: Top 5 configuraciones óptimas para SVM con búsqueda aleatoria.

5.3.3. Búsqueda Evolutiva

Tras la ejecución de la búsqueda evolutiva con población inicial de 20 individuos, probabilidad de cruce de 50 %, probabilidad de mutación de 20 % durante 10 generaciones se tiene:

Running Evolutive Search for SVM.

Generación	Población	avg	min	max	std
0	20	0.454043	0.0783992	0.78596	0.246309
1	10	0.638199	0.0783992	0.78596	0.206868
2	10	0.785673	0.781204	0.78596	0.00103586
3	15	0.78596	0.78596	0.78596	1.11022e-16
4	12	0.78596	0.78596	0.78596	1.11022e-16
5	10	0.730277	0.0783992	0.78596	0.173717
6	12	0.78596	0.78596	0.78596	1.11022e-16
7	13	0.76846	0.435952	0.78596	0.0762825
8	15	0.785936	0.785468	0.78596	0.000107239
9	9	0.78596	0.78596	0.78596	1.11022e-16
10	13	0.78268	0.720354	0.78596	0.0142985

El tiempo de búsqueda obtenido es 2362.6577773094177 segundos; donde se puede observar un incremento de precisión en las primeras cuatro generaciones, recae a 0.730277 en la quinta generación para luego mejorar hasta 0.78268 en la generación 10. En la novena generación se obtiene el mejor resultado. Sumando todas las poblaciones tenemos 139 individuos evaluados de entre 512 posibilidades, con el objetivo de búscar el mejor score. Los cinco mejores resultados se muestran en la Tabla 5.6.

	Hiperparámetros					
Precisión	**svc1**	**svc2**	**svc3**	**svc4**	**svc5**	**svc6**
78.6	1.5	ovr	2	rbf	True	True
78.6	1.5	ovr	2	rbf	True	True
78.6	1.5	ovo	2	rbf	True	False
78.6	1.5	ovr	2	rbf	True	False
78.6	1.5	ovr	2	rbf	False	False

TABLA 5.6: Top 5 configuraciones óptimas para SVM con búsqueda evolutiva.

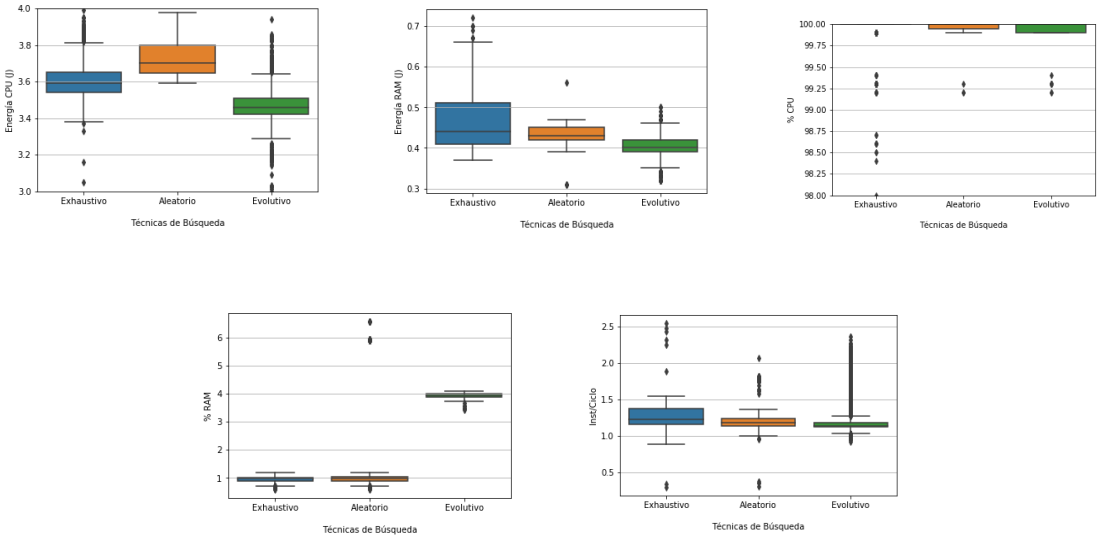

FIGURA 5.2: Comparación de energía consumida por el CPU, memoria RAM, %CPU, %RAM e instrucciones por ciclo para SVM.
Fuente: *Elaboración propia*

