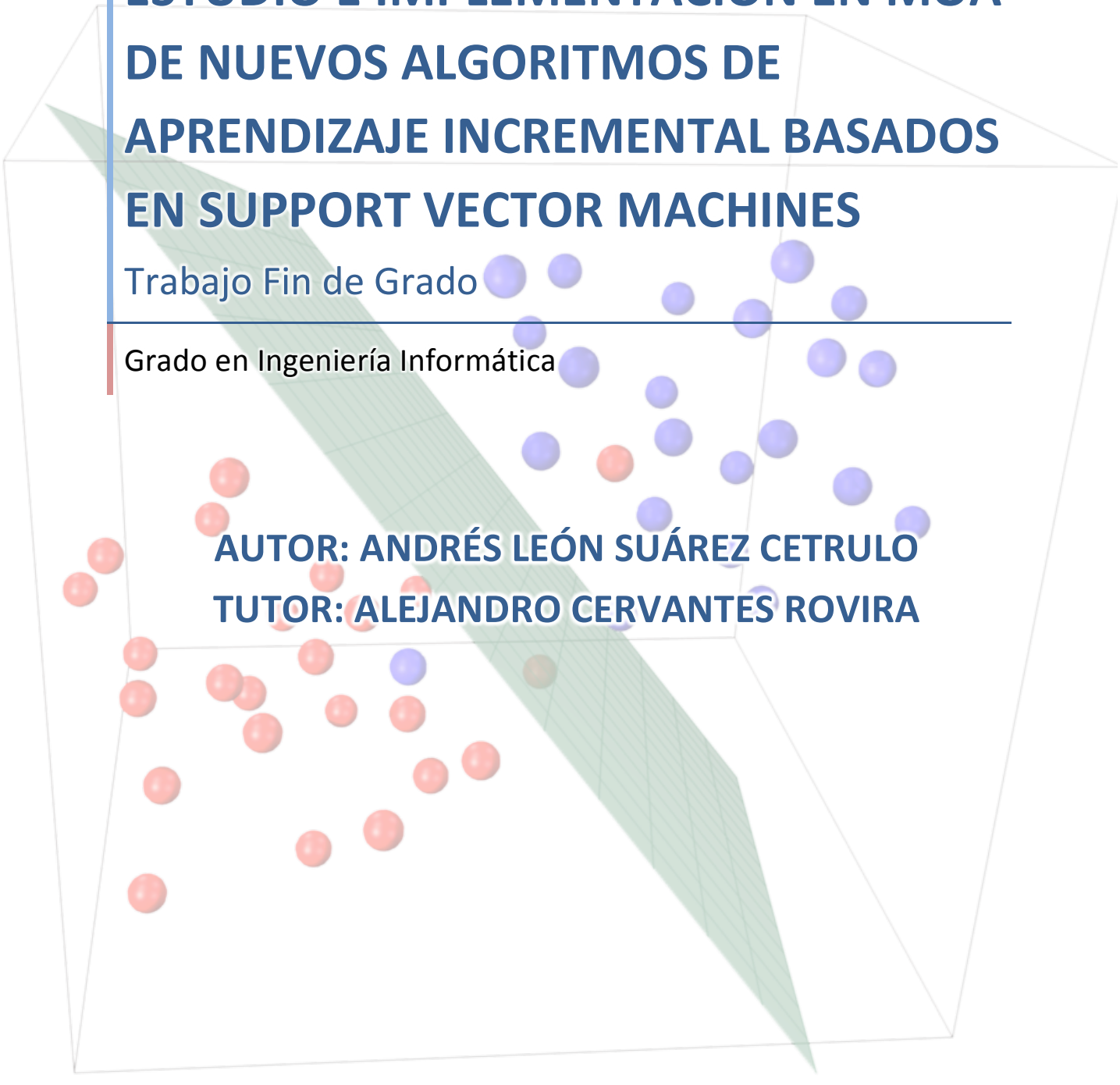
Universidad Carlos III de Madrid
www.uc3m.es

ESTUDIO E IMPLEMENTACIÓN EN MOA DE NUEVOS ALGORITMOS DE APRENDIZAJE INCREMENTAL BASADOS EN SUPPORT VECTOR MACHINES

Trabajo Fin de Grado

Grado en Ingeniería Informática



AUTOR: ANDRÉS LEÓN SUÁREZ CETRULO

TUTOR: ALEJANDRO CERVANTES ROVIRA

A Alejandro Cervantes, por mostrarme el camino a seguir;
a Alicia, por ayudarme a caminarlo;
a Alejandro Vegas, por escuchar cada metro recorrido.

Tabla de contenido

1. INTRODUCCIÓN 7
1.1. Contexto 7
1.2. Objetivos 8
1.3. Estructura 9
1.4. Nomenclatura utilizada 10

2. ESTADO DEL ARTE 11
2.1. Fundamentos del aprendizaje automático 11
2.1.1 Representación de los datos en el aprendizaje 12
2.1.2 Tareas más representativas 12
2.1.3 Tratamiento de las entradas (batch learning vs incremental learning) 13
2.1.4 Aprendizaje incremental en problemas de gran escala 14
2.1.5 Aprendizaje incremental en problemas con cambio de contexto 15
2.1.6 Aplicaciones del aprendizaje incremental sobre datos no estacionarios. 16
2.2. Técnicas batch estudiadas 18
2.2.1 Generalidades sobre Support Vector Machines (SVM). 18
2.2.2 Generalidades sobre LVQ 21
2.3. Estudio previo sobre la incrementalización de SVM 23
2.3.1 Técnicas de muestreo y clustering 23
2.3.2 Modelos de SVM incrementalizados existentes en la literatura. 24
2.4. Técnicas incrementales estudiadas: 25
2.4.1 Estudio de los clasificadores de Descenso de Gradiente Estocástico 25
2.4.2 Variante incremental de LVQ (ILVQ) 27
2.4.3 Análisis del modelo Online Incremental SVM 32
2.4.4 Técnica de agrupación Growing Neural Gas (GNG) 34
2.4.5 Híbrido Incremental GNG-SVM 40

3. HERRAMIENTAS PARA EL ESTUDIO EXPERIMENTAL DE ALGORITMOS DE APRENDIZAJE 45
3.1. OBJETIVOS 45
3.2. PLATAFORMAS DE EXPERIMENTACIÓN: WEKA Y MOA 46
i. Aspectos generales sobre uso y funcionamiento de MOA. 46
ii. Creación de un nuevo clasificador en MOA 48
iii. MOA a partir de WEKA 49
iv. Uso de librerías de WEKA 51
v. Meta-clasificador CVParamaterSelection 51

4. CODIFICACIÓN Y VALIDACIÓN DE ALGORITMOS 52
4.1. Procedimiento seguido 52
4.2. Selección de parámetros 53
4.3. Comparación con publicaciones previas 53

5. ALGORITMOS INCREMENTALES ESTUDIADOS 55
5.1. DESCRIPCIÓN Y JUSTIFICACIÓN DE LA VALIDACIÓN REALIZADA 55
i. Conjuntos utilizados en la validación 55
ii. Tratamiento de los conjuntos 56
iii. Modelo de evaluación utilizado 57
iv. Validación realizada 57
5.2. ONLINE INCREMENTAL SUPPORT VECTOR MACHINES 58
i. Análisis del algoritmo 58
ii. Diseño e implementación del clasificador 59
iii. Parámetros usados 61

iv. Pruebas de validación de OISVM *61*
v. Conclusiones de validación de OISVM: *66*
vi. Problemas encontrados *67*
5.3. GROWING NEURAL GAS 70
i. Análisis del algoritmo *70*
ii. Diseño e implementación del algoritmo de clustering *70*
iii. Parámetros usados *72*
iv. Prueba de validación de GNG y conclusiones *73*
iv. Problemas encontrados *75*
5.4. INCREMENTAL GROWING NEURAL GAS - SUPPORT VECTOR MACHINES 76
i. Análisis del algoritmo *76*
ii. Diseño e implementación del clasificador *77*
iii. Parámetros usados *78*
iv. Prueba de validación de IGNGSVM *79*
iv. Problemas encontrados *80*

6. ESTUDIO COMPARATIVO 81
6.1. Modos de evaluación utilizados 81
6.2. Conjuntos de datos usados 82
6.3. Descripción del análisis y representación de los resultados 84
i. Objetivo del estudio *84*
ii. Descripción del análisis *84*
iii. Representación de las pruebas *85*
6.4. Experimentación preliminar 85
i. Tratamiento de los conjuntos y selección de tareas de evaluación *85*
ii. Obtención de parámetros y elección de kernel en LibSVM y SGD *86*
iii. Proceso de reducción en OISVM y IGNGSVM *87*
6.5. Comparación de los resultados obtenidos 88
6.5.1. Pruebas de la literatura *88*
6.5.2. Pruebas usando una misma tarea de evaluación *91*
6.5.3. Conclusiones del estudio *97*

7. PLANIFICACIÓN 99
7.1. Planificación inicial semanal de las tareas (Diagrama de GANTT) 99
7.2. Ampliación del plazo del proyecto y re-planificación 100
7.3. Horas planificadas y realizadas 101
7.4. Seguimiento del proyecto 103
7.5. Detalles sobre el proceso 105

8. PRESUPUESTO 107

9. CONCLUSIONES Y LÍNEAS FUTURAS 109

10. OPINIÓN PERSONAL DEL TRABAJO Y AGRADECIMIENTOS 111
10.1 Realización y planificación del trabajo 111
10.2 Conocimientos adquiridos y mejorados 112
10.3 Agradecimientos 112

11. BIBLIOGRAFÍA Y REFERENCIAS 114

ANEXOS 117
A GUÍA DE INSTALACIÓN Y EJECUCIÓN DEL SOFTWARE 117
B INFORMACIÓN DE SALIDA 117
C SOFTWARE DESARROLLADO 117
D REGISTROS DE ACEPTACIÓN DE LOS ALGORITMOS PROGRAMADOS 118

1. INTRODUCCIÓN

En 50 años, Internet ha pasado de ser únicamente un experimento militar, a saturar el aire mediante redes Wi-Fi o 3G, y llenar los bolsillos y las casas de la población mundial. Concretamente, los últimos 20 años han supuesto el boom definitivo a nivel de comunicación, gracias sobretodo a la evolución vertiginosa de las tecnologías.