Por otra parte, en la Figura 5.2 se muestran tres gráficas que describen el consumo computacional de cada búsqueda. Para la SVM la búsqueda evolutiva también es superior en la eficiencia computacional. En el caso del consumo del procesador está concentrado entre 3.4 J y 3.6 J, mientras que el exhaustivo está por debajo de 3.6 J y el aleatorio entre 3.6 J y 3.8 J.

En el caso de la memoria, las tres búsquedas presentan un consumo similar de entre 0.4 J y 0.5 J. El porcentaje de uso de los núcleos del procesador es aprovechado al límite por todas búsquedas, presentando algunos valores atípicos de entre 98 % y 99 %. En cambio el porcentaje de memoria es muy bajo llegando a un 4 % como mucho en la búsqueda evolutiva. Por último, en las instrucciones por ciclo los valores se concentran entre 1 y 1.5. La búsqueda evolutiva muestra valores atípicos en el rango de 1.3 a 2.4 aproximadamente.

5.4. Tuning de *k*-Nearest Neighbors

Para la realización de las pruebas de sintonización y búsqueda de la configuración más óptima de hiperparámetros para el algoritmo *k*-NN; se busca entre los siguientes posibles valores permitidos:

(**knn1**) **algorithm:** ball_tree, kd_tree, brute.

(**knn2**) **leaf_size:** 10, 15, 20, 25, 30, 35, 40, 45, 50.

(**knn3**) **n_neighbors:** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

(**knn4**) **p:** 1, 2, 3, 4, 5.

(**knn5**) **weights:** uniform, distance.

Este modelo tiene en total 2700 candidatos a mejor configuración de hiperparámetros. A cinco splits para verificar eficiencia, hacen 13500 validaciones cruzadas.

5.4.1. Búsqueda Exhaustiva

Tras la ejecución de la búsqueda exhaustiva se tiene:

Running Exhaustive Search for *k*-NN.

Fitting 5 folds for each of 2700 candidates, totalling 13500 fits
[Parallel(n_jobs=8)]: Done 13500 out of 13500 | elapsed: 305.9min finished

Incluyendo el tiempo de comparación y validación, le tomó a esta búsqueda 18353.91120481491 segundos; los 5 mejores resultados se exponen en la Tabla 5.7.

		Hiperparámetros				
Precisión	**Desviación**	**knn1**	**knn2**	**knn3**	**knn4**	**knn5**
84.83	1.02	ball_tree	10	7	1	distance
84.83	1.02	ball_tree	15	7	1	distance
84.71	1.02	ball_tree	35	7	1	distance
84.71	1.02	ball_tree	30	7	1	distance
84.71	1.02	ball_tree	25	7	1	distance

TABLA 5.7: Top 5 configuraciones óptimas para *k*-NN con búsqueda exhaustiva.

5.4.2. Búsqueda Aleatoria

Tras la ejecución de la búsqueda aleatoria se tiene:

Running Randomized Search for *k*-NN.

Fitting 5 folds for each of 10 candidates, totalling 50 fits

[Parallel(n_jobs=8)]: Done 50 out of 50 | elapsed: 39.3s finished

Incluyendo el tiempo de comparación y validación, le tomó a esta búsqueda 39.31176447868347 segundos; los 5 mejores resultados se exponen en la Tabla 5.8.

		Hiperparámetros				
Precisión	**Desviación**	**knn1**	**knn2**	**knn3**	**knn4**	**knn5**
83.88	1.14	ball_tree	15	5	1	distance
81.94	0.79	ball_tree	50	10	1	uniform
81.96	0.20	kd_tree	45	5	2	3 distance
80.91	0.58	kd_tree	15	2	2	distance
80.63	0.51	kd_tree	40	3	3	distance

TABLA 5.8: Top 5 configuraciones óptimas para *k*-NN con búsqueda aleatoria.

5.4.3. Búsqueda Evolutiva

Tras la ejecución de la búsqueda evolutiva con población inicial de 25 individuos, probabilidad de cruce de 50 %, probabilidad de mutación de 20 % durante 12 generaciones se tiene:

Running Evolutive Search for *k*-NN.

Generación	Población	avg	min	max	std
0	25	0.814151	0.792849	0.842053	0.0153317
1	18	0.824182	0.792685	0.842053	0.0141514
2	11	0.834797	0.819419	0.842053	0.00663437
3	15	0.839587	0.828768	0.842053	0.00341954
4	22	0.840859	0.825652	0.842546	0.00324916
5	16	0.842093	0.841561	0.842546	0.000275545
6	14	0.841647	0.82926	0.842546	0.00260743
7	7	0.842467	0.842053	0.842546	0.000180387
8	12	0.840033	0.80023	0.842546	0.00906227
9	16	0.842546	0.842546	0.842546	1.11022e-16
10	13	0.842546	0.842546	0.842546	1.11022e-16
11	19	0.841384	0.813515	0.842546	0.00568883
12	8	0.842546	0.842546	0.842546	1.11022e-16

El tiempo de búsqueda obtenido es 136.8668167591095 segundos; donde se tiene el incremento del fitness precisión a lo largo de las generaciones llegando a 0.842546 en la generación 12. Los resultados del máximo score se muestran la Tabla 5.9.

Sumando todas las poblaciones tenemos 196 individuos evaluados de un total de 2700 posibilidades a lo largo de las 12 generaciones.

	Hiperparámetros				
Precisión	**knn1**	**knn2**	**knn3**	**knn4**	**knn5**
84.21	brute	10	6	1	distance
84.21	brute	15	6	1	uniform
84.21	brute	15	6	1	uniform
84.21	brute	10	6	1	distance
84.21	brute	10	6	1	distance

TABLA 5.9: Top 5 configuraciones óptimas para *k*-NN con búsqueda evolutiva.

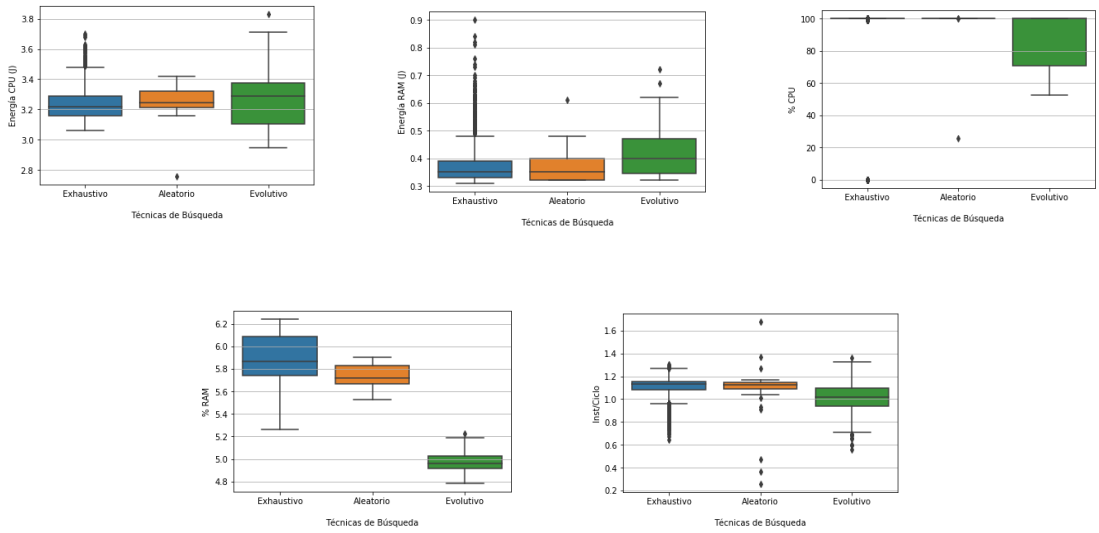

FIGURA 5.3: Comparación de energía consumida por el CPU, memoria RAM, %CPU, %RAM e instrucciones por ciclo para *k*-NN.
Fuente: *Elaboración propia*