La red ha invadido definitivamente las vidas humanas, todo se mueve en Internet. Publicidad, marketing, ventas, son el principal negocio aquí. Las webs luchan por tener el primer puesto en buscadores y por elevar su PageRank; las empresas pagan grandes cantidades de dinero a las redes sociales por comprar información personal de sus usuarios... En los últimos tiempos, la minería de datos ha adquirido un papel importantísimo en los fines comerciales de Internet. Es aquí donde juegan un gran rol las herramientas de análisis de datos basadas en algoritmos de aprendizaje automático.

Este trabajo se basa en la integración de nuevas técnicas de aprendizaje automático a los entornos de análisis WEKA y MOA, así como en su posterior estudio experimental. Las pruebas realizadas en esta última fase, consisten en la comparación de los resultados obtenidos entre los algoritmos, en cuanto precisión de las predicciones y coste computacional del proceso.

A continuación, se abordarán los objetivos del proyecto, los motivos que han ayudado a su realización y, por último, se describirá la estructura del mismo.

1.1. Contexto

Los algoritmos de decisión clásicos realizan sucesivas iteraciones sobre un conjunto de entrenamiento con vistas a la construcción del modelo final. Se basan en procesos de aprendizaje en los que el dominio puede ser descrito desde el inicio del problema, y que pueden guardar en memoria sus ejemplos de entrenamiento durante el tiempo limitado que este dure.

Hasta ahora, el éxito de estas técnicas se ha basado en la poca cantidad de conjuntos de datos existentes para su análisis. Sin embargo, más recientemente han surgido áreas de gran interés donde los datos surgen de entornos dinámicos y son adquiridos a lo largo del tiempo. Además, la automatización de procesos como la minería de datos, su análisis y almacenado, han hecho necesarias técnicas capaces de analizar grandes bases de datos sin el costo computacional que supone un volcado masivo a memoria. Todo ello, obliga a los sistemas de análisis de datos, a procesar los individuos de entrenamiento secuencialmente, y por tanto, a ir adaptando el modelo según varía su distribución de probabilidad. Esto es conocido como Aprendizaje Incremental.

A diario crece cada vez más el tamaño de los conjuntos de datos utilizados con técnicas de datamining. Esto hace aumentar la velocidad con la que los algoritmos de

aprendizaje reciben los nuevos ejemplos de entrenamiento, a modo de flujos continuos de datos. Un ejemplo de estos pueden ser eventos y registros en redes de telecomunicación, transacciones de datos por satélite, operaciones financieras de bolsa, o los datos de aprendizaje para el filtrado de un spam que en 2010 supuso el 89,1% de los mensajes enviados por correo electrónico alrededor del mundo (unos 262.000 emails de spam diarios) (Pingdom 2011). En estas condiciones, resulta imposible reunir y filtrar en una sola iteración, el número de ejemplos requeridos en un proceso de aprendizaje. Esto exige, tanto operar en línea, de forma continua, como procesar cada ejemplo a tiempo real.

Por su parte, cada día crece más la necesidad de utilizar de técnicas que compriman la cantidad de datos existentes para agilizar su procesamiento. Un ejemplo lo constituyen los problemas de segmentación de mercado para operaciones de marketing. Aplicaciones de este tipo están teniendo cada vez más éxito comercial, especialmente en Internet debido a la minería de datos realizada en las redes sociales como Twitter o Facebook, meta información de sitios web, etc.

En este contexto, las técnicas de aprendizaje automático online, de forma incremental, suponen un gran beneficio. Estas permiten analizar flujos de datos continuos o de gran tamaño, permiten dar una respuesta en cualquier momento, no necesitan de reentrenamiento previo y se adaptan a los llamados cambios de contexto, o concept drift (cambio de tendencia en la aparición de ejemplos y clasificación de resultados, a menudo producido por un factor externo pero relacionado al objeto de estudio) , dentro del flujo de datos. Todo ello, hace que este proyecto sea de gran utilidad, no sólo a nivel de investigación, si no también de gran interés para fines comerciales como los ejemplos citados anteriormente.

1.2. Objetivos

El objetivo de este proyecto es el estudio comparativo de tres nuevos algoritmos de clasificación sobre conjuntos de datos de las características mencionadas anteriormente, de forma que se pueda precisar de qué manera se pueden utilizar y qué utilidad tienen con respecto a las técnicas convencionales. Todo, con la esperanza de obtener resultados que ayuden en las investigaciones actuales.

Una primera fase del trabajo se centra en la integración de los algoritmos en una plataforma única que permita realizar la experimentación de forma lo más automatizada posible. Para ello hemos seleccionado la plataforma MOA (Massive Online Analysis), en la que se han tenido que codificar varios algoritmos no incluidos que se describirán más adelante. El estudio se centra en la validación, y comprobación de la eficiencia de estos algoritmos sobre conjuntos de datos que puedan aplicarse a fines comerciales a fecha de hoy. Partiendo de estudios previos, así como de la literatura pertinente de cada algoritmo, se procederá a la integración a las herramientas de desarrollo WEKA y MOA, identificando las posibles dificultades y errores que pudieran surgir y validarlos mediante pruebas realizadas en artículos anteriores de la literatura pertinente. Por último se realizará un estudio comparativo

entre los tres algoritmos, buscando el algoritmo que mejor se adapte a los conjuntos de pruebas planteados, y que mejores soluciones ofrezca.

El problema de la clasificación incremental supone muchos retos a nivel de estudio, debido a la profundidad teórica utilizada y cantidad de literatura existente en torno a éste. Sin embargo, una vez se vean sus posibilidades, podrá comprobarse como no deja de ser un tema interesante a tratar, no sólo a nivel de investigación, si no de utilidad en las aplicaciones de software como posible mejora y actualización a las mismas.

1.3. Estructura

El presente trabajo se presenta dividido en diez secciones, de las cuales, esta presenta una breve introducción general sobre el proyecto a realizar, la motivación de su desarrollo, y sus objetivos.

El apartado 2 se centra en el estado del arte, la documentación y bases teóricas de todo el proyecto. Empieza explicando a grandes rasgos qué es el aprendizaje automático, y va profundizando hasta llegar a cada uno de los tres algoritmos propuestos.

En la sección 3 se muestra el entorno de análisis utilizado en el estudio de los algoritmos, se habla sobre su funcionamiento y sobre la integración de los clasificadores.

En el apartado 4 se habla del proceso seguido, de forma general, en la codificación y validación de los algoritmos. Por otro lado, en la sección 5, particularizamos el proceso anterior para cada uno de los tres algoritmos separadamente.

En la parte 6 se realiza el estudio comparativo de los algoritmos sobre una serie de conjuntos de datos de prueba. Se seleccionan determinados modos de evaluación y se intenta extraer conclusiones relacionadas al funcionamiento y utilización de los algoritmos en cada escenario. Se habla de los modos de evaluación utilizados, de los conjuntos de datos usados; se muestra un análisis detallado, así como representado gráficamente, de los resultados; y se comparan los resultados obtenidos con las pruebas en los distintos clasificadores.

En la sección 7, se aborda el proceso de planificación seguido, tanto en horas como en días, y se compara al trabajo realizado final. Acto seguido, en el apartado 8, se muestra un presupuesto del trabajo, deducido de la planificación mostrada.

En la secciones 9 y 10, se recogen las conclusiones extraídas durante todo el trabajo, así como se habla sobre posibles mejoras y otras pruebas a realizar en un futuro.
Como último apartado, la sección 11 recoge la bibliografía utilizada a lo largo de este trabajo.

Además de todo lo anterior, se ofrece un anexo con información adicional que puede resultar de utilidad como añadido del trabajo. Ficheros de salida en la fase de experimentación, una guía de usuario, o el software desarrollado, son algunos de los contenidos que se pueden encontrar en este.

1.4. Nomenclatura utilizada

De cara a facilitar la lectura de este documento, se proporcionan acto seguido indicaciones sobre la nomenclatura seguida.