La Figura 5.3 se muestran tres gráficas que describen el consumo computacional de cada búsqueda para el *k*-NN. El consumo computacional del procesador es relativamente similar, pero en la búsqueda evolutiva casi el 75 % de los registros están entre 3.0 J y 3.4 J, mientras que en los otros dos casi todos los datos se concentran entre 3.1 J y 3.3 J. Similar patrón figura para el consumo de la memoria RAM: los datos con la búsqueda evolutiva están más dispersos que en los otros dos.

En el porcentaje de uso del procesador, tanto la búsqueda evolutiva y aleatoria usan al máximo todos los núcleos, en cambio con la evolutiva hay un mayor rendimiento.

El mejor porcentaje de uso de la memoria es para la búsqueda evolutiva que consume en promedio el 5 %. En el caso de las instrucciones por ciclo se observa un patrón similar para los tres casos.

5.5. Resultados

En esta sección se expone mediante dos cuadros comparativos el desempeño de los modelos de búsqueda de configuración óptima de hiperparámetros

sobre los algoritmos de prueba. También se hace una comparación del coste computacional que consumen cada técnica de búsqueda con los algoritmos de machine learning.

Resultados de búsqueda

En la Tabla 5.10 se compara el top 3 de los resultados de precisión obtenidos en la búsqueda de hiperparámetros sobre de los algoritmos de prueba.

	Decision Tree (%)	**Support Vector Machine (%)**	***k*-Nearest Neighbors (%)**
Búsqueda Exhaustiva	78.24	79.20	84.83
	77.86	79.20	84.83
	77.69	79.20	84.83
Búsqueda Aleatoria	77.66	78.60	83.88
	75.53	78.55	81.94
	77.17	78.12	80.96
Búsqueda Evolutiva	77.68	78.60	84.21
	77.68	78.60	84.21
	77.68	78.60	84.21

TABLA 5.10: Comparación de resultados de precisión.

Para DT, la búsqueda exhaustiva obtiene el mayor score, siguiendo la búsqueda evolutiva, y con menor score la búsqueda aleatoria.
Para SVM, la búsqueda exhaustiva supera a la búsqueda evolutiva con una diferencia de 0.6 % aproximadamente. La búsqueda aleatoria es la que obtuvo menor score en este caso.
Para *k*-NN, la búsqueda exhaustiva supera ligeramente a la búsqueda evolutiva, pero hay una diferencia notable con respecto a la búsqueda aleatoria que es muy superada por la búsqueda evolutiva.
Con respecto a los modelos entrenados, el *k*-NN tiene un notable mayor score de aproximandamente 5 % con respecto al DT y SVM. Con respecto a las técnicas de búsqueda, la exhaustiva presenta mejor score, pero su desempeño es muy bajo con respecto a la evolutiva como se verá en los resultados siguientes.

Número de evaluaciones realizadas

En la Tabla 5.11 se tiene el cuadro comparativo del número de evaluaciones realizadas por las búsquedas de hiperparámetros sobre los algoritmos de prueba.

	Decision Tree (evals)	**Support Vector Machine (evals)**	***k*-Nearest Neighbors (evals)**
Búsqueda Exhaustiva	96	512	2700
Búsqueda Aleatoria	10	10	10
Búsqueda Evolutiva	59	139	196

TABLA 5.11: Comparación del número de evaluaciones.