- Las palabras en cursiva hacen referencia a algoritmos o conceptos explicados en el documento. En el propio apartado de la explicación de dicho concepto no se pone dicha palabra en cursiva (p.e. *OISVM* no aparece en cursiva en su apartado del estado del arte).

- Los pies de imagen, tablas, gráficas y fórmulas, son numerados con la numeración de la sección en la que se encuentran, y con el número de imagen, tabla, etc. que tengan en dicho apartado (p.e. la segunda gráfica del apartado 4.5 tendría la numeración 4.5.2).

- La numeración de los pies, tablas, etc. puede verse repetida si los objetos son distintos (p.e. puede haber una tabla 4.5.2 y una gráfica 4.5.2).

- Las referencias bibliográficas se encuentran en el apartado 11 numeradas entre corchetes. Sin embargo, para hacer referencia a cada una se cita el nombre del autor y el año separado por comas, entre paréntesis.

2. ESTADO DEL ARTE

En este punto, se tratarán los aspectos teóricos necesarios para comprender el trabajo realizado en el TFG. Se analizará cada uno de los conceptos base, y se estudiarán las elecciones a tomar de cara a enriquecer las conclusiones inferidas en el proyecto.

Por un lado, se explicarán los conceptos de aprendizaje automático necesarios para entender el trabajo realizado, así como las nociones y funcionamiento de los clasificadores escogidos para el estudio y evaluación.

Por otro lado, puesto que la finalidad de este trabajo es la evaluación de nuevas técnicas de aprendizaje incremental mediante el uso de MOA, se estudian las bases de los algoritmos de aprendizaje incremental a integrar. Para ello, se explica la importancia de los conjuntos de datos a probar, y se abarcarán distintos tipos de estos. Para describir cada algoritmo se ha utilizado la notación original de los artículos de la literatura de los que se han extraído.

2.1. Fundamentos del aprendizaje automático

Entendemos por aprendizaje automático la rama de la Inteligencia Artificial cuyo propósito se dirige al desarrollo de técnicas de adquisición de conocimiento de forma automática, con el fin de resolver nuevos problemas o mejorar sus decisiones a partir de la experiencia acumulada. Dicho de otro modo, es un tipo de programación que trata de generalizar comportamientos extraídos de una información no estructurada proveída a modo de ejemplos.

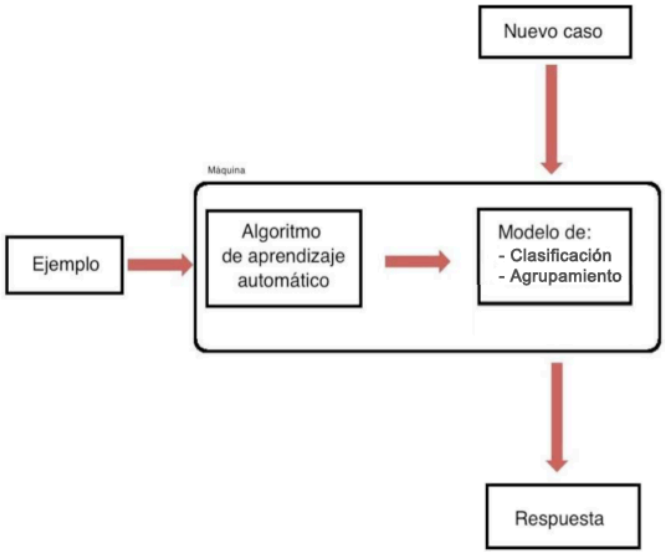


Figura 2.1: *Se representa en varias cajas la dinámica de funcionamiento de las máquinas de aprendizaje. Este trabajo se centra en los modelos de clasificación.*

2.1.1 Representación de los datos en el aprendizaje

En primer lugar, hay que resaltar la importancia de la estructura de MOA basada en instancias para el desarrollo de este proyecto. Entendemos por instancias los datos utilizados como fuente de análisis, en el aprendizaje realizado por los clasificadores abarcados más adelante.

Cada instancia estará formada por un vector de valores. Estos valores contienen los atributos, que en nuestros ejemplos de validación y evaluación son de valor numérico. El vector además contiene un valor denominado clase o etiqueta, que hace referencia a los distintos tipos en que pueden clasificarse los ejemplos entrenados.

2.1.2 Tareas más representativas

A continuación se citarán las dos tareas más representativas de los problemas de aprendizaje: clasificación y agrupación (clustering).

i. Clasificación

Dicha tarea en aprendizaje automático puede definirse como la búsqueda de una representación de correspondencia entre las observaciones y las clases a través de un conjunto de reglas de inferencia. Para ello deben conocerse previamente las clases, a diferencia de la tarea de agrupación, donde son resultado de un proceso de construcción.

Se denomina **aprendizaje supervisado** a aquel en el que existe una clasificación previa de instancias. Dicho de otro modo, cuando se dispone de las etiquetas del conjunto de entrenamiento.

Un tipo de aprendizaje supervisado se denomina aprendizaje basado en instancias En el caso más simple (vecino más cercano), se guardan todos los ejemplos de entrenamiento. A la hora de clasificar un nuevo objeto, se extraen los objetos más parecidos y se usa su clasificación para clasificar este. A partir de este esquema básico aparecen versiones que no guardan todas las instancias sino que generan un conjunto de prototipos (*locally weigthed regression*, *funciones de pesos o kernels*, etc.).

ii. Agrupación

Esta tarea de aprendizaje parte de un conjunto de ejemplos cuya clase o etiqueta se desconoce. Su objetivo es hallar las clases en las que agrupar dichos ejemplos de tal forma que los objetos de un grupo o clúster tengan una similaridad alta entre ellos, y baja con objetos de otros clúster.

Se denomina **aprendizaje no supervisado** a aquel en el que no existe una clasificación o etiquetado previo de instancias.

2.1.3 Tratamiento de las entradas (batch learning vs incremental learning)

Las instancias que entran a un clasificador pueden ser tratadas de forma incremental (incremental o online) o simultánea (batch). En la forma simultánea, todas las experiencias son tratadas a la vez. En el modo incremental, se consideran las instancias en grupos o de forma aislada, a medida que van entrando al sistema.

i. Aprendizaje simultáneo (batch)

En el aprendizaje batch se aprovecha la existencia de un conjunto completo de instancias para tratarlas a la vez, con el que se "entrena" el clasificador antes de poder utilizarse. Esta consideración simultánea permite introducir un análisis estadístico con el que aislar errores posiblemente cometidos en la obtención de las instancias.

ii. Aprendizaje incremental

Se dice que se está realizando un tratamiento incremental de las instancias cuando son tratadas de forma aislada y secuencial (de una en una). El aprendizaje incremental acostumbra a descartar o reajustar continuamente, y según aparezcan nuevas instancias, las hipótesis generadas a lo largo del proceso.

Las técnicas de aprendizaje simultáneo (pertenecientes al batch learning) no pueden aplicarse sobre problemas de entorno dinámicos donde los datos se obtienen a lo largo del tiempo, esto obliga a procesar los datos secuencialmente (bien individual o bien en tandas), de forma que el modelo se va construyendo de forma incremental, es decir, ampliándolo o corrigiéndolo a medida que se extrae nuevo conocimiento.

Pueden señalarse los factores determinantes en el diseño de un algoritmo incremental como tres:

- Influencia del orden de llegada. Pese a que los ejemplos no tienen porqué depender del tiempo, puede ser necesario contar con el instante de llegada según el problema.
- Adaptación al cambio. Los sistemas incrementales asumen la hipótesis de mundo abierto, por la que puede contemplarse el posible cambio de la forma de clasificar a lo largo del aprendizaje. Este supuesto permite aplicaciones reales, donde los datos generalmente presentan ruido, no completitud, valores nulos, etc.

- Curva de aprendizaje. Los sistemas incrementales comienzan con un modelo vacío que se va actualizando, ampliando y simplificando a lo largo del aprendizaje. Esto hace que el nivel de confianza en predicciones de épocas tempranas puede que no sea tan bueno como en etapas posteriores .

2.1.4 Aprendizaje incremental en problemas de gran escala

Como primer aspecto del estudio de los modelos incrementales, hay que recalcar que no todo sistema online se basa en un modelo de aprendizaje incremental. Los sistemas online son aquellos capaces de dar respuesta en todo momento, cosa que podría llegar a abordarse mediante un modelo batch (tema aparte es la recepción continua de datos). Esto se debe a la posibilidad de almacenamiento de datos previos en memoria, con lo que un sistema batch podría dar respuestas sin haber evaluado aún el lote de instancias.

Una alternativa es añadir las nuevas instancias a los ejemplos de entrenamiento y aplicar el algoritmo de aprendizaje. Dichos sistemas, conocidos como sistemas batch temporales tienen como ventaja obviar el orden de entrada de las instancias. Sin embargo, crean bastantes problemas en memoria, ya que en dominios de aprendizaje online los datos entran continuamente a lo largo del tiempo. Por otro lado, la forma de clasificar variará en función del contexto. Los algoritmos batch temporales intentarán mantener consistente un modelo preparado para un contexto distinto (no percibido por este) que volverá el aprendizaje obsoleto.

Un algoritmo de aprendizaje incremental puro sólo tiene en cuenta las instancias recibidas en cada instante de tiempo, es decir, que no almacena datos en memoria. Como caso opuesto, un algoritmo incremental puro también puede ser utilizado en un dominio no incremental. El primer posible inconveniente de esta aplicación es que no se aprovecha toda la información del dominio proveído a priori, ya que sólo se consideran las instancias recibidas en cada momento junto al modelo previamente aprendido. Otro inconveniente puede ser la influencia del orden de llegada de las instancias, en caso de que dicho factor no debiera ser determinante en el dominio. Como ventaja, su coste computacional siempre será menor (por ser incremental) que la de un sistema batch.

En las siguientes secciones, se abarcan algunas técnicas de aprendizaje batch aplicadas a algoritmos incrementales que son capaces de detectar cambios en la función objetivo, aprendiendo por tandas temporales (mediante buffer de datos). En ellos, partimos del principio de que podría ser interesante mezclar instancias pasadas y actuales, siempre que estas se solapen conceptualmente y ayuden a converger de forma más óptima hacia la función objetivo del nuevo contexto.

Problemas con grandes y pequeños conjuntos de datos

Se llaman problemas de gran escala (Large-scale Data Problems) aquellos cuya dimensión consume gran cantidad de recursos computacionales. Los problemas de gran escala, se ven en muchos casos beneficiados por los métodos de aprendizaje incremental, debido a su mejoría en cuanto a consumo de recursos. A continuación se exponen, a modo comparativo, las principales limitaciones que encuentran las técnicas de aprendizaje automático en problemas de gran escala, frente a las encontradas al tratar con pequeños sets de datos según (Bottou, Large-Scale Machine Learning with Stochastic Gradient Descent 2010).

Por un lado, los problemas de aprendizaje con pequeños conjuntos de datos, se encuentran limitados por el número de ejemplos existentes en el set de entrenamiento. El tiempo de cálculo no supone un problema. Por otro, se puede reducir el decrecimiento de la precisión de acierto, ocasionado por la partición del conjunto de ejemplos (Bottou, Large-Scale Machine Learning with Stochastic Gradient Descent 2010), a niveles insignificantes si la reducción aplicada sobre cada bloque es mínima. Por tanto, se puede recuperar el balance entre aproximación y estimación que ha sido ampliamente estudiado en la estadística y en la teoría del aprendizaje.

Por el contrario, los problemas de gran escala están limitados por el coste temporal computacional que consumen. Las tareas de compresión (agrupamiento) de los ejemplos harán reducir este impacto, debido a que puede ser procesada una representación mayor de ejemplos de entrenamiento durante el tiempo consumido.

2.1.5 Aprendizaje incremental en problemas con cambio de contexto

Uno de los problemas que surgen en la realidad y que los algoritmos tipo batch no pueden considerar, es una posible dependencia del contexto por parte de la función objetivo que no es recogida por la entrada. Por ejemplo, las reglas del problema de predicción meteorológica varían drásticamente dependiendo de la estación del año. Los problemas en los cuales las causas de los cambios aparecen como ocultas son conocidos como *hidden context*, y sólo pueden tratarse mediante algoritmos incrementales, ya que pueden ampliar y corregir las hipótesis generadas.

En el análisis estadístico, se define una serie estacionaria como aquella cuya media y varianza respectivas no cambian a través del tiempo. Por tanto, podemos llamar aprendizaje sobre datos no estacionarios a aquel en que los datos pueden considerarse muestras de distribuciones aleatorias cuyos parámetros varían con el tiempo.

i. El problema de concept drift

El contexto oculto puede variar de forma que invalida el conocimiento ya obtenido (A.Tsymbal 2004) (G.Widmer 1996). En estos casos, decimos que estamos en un problema con "concept drift". El problema aquí es que al tratar con datos con probabilidad de errores o ruido, dichos datos con ruido pueden ser confundidos con un cambio en el contexto (concept drift). Los sistemas más robustos con respecto al ruido se adaptarían a los cambios con bastante retardo, pudiendo llegar a ignorar algunos. Por otro lado, un sistema muy sensible tomaría datos con ruido como cambios. Por ello, es decisivo el *balance* entre robustez al ruido y sensibilidad al cambio en el rendimiento de un sistema de aprendizaje con datos no estacionarios.

ii. Dominios de contextos recurrentes

Otro de los problemas que puede encontrarse un sistema de aprendizaje incremental es la posibilidad de un contexto que cambia de manera recurrente debido a fenómenos cíclicos (p.e. estaciones del año).

Debido al reducido número de técnicas diseñadas para la adaptación a contextos recurrentes, muchos sistemas de aprendizaje basan su estrategia en almacenar tendencias ya ocurridas en memoria, de forma que puedan ser consultadas si el ciclo vuelve a repetirse, acelerando la adaptación al cambio de contexto. Por ello, al balance decisivo entre robustez y sensibilidad, habrá que añadírsele la capacidad para identificar contextos recurrentes.

iii. Tipos de cambio

En función de la frecuencia de recepción de los datos, se distinguen dos tipos de *concept drift* o tipos de cambio: gradual o abrupto (cambios repentinos). Por otro lado, los problemas *hidden context* no sólo pueden variar en la función objetivo, sino también en la distribución de los atributos en las instancias. Puede diferenciarse entre *sampling shift* (los cambios afectan sólo a los datos de entrada) y *concept shift (cambio de la forma de clasificar)*, no obstante el tipo de cambio es irrelevante en la práctica, ya que ambos llevan a la necesidad de revisar y actualizar el modelo.

2.1.6 Aplicaciones del aprendizaje incremental sobre datos no estacionarios.

Los sistemas de aprendizaje incremental, al contrario que los batch, evolucionan y actualizan su modelo a medida que son procesadas las nuevas instancias. Por ello es el primero es el más adecuado a la hora de tratar el problema del *concept drift*.

A continuación se citarán varios ejemplos de áreas de aplicación para procesos incrementales no estacionarios:

- Agentes Inteligentes. El aspecto incremental es inherente al aprendizaje de todo agente (p.e. gestión del trafico y optimización del rendimiento en redes).
- Robótica de exploración. Un robot de exploración debe poder adaptarse incrementalmente a los cambios de un entorno a menudo imprevisible.
- Perfiles de usuario y detección de intrusos. Muchos estudios avalan la variación radical de los intereses y comportamientos de un usuario cada medio año.
- Aprendizaje en stream de datos: donde la distribución de los datos y el modo de clasificación son sensibles al contexto y varían de forma impredecible.

Otros ejemplos de áreas de aplicación para sistemas incrementales son: modelos de comportamiento para movimientos de bolsa, el reconocimiento de patrones de compra en grandes superficies, sistemas de reconocimiento de spam, análisis de imágenes para climatología y planificación de procesos.

Por otro lado, al trabajar con flujos de datos continuos a tiempo real, se hace imposible recopilar a tiempo todos los ejemplos necesarios para el proceso de aprendizaje. Además, las restricciones en el tiempo de respuesta y las limitaciones en memoria, hacen que los algoritmos incrementales sean óptimos para procesar estos datos.

2.2. Técnicas batch estudiadas

2.2.1 Generalidades sobre *Support Vector Machines* (SVM).

Las máquinas de soporte vectorial proyectan el espacio de entrada (el de atributos) en otro espacio (kernel), de forma que las clases sean separables por un hiperplano en dicho espacio. Su objetivo es encontrar el hiperplano que separa las clases y está a máxima distancia de ambas en éste nuevo espacio.

Para explicar en un principio el funcionamiento de las máquinas de soporte vectorial, partiremos de los modelos lineales de SV, donde podamos observar gráficamente los resultados del proceso de generalización. Una vez abarcado el concepto lineal, se resumirá su uso en máquinas de soporte vectorial no lineales.

i. Funcionamiento teórico de las máquinas de soporte vectorial.

En primer lugar, para la explicación del funcionamiento de un modelo lineal, determinamos un problema binario (dos posibles clases): A y B. Formalmente: Y = { A , B }

Para continuar con la explicación, antes hacemos hincapié en el concepto de conjunto de vectores separable.

Un conjunto de vectores $\{(x_1, y_1), \cdots, (x_n, y_n)\}$ *donde* $x_i \in R^d$ *e* $y_i \in \{A, B\}$ *para* $i = 1, \cdots, n$ *será separable si existe algún hiperplano en* R^d *capaz de separar los vectores* $X = \{x_1, \cdots, x_n\}$ *con clase* $y_i = A$ *de aquellos con clase* $y_i = B$.

Dado un conjunto separable, existirá como mínimo un hiperplano π: $w \cdot x + b = 0$ que separe los vectores $X = \{x_1, \cdots, x_n\}$. Las máquinas de soporte vectorial buscan el hiperplano separador que maximice la distancia de separación entre los conjuntos de las clases existentes. La siguiente figura representa un conjunto separable.