El número máximo de evaluaciones o configuraciones posibles los hace el modelo de búsqueda exhaustiva; todos contra todos. La búsqueda aleatoria siempre evalua 10 posibles configuraciones aleatoriamente, mezclando lo mejor posible los datos. La búsqueda evolutiva realiza una menor cantidad de evaluaciones con respecto a la exhaustiva, esto porque a partir de una población inicial aleatoria se va combinando y mutando algunas características pasando las mejores a las siguientes generaciones, esto evita que se evalúen malos genes en el modelo. El número de evaluaciones será aleatorio, dependiendo de la población inicial y del número de generaciones. Con esto, el número de validaciones se redujo de 96 a 59 en el DT, 512 a 139 en el SVM, y de 2700 a solo 196 en el *k*-NN. A medida que el número de evaluaciones máximas aumenta, la diferencia entre el número de evaluaciones por el modelo de búsqueda evolutiva y exhaustivo se hace más grande.

Tiempos de búsqueda

En la Tabla 5.12 se muestran el cuadro comparativo de los tiempos de ejecución de los modelos de búsqueda sobre los algoritmos de prueba. El tiempo de ejecución está relacionado con el número de evaluaciones, la potencia

	Decision Tree (seg.)	**Support Vector Machine (seg.)**	***k*-Nearest Neighbors (seg.)**
Búsqueda Exhaustiva	2.6660077571868896	6855.664493322372	18353.9112048149
Búsqueda Aleatoria	0.5133950710296631	85.41693353652954	39.31176447868347
Búsqueda Evolutiva	2.2804245948791504	2362.6577773094177	136.8668167591095

TABLA 5.12: Comparación de tiempos de ejecución.

de CPU, RAM, la técnica de búsqueda y el modelo. Los tiempos obtenidos en segundos son el producto del tiempo de ejecución del algoritmo y el número de evaluaciones por modelo.

De los datos recolectados se infiere que DT es de bajo costo, mientras que el algoritmo SVM tiene un costo muy alto de recursos, con fines de prueba se limitó a 300 iteraciones.

El algoritmos *k*-NN también tiene un coste elevado pero muy por debajo del SVM en el caso de las búsquedas aleatoria y exhaustiva.

Según el análisis del score obtenido en la Tabla 5.11 y del tiempo de ejecución de la Tabla 5.12, este último algoritmo junto con la técnica de búsqueda evolutiva parecen ser los más adecuados para el conjunto de datos que estamos estudiando.

Coste computacional de cada algoritmo supervisado con respecto a una técnica de búsqueda

La Figura 5.4 muestra el coste computacional para la búsqueda exhaustiva. Se observa que el algortimo DT tiene menor coste debido a que tiempo de ejecución es mucho menor, aún así el consumo de energía de de la memoria es mayor que en SVM y *k*-NN. Estos dos últimos tienen tiempos de ejecución comparables y se oberva que el *k*-NN es superior al SVM tanto en coste de procesador como en RAM, pero en el porcentaje de uso de la RAM el *k*-NN es muy superior comparado con las otras dos.