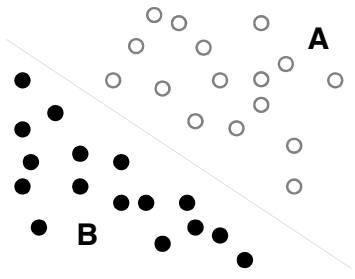


Figura 2.2.1.1: *Conjuntos de vectores en* IR^2 $\{(x_i, A)\}$ *y* $\{(x_i, B)\}$ $\setminus$ $i = 1, \cdots, n$. *para un conjunto de datos separable. Imagen extraída de* (Abril 2003).

Una vez fijado un hiperplano separador, será siempre posible re-escalar los parámetros w y b de la forma **(Ecuación π1)** y **(Ecuación π2)** (dicha nomenclatura se

utiliza por nombrar ecuaciones de planos). Debido a ello, la separación mínima entre los vectores y el hiperplano separador es uno, pudiendo simplificar ambas inecuaciones en **(Ecuación π3)**.

$$x_i \cdot w + b \geq +1 \text{ para } y_i = A \qquad \textbf{(Ecuación π1)}$$
$$x_i \cdot w + b \leq -1 \text{ para } y_i = B \qquad \textbf{(Ecuación π2)}$$
$$y_i(x_i \cdot w + b) - 1 \geq 0,\ i = 1, \cdots, n \qquad \textbf{(Ecuación π3)}$$

+1 y -1 representan a las etiquetas A y B respectivamente, para su uso en las desigualdades. w representa el vector normal al correspondiente hiperplano. Formulas extraídas de (Abril 2003).

Despejando se obtiene que la distancia perpendicular hasta el origen es el valor absoluto de la diferencia entre valor de clase (+1 o -1) y b, dividido entre la norma euclídea del vector normal w (‖w‖) .

De esta forma se obtiene que ambos hiperplanos **(Ecuaciones π1 y π2)** son paralelos, siendo la separación entre ambos dos entre la norma euclídea del vector normal, y que ningún vector del conjunto de entrenamiento se encuentra entre ellos (son Support Vectors). Los planos se escogen de tal forma que la separación entre ambos sea la mayor posible. Con ello se mejora la distinción entre regiones con puntos de distinta clase.

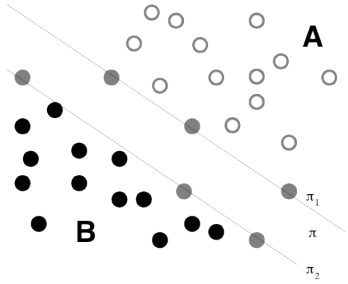


Figura 2.2.1.2:*Hiperplanos paralelos y vectores soporte en IR^2. Imagen extraída de* (Abril 2003).

Gracias a encontrarnos en un problema bidimensional podemos interpretar la solución al ver la figura. Se verifica como al añadir o eliminar cualquier número de vectores que verifiquen la inecuación, no se afectará a la solución. Sin embargo, si se añade un vector entre los dos hiperplanos separadores SV, la solución cambiará radicalmente.

La separación más sencilla es la formada por una línea, plano, o un hiperplano N-dimensional. Sin embargo, casi la totalidad de los conjuntos existentes no son representables en dos dimensiones. Las máquinas de soporte vectorial deben tratar con:

- Más de dos atributos.
- Conjuntos no totalmente separables.
- Curvas de separación (separación no lineal).
- Clasificación multiclase.

ii. Máquinas de soporte vectorial no lineales

Por su gran número de limitaciones del ámbito computacional, las máquinas de aprendizaje lineal no pueden emplearse en la gran mayoría de utilidades reales. Es aquí donde el uso de funciones kernel proporciona una solución. Las funciones kernel proyectan el espacio de entradas original (x) en otro de mayor dimensión, de manera que, en ese espacio, el problema se hace linealmente separable.

Se transforman los vectores, de forma que la notación utilizada será de la forma $\{\Phi(x_1),\cdots,\Phi(x_n)\}$, de tal forma que si se plantea el problema sin transformar sobre los vectores transformados, se obtiene que estos sólo forman parte de la solución a través del producto escalar $\langle\Phi(x_i),\Phi(x)\rangle$. Para resolver, únicamente se requerirá el conocimiento de la función kernel utilizada.

Por tanto, al reemplazar el producto escalar del espacio de entradas por el del núcleo correspondiente, obtenemos que una SVM planteada en un nuevo espacio, y la ejecución de la técnica no lineal, consumen los mismos recursos computacionales que con una técnica lineal.

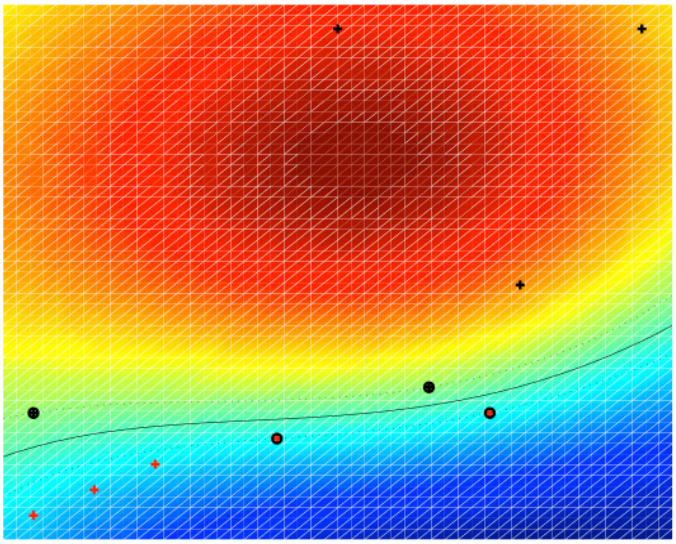


Figura 2.2.1.3:: *Solución al problema de la sección i a partir de un kernel gaussiano. Los puntos en círculos representan los vectores soporte (SV). Imagen extraída de* (Abril 2003).

iii. Máquinas de soporte vectorial multiclase

Las máquinas de vectores soporte (SVM) fueron diseñadas originalmente para la clasificación binaria. Extender su uso para la clasificación en más de dos categorías (multiclase) de forma que sea efectiva, sigue siendo un tema de investigación vigente. Sin embargo, varios métodos han sido ya propuestos a modo de clasificador multiclase mediante la combinación de varios clasificadores binarios.

Por otro lado, algunos autores también proponen métodos que tratan todas las clases al mismo tiempo. Como tiene mayor coste computacional resolver problemas multiclase, no se han llevado a cabo comparaciones exhaustivas de estos métodos para problemas de gran tamaño. En caso de conjuntos de datos de gran tamaño, abordar problemas grandes y multiclase se hace aún más complejo, ya que el coste de este tipo de máquinas de es mayor que el de las análogas de dos clases.

2.2.2 Generalidades sobre LVQ

Learning Vector Quantization (LVQ), es un modelo basado clasificación de instancias de entrada según los prototipos entrenados que haya más cercanos. En el proceso de aprendizaje, se crearán *grupos* o *clústeres* que luego serán etiquetados.

Dado el conjunto de entrenamiento C, el objetivo es la construcción de un conjunto de referencia, C_{LVQ}, cuyo número de elementos sea menor. Las instancias a clasificar se etiquetarán usando el concepto de *regiones de Coronio* (en la práctica puede usarse la regla 1-NN) tomando como referencia el conjunto C_{LVQ}.

i. Conceptos teóricos básicos: diagrama de Voronoi

Puede definirse un diagrama de Voronoi como una descomposición de un espacio métrico en regiones, asociada a la presencia de objetos, de tal forma, que en dicha descomposición, a cada objeto se le asigne una región del espacio métrico formada por los puntos que son más cercanos a él que a ninguno de los otros objetos. p.e. si tenemos la figura de la izquierda, ante la presencia de un nuevo punto, es inmediato reconocer cuál de los 8 originales es el más cercano a él (el generador de la región en la que ha sido colocado el nuevo).

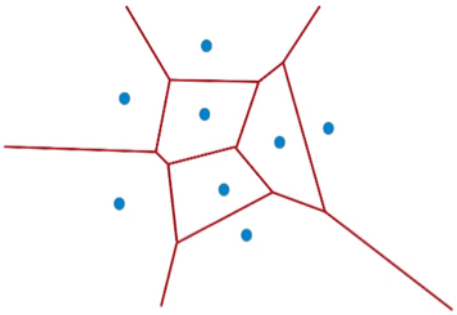


Figura 2.2.2.1:: *Diagrama de Voronoi que representa un conjunto de 8 puntos en dos dimensiones.*

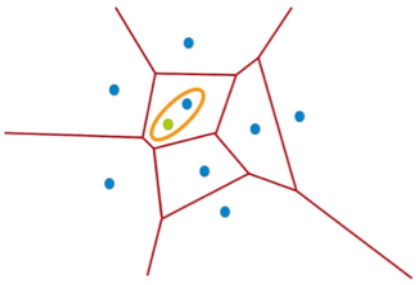


Figura 2.2.2.2:: *Diagrama de Voronoi que representa un conjunto de 8 puntos en dos dimensiones sobre el que se añade un nuevo punto (verde).*

LVQ usa la información de clases para mover los prototipos existentes ligeramente. Para clasificar una instancia de entrenamiento, este debe ser comparada con todos los prototipos existentes, donde cada uno representa a una región. Si la clase

etiqueta el vector de entrada de acuerdo un prototipo, dicho prototipo se mueve en la dirección a la entrada. Si la etiqueta en desacuerdo, se aleja de la dirección de la entrada. Para seleccionar el prototipo mas cercano a la instancia de entrenamiento, se utiliza la distancia euclídea (en el siguiente apartado, se cita el algoritmo k-NN para realizar esto).

ii. Descripción del clasificador

En resumen, la red LVQ se considera como un clasificador supervisado que divide el espacio de entrada en regiones disjuntas. Para explicar de forma resumida el proceso de *reducción que tiene lugar en* LVQ, puede plantearse un algoritmo, dividido en dos fases, que produce un conjunto de prototipos C_{LVQ}.