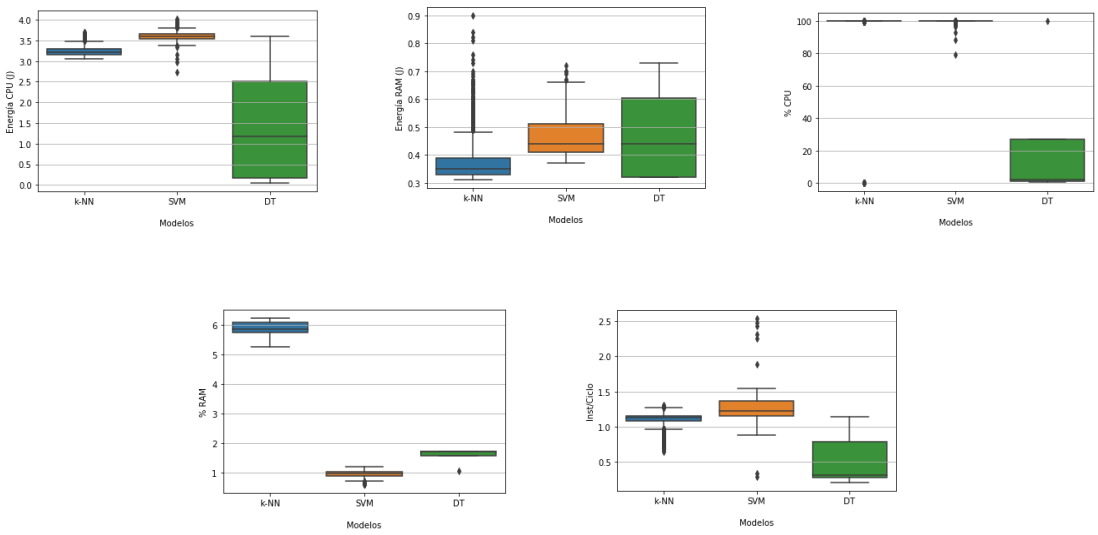

FIGURA 5.4: Comparación de energía consumida por el CPU, memoria RAM, %CPU, %RAM e instrucciones por ciclo para la búsqueda exhaustiva. Fuente: *Elaboración propia*

Con respecto al último gráfico, el algoritmo DT es el más lento ya que los resultados están a menos de una instrucción por ciclo, siendo el SVM el más rápido superando ligeramente al *k*-NN.

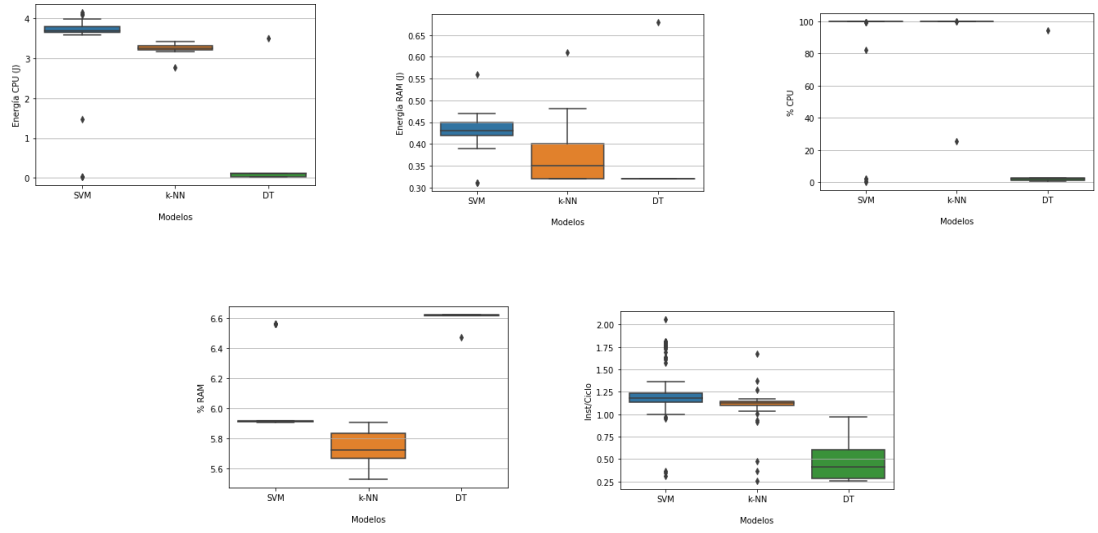

FIGURA 5.5: Comparación de energía consumida por el CPU, memoria RAM, %CPU, %RAM e instrucciones por ciclo para búsqueda aleatoria. Fuente: *Elaboración propia*

La Figura 5.5 muestra el coste computacional para la búsqueda aleatoria. De igual manera, el algortimo DT tiene un menor consumo de procesador (tanto en energía como en porcentaje) y memoria debido a que el tiempo de ejecución

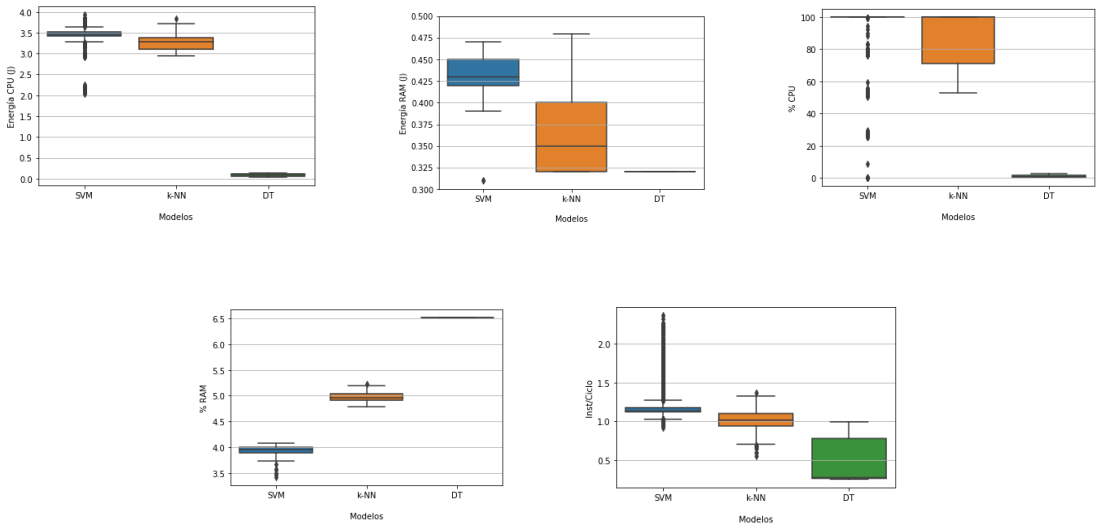

FIGURA 5.6: Comparación de energía consumida por el CPU, memoria RAM, %CPU, %RAM e instrucciones por ciclo para búsqueda evolutiva. Fuente: *Elaboración propia*

es pequeño, pero este patrón no se muestra en el porcentaje de memoria usada donde los valores se concentran alrededor del 6.6 %, aunque no es significativo. De nuevo el algoritmo DT es más lento al procesar menos instrucciones por ciclo que las demás que se mantienen por encima de 1.

La Figura 5.6 muestra el coste computacional para la búsqueda evolutiva. DT sigue manteniendo valores bajos para el consumo de energía del procesador y memoria aunque está última se use más. Por otro lado, tanto el SVM como el *k*-NN poseen resultados similares, teniendo éste último valores más dispersos en el consumo de energía de memoria y porcentaje de uso del procesador. Por último se nota que muchos valores del algoritmo SVM registran valores por arriba de una o dos instrucciones por ciclo lo que podríamos deducir que se ejecuta instrucciones más rápido.

Capítulo 6

Conclusiones y Trabajos a Futuro

En este capítulo detalla las conclusiones generales del presente trabajo, y las expectativas a futuro en este tema.

1. BÚSQUEDA EXHAUSTIVA: Se obtiene la mayor precisión y mejor configuración ya que evalúa todas las configuraciones de hiperparámetros posibles, pero tiene un alto coste computacional.

2. BÚSQUEDA ALEATORIA: Debido al empleo de probabilidades puede lograr una óptima precisión para pocos hiperparámetros y el coste computacional es menor que en el exhaustivo para los tres algoritmos trabajados.

3. BÚSQUEDA EVOLUTIVA: Obtiene alta presición muy cerca de la óptima, pero con un coste un poco mayor que la búsqueda aleatoria. Se puede considerar que tiene el mejor balance entre rendimiento y coste computacional.

4. Según los resultados obtenidos, el algoritmo *k*-NN junto con la búsqueda evolutiva generan la mejor optimización de hiperparámetros para los datos analizados.