1. Inicialización del conjunto de entrenamiento original C (se elige un subconjunto de N_p instancias de C ($|C_{LVQ}| = N_p$)). Se puede asegurar que una instancia de entrenamiento está dentro de un agrupamiento o región si su clasificación mediante la regla k-NN, es correcta. En ese caso, las instancias son añadidas al subconjunto de referencia C_{LVQ}.

   En el algoritmo k-NN (k-nearest neighbours), los prototipos vecinos más cercanos a una instancia se obtienen utilizando la distancia euclídea sobre los n posibles atributos.

2. Aprendizaje. Consiste en la actualización iterativa de los prototipos del conjunto de referencia actual en tantos pasos como número total de prototipos (o regiones) haya. La salida será último conjunto de referencia corregido. En cada iteración se repite lo siguiente:

   a. Se selecciona una instancia del conjunto de entrenamiento.
   b. Se actualiza el conjunto de prototipos C_{LVQ} a partir de la nueva entrada. Dichas actualizaciones, al utilizar prototipos etiquetados, son de premio o castigo

2.3. Estudio previo sobre la incrementalización de SVM

Una de las deficiencias de las máquinas de soporte vectorial es el elevado coste computacional espacial que tiene el proceso de entrenamiento. Esto se convierte en un problema significativo en la clasificación de **grandes conjuntos de datos**. La complejidad del proceso de entrenamiento depende directamente del tamaño del conjunto de datos que se usa para dicho proceso. Por ello existe, la necesidad de reducir el tamaño de los conjuntos. En la mayoría de casos, dicho problema se resuelve mediante diversas técnicas de muestreo de datos o técnicas de agrupamiento.

Por otro lado, en este trabajo también tratamos con **conjuntos de datos no estacionarios**. Una forma de procesar estos es su partición en varios bloques de datos, utilizando parte de la información de los datos anteriores. Esto hace que el modelo se actualice durante el aprendizaje y tenga posibilidad de percibir el cambio de contexto; al contrario de una SVM batch, que clasificaría todos los ejemplos en el contexto medio del conjunto al utilizar toda la información en una iteración. El primer algoritmo mencionado en **2.3.1** *OISVM*, sigue esta filosofía de bloques. Además, en el **punto 2.3.2** se citan varios ejemplos de SVM incrementales.

2.3.1 Técnicas de muestreo y clustering

- Las técnicas de muestreo seleccionan un subconjunto representativo de instancias del conjunto de datos original. El muestreo selectivo, el aprendizaje activo, o el muestreo aleatorio, intentan maximizar el grado de aprendizaje, tratando de aprender lo máximo posible con el menor número de instancias entrantes (J. L. Balcázar 2001)(S. Tong 2000). Más ejemplos de técnicas de muestreo son los numerosos modelos basados en clúster citados en (Jun Zheng 2010).

  El modelo **Online Incremental SVM** (Jun Zheng 2010) trata de mejorar, ayudándose de *ILVQ*, las máquinas de soporte vectorial en tres aspectos principales:

  - realizar aprendizaje incremental sobre dominios online;
  - resolver problemas de gran escala o de dinámica no estacionaria,
  - reducir el número de SVs.

  Otro modelo, **Growing Neural Gas SVM** (Ondrej Linda 2009), se ayuda del algoritmo GNG para extraer la información topológica de los grupos de datos, y proveérsela a la máquina de soporte vectorial como conjunto de datos de entrenamiento. De esta forma, pueden resolverse problemas de gran escala mediante SVM, utilizando subconjuntos de entrenamiento que reducen significativamente el tamaño de los datos de entrada, sin apenas afectar al rendimiento.

- Las técnicas de clustering agrupan las instancias similares con el fin de eliminar la información redundante o de utilizar los prototipos a partir de los grupos creados para representar los datos originales. Pueden utilizarse varios tipos de agrupamiento para seleccionar instancias de datos pertinentes cerca del límite de separación o para crear prototipos de puntos de datos (H. Yu 2003), (B. Jin 2006), (S. W. Xiong 2005).

2.3.2 Modelos de SVM incrementalizados existentes en la literatura.

Una forma de aplicar las SVM's a problemas de aprendizaje en grandes conjuntos de datos, así como a conjuntos de datos no estacionarios, es su incrementalización. A continuación se citan varios ejemplos.

Syed, Liu y Sung (Syed 1999) propusieron un modelo incremental SVM (ISVM) donde se re-entrenaba una SVM nueva, en la que se combinaba los nuevos ejemplos con los viejos vectores de soporte. Dicho modelo simplemente resolvía los problemas que le surgen a un algoritmo batch al trabajar en un dominio incremental. Estos autores, a su vez proponen un método que no necesita ser reentrenado para elegir los vectores de soporte (H. L. Nadeem Ahmed Syed n.d.). En dicho modelo, una vez descartados los SVs, no se vuelven a considerar. Consiguen un modelo empíricamente demostrado que soluciona el problema del concept drift.

En el modelo de T.Poggio y G. Cauwenberghts (T.Poggio 2000) se ofrece una versión computacionalmente eficaz y de esquema simple basado en aprendizaje incremental y des-aprendizaje decremental. Lamentablemente la aceptación de dicho algoritmo en la comunidad de aprendizaje automático ha sido algo marginal, por no hablar de que sigue siendo un gran desconocido para los profesionales potenciales.

Rüping Stefan (Stefan 2001), propone un modelo en el que los vectores de soporte antiguos son más costosos, mediante la adición de una constante que puede variar con diferentes problemas. Con ello, consigue hacer frente a los cambios de la función objetivo, que son resultado de las configuraciones de aprendizaje graduales.

Laskov et al. (Laskov 2006) propuso un algoritmo de aprendizaje en línea para vectores de soporte. Sin embargo, el algoritmo sufría por requerir demasiado coste computacional espacial.

2.4. Técnicas incrementales estudiadas:

2.4.1 Estudio de los clasificadores de Descenso de Gradiente Estocástico

Pese a su pobre rendimiento en tareas de optimización, los algoritmos de aprendizaje basados en Descenso de Gradiente Estocástico (SGD) son conocidos por su gran rendimiento en el aprendizaje (Olivier Bousquet 2008). Los algoritmos de gradiente estocástico (*GD*) se han asociado históricamente a los algoritmos de backpropagation de las redes neuronales multicapa, que pueden presentar grandes retos en problemas no convexos (de minimización). También son notoriamente difíciles de depurar, ya que a menudo parecen funcionar a pesar de la presencia de errores.

Por ello, es útil ver cómo SGD realiza problemas convexos simples como SVMs. En (Bottou, *SGD* 2012) se proponen ejemplos de código que ilustran las buenas propiedades de los algoritmos SGD en problemas lineales de gran escala.

i. Aprendizaje mediante descenso de gradiente

Para explicar el uso de SGD en aprendizaje automático, debemos explicar previamente los fundamentos de descenso de gradiente (*GD*), algoritmo a menudo propuesto para minimizar el riesgo empírico $E_n(f_w)$ (por ejemplo en (Rumelhart D.E. 1986)).

Consideremos, en primer lugar, una configuración sencilla de aprendizaje supervisado, donde cada ejemplo *z* es un par (x, y) compuesto de una entrada *x* arbitraria y una salida escalar y. Consideramos una función de pérdida $l(\hat{y}, y)$que mide el costo de la predicción $\hat{y}$ cuando la respuesta real es y, y elegimos una familia *F* de funciones de pesos $f_w(x)$.

El objetivo que persigue *GD* es la búsqueda de la función $f \in F$ que minimice la pérdida $Q(z,w) = l(f_w(x), y)$ en los ejemplos.

Por un lado, el riesgo empírico $E_n(f)$ mide el rendimiento sobre el conjunto de entrenamiento. Por otro, el riesgo esperado $E(f)$ mide el rendimiento de la generalización, es decir, el rendimiento esperado en ejemplos futuros.