6.1. Trabajo Futuro

En este primer seminario he desarrollado los conceptos principales en el entrenamiento y optimización de modelos de Machine Learning lo que me da

el primer paso para poder seguir avanzando. Como primera parte he estudiado las técnicas tradicionales de búsqueda y uno evolutivo simple, además de tres algoritmos supervisados; como trabajo futuro me abocaré en el estudio de optimización con métodos netamente del área de la computación evolutiva como algoritmos genéticos y algoritmos de estimación de distribuciones, trabajar con más algortimos de machine learning y trabajar con un clúster de gran poder de cómputo con los que pueda obtener resultados más confiables.

Bibliografía

[1] Stanford ML Group. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. https://stanfordmlgroup.github.io/projects/chexnet/. Accessed: 09/06/2019.

[2] SpaceML. Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit. http://space.ml/proj/GalaxyGAN. Accessed: 09/06/2019.

[3] Mark DePristo and Ryan Poplin. Deepvariant: Highly accurate genomes with deep neural networks. https://ai.googleblog.com/2017/12/deepvariant-highly-accurate-genomes.html. Accessed: 09/06/2019.

[4] Phillip Isola y Alexei A. Efros un Yan Zhu, Taesung Park. Unpaired image-to-image translationusing cycle-consistent adversarial networks., 2018. Accessed: 10/06/2019.

[5] Chun Yen Lee Yi y Ping Phoebe Chen. Machine learning on adverse drug reactions for pharmacovigilance, 2019. Accessed: 10/06/2019.

[6] Yongzhi Wang Lianwang Li Runting Li Kai Wang Shaowu Li Ke Tang Chuanbao Zhang Xing Fan Baoshi Chen Wenbin Li Zenghui Qian, Yiming Li. Differentiation of glioblastoma from solitary brain metastases using radiomic machine-learning classifiers, 2019. Accessed: 10/06/2019.

[7] Barrero Ortiz Giovanni. Fundamentals of machine learning for predictive data analytics. 2015. Accessed: 10/06/2019.

[8] Marco Federici Chiara Ghidini Fabrizio Maria Maggi Williams Rizzi y Luca Simonetto Chiara Di Francescomarino, Marlon Dumas. Genetic

algorithms for hyperparameter optimization in predictive. Business Process Monitoring, 2018. Accessed: 10/06/2019.

[9] A. Sava M. Siadat R. Laref, E. Losson. On the optimization of the support vector machine regressionhyperparameters setting for gas sensors array applications, 2019. Accessed: 10/06/2019.

[10] scikit-learn developers. Decision tree, mathematical formulation. http://scikit-learn.org/stable/modules/tree.html. Accessed: 15/06/2019.

[11] Decision tree - classification. http://www.saedsayad.com/decision_tree.htm. Accessed: 15/06/2019.

[12] scikit-learn developers. Support vector machines. http://scikit-learn.org/stable/modules/svm.html. Accessed: 15/06/2019.

[13] Support vector machine - classification (svm). http://www.saedsayad.com/support_vector_machine.htm. Accessed: 15/06/2019.

[14] scikit-learn developers. Nearest neighbors. http://scikit-learn.org/stable/modules/neighbors.html. Accessed: 15/06/2019.

[15] Shih-Wen Ke Li-Yu Hu, Min-Wei Huang and Chih-Fong Tsai. The distance function effect on k-nearest neighbor classification for medical datasets. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4978658/. 15/06/2019.

[16] K nearest neighbors - classification. http://www.saedsayad.com/k_nearest_neighbors.htm. Accessed: 15/06/2019.

[17] Gusseppe Bravo-Rocca Luis Orozco Barbosa e Ismael García Varea Manuel Castillo Cara, Jesús Lovón-Melgarejo. An empirical study of the transmission power setting for bluetooth-based indoor localization mechanisms. http://www.mdpi.com/1424-8220/17/6/1318. Accessed: 15/06/2019.