En GD, cada iteración actualiza los pesos w a partir del gradiente de $E_n(f_w)$.

$$w_{t+1} = w_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_w Q(z_i, w_t) \quad (2.4.1)$$

γ es el coeficiente de mejora (o aumento) en la función, y debe ser adecuadamente elegido

ii. Aprendizaje mediante Descenso de Gradiente Estocástico (SGD)

El algoritmo de descenso de gradiente estocástico (SGD) es una simplificación de GD en la que, en lugar de calcularse el gradiente de $E_n(f_w)$ con exactitud, cada iteración estima el gradiente basándose en un único ejemplo z_t escogido al azar:

$$w_{t+1} = w_t - \gamma_t \nabla_w Q(z_i, w_t) \quad (2.4.2)$$

El proceso estocástico $\{ w_t, t=1, \dots \}$ depende de ejemplos recogidos al azar en cada iteración. Se espera que la fórmula (2.4.2) se comporte como (2.4.1), a pesar del ruido introducido por el procedimiento simplificado de SGD.

La convergencia (hacia el óptimo) del descenso de gradiente estocástico se ha estudiado extendidamente en la literatura referente a la aproximación estocástica. El teorema de Robbins-Siegmund (Robbins y Siegmund, 1971) proporciona los medios para establecer la convergencia casi segura en condiciones suaves (Bottou, Online Algorithms and Stochastic Approximations 1998), teniendo en cuenta casos en que la función de la pérdida no es siempre diferenciable.

Cuando la estimación inicial de pesos w_0 está lo suficientemente cerca de su óptimo, y cuando γ es suficientemente pequeño, SGD obtiene convergencia lineal (Dennis J.E. 1983).

La velocidad de convergencia de SGD se encuentra limitada por la aproximación de ruido del gradiente real. Cuando el coeficiente de mejora γ disminuye muy lentamente, la varianza de la estimación del parámetro peso w_t disminuye también lentamente. En caso contrario, cuando disminuyen con demasiada rapidez, los parámetros de estimación w_t se toman mucho tiempo para acercarse al óptimo.

2.4.2 Variante incremental de LVQ (ILVQ)

Como se ha visto, *LVQ* es una técnica de aprendizaje supervisado basada en la cercanía de los prototipos. En la literatura pueden encontrarse muchas variantes de LVQ. Sin embargo, los métodos basados en LVQ tienen algunas deficiencias. En primer lugar, se necesita de un usuario para inicializar el valor de los prototipos. Por tanto, el rendimiento del aprendizaje se verá afectado por los valores iniciales. Además, el número de prototipos de cada clase es, arbitrariamente, determinado sin ningún conocimiento previo. Por último, estos métodos no pueden manejar el conjunto de datos incremental.

En (S. F. Ye Xu 2009), se propone un método autónomo de aprendizaje supervisado llamado Incremental Learning Vectorial Quantization (ILVQ) para hacer frente a las deficiencias mencionadas antes. El objetivo es desarrollar una red que pueda funcionar de forma autónoma en un ambiente no estacionario de datos. La principal contribución de ILVQ es que aprende prototipos gradualmente no sólo dentro de una clase (within-class incremental learning), sino que también aprende progresivamente el número de clases durante el entrenamiento (between-class incremental learning).

Otra contribución es que no necesita usuarios para predeterminar el número o valor de los prototipos antes de aprender. La propuesta ILVQ puede crecer de forma dinámica, aprendiendo el número de prototipos que cada clase necesita para resolver el problema, y ajustar el número y el valor de los prototipos durante el entrenamiento. La tercera contribución de ILVQ es la eliminación de ruido en los datos de entrada. Es decir, el ruido o los valores extremos entre los patrones de entrada en el conjunto de entrenamiento tienen poca influencia en el rendimiento de ILVQ.

i. Aprendizaje con ILVQ

Como se menciona arriba, con ILVQ se propone un método para hacer frente a las deficiencias de los clasificadores típicos. Para el cumplimiento de los objetivos expuestos arriba, se propone ILVQ de la siguiente forma (ver **figura 2.4.2.1**).

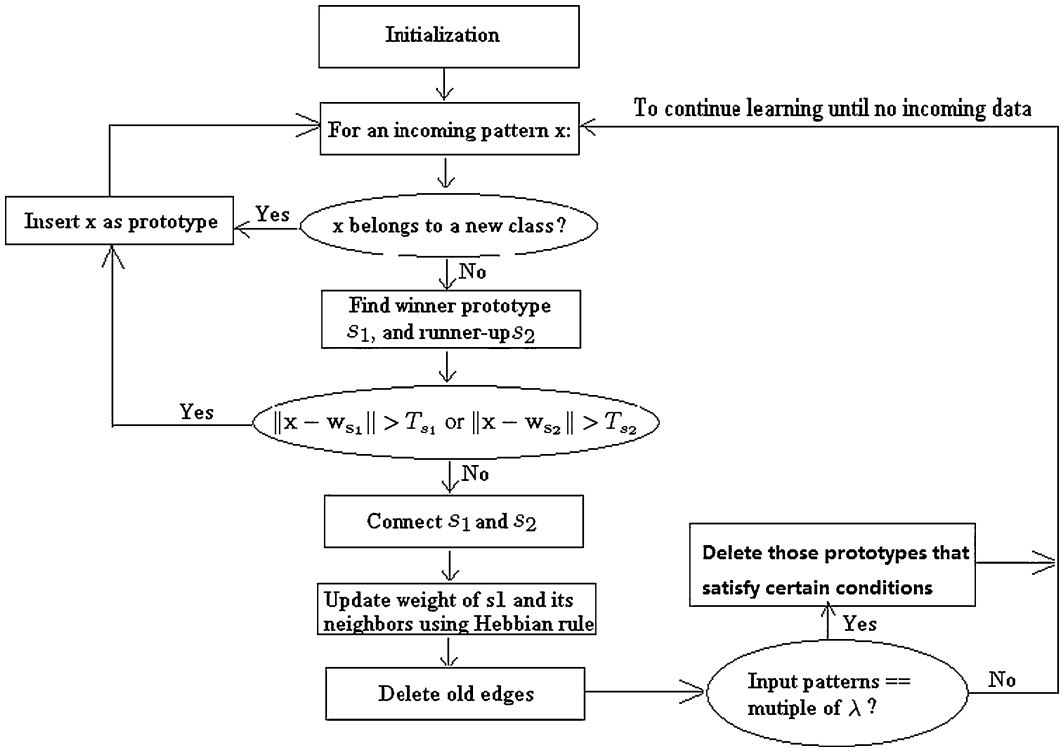


Figura 2.4.2.1: *Proceso de aprendizaje de ILVQ. Extraído de (F. S. Ye Xu 2010).*

En primer lugar, se utilizan las dos primeras instancias que entran al algoritmo para inicializar el conjunto de prototipos G. Para todo vector de entrada **x**, primero hay que justificar si **x** pertenece a una clase que nunca se haya aprendido antes. Si **x** es de una clase nueva, se acepta como nuevo prototipo (between-class incremental learning). De lo contrario, se busca al ganador $\mathbf{s_1}$ y al subcampeón $\mathbf{s_2}$ de los vectores de entrada.

$$winner = argmin_{\mathbf{w_c}} \in G \parallel \mathbf{X} - \mathbf{W_c} \parallel$$

$$runnerup = argmin_{\mathbf{w_c} \in G \setminus \{winner\}} \parallel \mathbf{X} - \mathbf{W_c} \parallel$$

Si la condición de inserción es cierta (ver **figura 2.4.2.1**) , se inserta **x** como prototipo en G (within-class incremental learning). Si la condición de inserción no se cumple: se actualizan las relaciones topológicas entre ganador y subcampeón y los pesos de los vecinos del ganador. Además, se eliminan las conexiones de $\mathbf{s_1}$ que superan la edad máxima permitida y los prototipos vecinos (es decir, los conectados a $\mathbf{s_1}$) cuyos patrones de entrada sean múltiplo de lambda (parámetro establecido a priori).

Aprendizaje y actualización incremental de prototipos

Con el fin de cumplir con la tarea de aprendizaje incremental, ILVQ debe crecer de forma dinámica. Por otro lado, no se debe tomar una política de inserción permanente, ya que puede dar lugar a una inserción interminable y sobreajuste (overfitting). Por ello, es importante la decisión de detención o inserción de nuevos prototipos. Como ya se ha mencionado, se distinguen dos tipos de aprendizaje: between-class y within-class.

- Between-class se basa en el aprendizaje de instancias de entrenamiento cuya clase no ha sido aprendida previamente. Para realizarlo, ILVQ añade a G directamente las instancias cuya clase sea 'nueva'. Esta política permite ILVQ aprender prototipos de distintas clases, sin que el número de clases deba ser especificado a priori.
- El aprendizaje within-class se basa en determinar si un prototipo cuya clase exista debe ser insertado a G. Para ello, se tiene en cuenta la distancia entre el vector de entrada y los prototipos de G $\mathbf{s_1}$ (ganador) y $\mathbf{s_2}$ (subcampeón) con dicho vector (ver **figura 2.4.2.1**).

Para aquellas clases que etiquetan al menos a un prototipo contenido en G, se encuentran $\mathbf{s_1}$ y $\mathbf{s_2}$ con el vector de entrada. Se utiliza un umbral T para tomar la decisión de inserción: si $|w_{winner} - x| > T$, inserta **x**. Para no perder prototipos no directos que sean útiles, también se tiene en cuenta la distancia entre x y el subcampeón: si $|w_{runnerup} - x| > T$, inserta **x**. ILVQ, utiliza una técnica umbral de distancia autoajustable (se ajusta al valor de T_i, cuando el prototipo i se convierte en ganador o subcampeón de un patrón de entrada).

Para ajustar el valor de los prototipos, el ganador deberá moverse cerca o lejos de la entrada, dependiendo del valor de su etiqueta. Los prototipos vecinos de $\mathbf{s_1}$ cuya etiqueta es diferente a la de $\mathbf{s_1}$ podrían ser patrones de ruido del conjunto de entrenamiento. Por ello, se actualiza estos prototipos de acuerdo a la regla de Kohonen (Kohonen 1990). Así, dichos prototipos se moverán de forma opuesta a $\mathbf{s_1}$. Es decir, si el ganador es movido hacia **x**, esos prototipos se alejarán de **x** (ver **figura 2.4.2.2**).

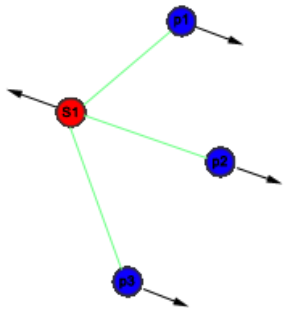


Figura 2.4.2.2: Prototipos p (clase azul) de distinta clase que su vecino (conexiones representadas por aristas verdes) $\mathbf{s_1}$ (clase roja), moviéndose alejándose de $\mathbf{s_1}$ según la regla de Kohonen.

Tasa de Aprendizaje

En ILVQ, las tasas de aprendizaje [η_1 y η_2 en (F. S. Ye Xu 2010) y (S. F. Ye Xu 2009)] afectan a la medida en que el prototipo ganador y los vecinos del ganador se mueven más cerca o más lejos del patrón de entrada. Se adopta un método utilizado por algunos de los mismos autores en (Shen F 2008) para ajustar la tasa de aprendizaje dependiendo del ciclo de aprendizaje. La posición del prototipo es más estable cuando se convierte en ganador más y más veces durante el entrenamiento, es decir, se reduce con ello la tasa de aprendizaje relativa a dicho prototipo.

La tasa de aprendizaje se descompone poco a poco, satisfaciendo la condición de aproximación estocástica de Wasan en (Wasan 1969) que asegura que ILVQ logra un buen balance entre plasticidad y la estabilidad (tradeoff entre exploración y explotación, o dicho de otro modo, 'intuir' cuando parar de buscar mejores resultados).

Estrategia para la eliminación de ruido

En (F. S. Ye Xu 2010) se menciona que el establecimiento de una estructura topológica razonable es beneficioso para detectar el ruido en el conjunto de prototipos, basándose en (Shen F 2008). Por ello, ILVQ aplica la regla topológica vista en (Villegas M 2008) para conectar dos prototipos cuando se convierten en ganador y subcampeón. La regla se sustenta del hecho de que en un entorno incremental, los prototipos no puede seguir siendo vecinos de sus relaciones constantemente (sobre todo cambiarán las relaciones más tempranas al recibir nuevos patrones de entrada). Para ello, ILVQ implementa un mecanismo de edad donde se retiran las conexiones que no han sido actualizadas recientemente (edad mayor que el parámetro AgeOld definido por el usuario).

Los prototipos que rara vez se convierten en ganador o subcampeón durante el proceso de aprendizaje tienden a ser ruido o valores atípicos. Por ello se eliminan dichos prototipos con probabilidad de ruido. Los prototipos con uno o ningún vecino topológico tenderán a desaparecer al envejecer todas sus conexiones.

ii. Resultados en la literatura con ILVQ

ILVQ crece gradualmente y almacena los prototipos aprendidos de tal forma que pueda realizarse bien la tarea de aprendizaje incremental. Se aprovecha de un criterio de umbral basado en la inserción para generar automáticamente el número de prototipos de cada clase. Esto evita tener que determinar antes del aprendizaje dicha cifra, como ocurre en *LVQ*. Por otra parte, ILVQ utiliza una estrategia de compresión para eliminar el ruido que entra en la red durante el aprendizaje. En las **figuras 2.4.2.3 y 2.4.2.4**, se muestra un experimento que ofrece una vista gráfica del funcionamiento de ILVQ.

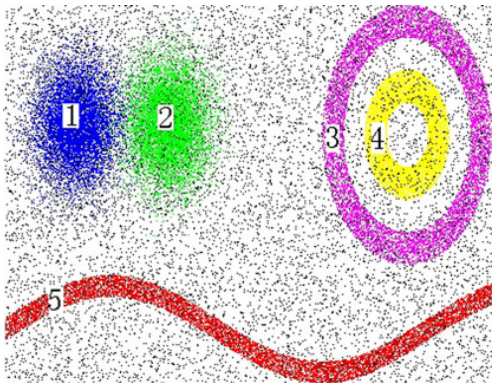


Figura 2.4.2.3: *Conjunto de datos artificial original. Los datos se dividen en cinco clases: clase 1 y clase 2 son dos distribuciones gaussianas, los tipos 3 y 4 son dos anillos concéntricos, y la clase 5 es una curva sinusoidal. Además, se añade ruido al conjunto de datos completo. Extraído de (S. F. Ye Xu 2009).*

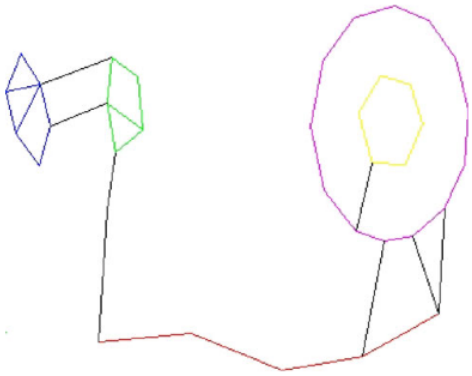


Figura 2.4.2.4: *Estructura topológica de ILVQ resultante de la* ***figura 2.4.2.3****: within-class connections en la clase 1 se expresan como líneas azules, within-class connections de la clase 2 se expresan como líneas verdes, within-class connections de la clase 3 se expresan como líneas de color magenta, within-class connections en la clase 4 se expresan como líneas amarillas, y within-class connections de la clase 5 se expresan como líneas rojas. Las between-class connections (conexiones entre clases) se destacan como líneas negras. Extraído de (S. F. Ye Xu 2009)*

En los experimentos de (F. S. Ye Xu 2010), se comparan algunos clasificadores típicos con ILVQ. Los resultados muestran que la tasa de reconocimiento de ILVQ es mejor en la mayoría de condiciones. Respecto a la compresión, el rendimiento de ILVQ también se muestra bueno. Se muestra como el Clasificador K-Means (KMC) reduce más los conjuntos de entrenamiento (mayor ratio de compresión), pero se destaca que ILVQ cumple mejor el compromiso entre rendimiento en clasificación y eficiencia en almacenamiento. Además, los experimentos demuestran la realización satisfactoria de la tarea de aprendizaje incremental en ILVQ.

2.4.3 Análisis del modelo Online Incremental SVM

Los buenos resultados de las máquinas de soporte vectorial en problemas de clasificación no lineales han colocado *SVM* como uno de los algoritmos de aprendizaje con datos de gran dimensión. La buena capacidad de generalización alcanzada por las Support Vector Machines, dependen de los vectores de soporte (*SVs*).

Como desventaja de estos algoritmos, su entrenamiento consume muchos recursos computacionales, tanto en tiempo y en espacio. Este hecho hace inviable en muchos casos el uso de las máquinas de soporte vectorial en problemas de gran escala.

i. Descripción y funcionamiento de OI-SVM.

OISVM adapta las máquinas de soporte vectorial a una versión online e incremental para hacer frente a los problemas de aprendizaje gradual y de gran escala. El algoritmo se divide conceptualmente en dos partes .

1. Learning Prototypes (LPs): Esta primera parte se encarga de la generación de prototipos.

2. Learning Support Vectors (LSVs): Esta parte obtiene los *SVs* de los prototipos generados en la fase anterior. Combina el nuevo conjunto de prototipos (generado por *LPs*) con los *SVs* almacenados de iteraciones anteriores para formar nuevos *SVs*.

En resumen, *LPs* se encarga de la generación de prototipos, que son unidos a los *SVs* de la anterior iteración por *LVs* para generar los nuevos. Tras ello, *LPs* genera un nuevo conjunto de prototipos. Este procedimiento se repite hasta terminar los ciclos de aprendizaje.

En *OISVM*, además de los dos parámetros de SVM, el parámetro de regularización *C* y el parámetro de base radial función del núcleo γ (FSR), que a menudo se deciden por la búsqueda de la red, hay otros dos parámetros introducidos por el componente *LP*: *OldAge* y λ (El número de ejemplos de entrenamiento es un entero múltiplo de λ). Los dos parámetros son determinados por los usuarios.

Generación de prototipos de aprendizaje (LPs):

Esta parte del algoritmo, recibe las instancias entrantes mediante un buffer de tamaño prefijado por el usuario, según el número y tamaño de ciclos que se desee. Es decir, solo se comienza a generar los LP cuando se llena dicho buffer.

Datos recibidos: $(x_1,d_1), (x_2,d_2), \ldots, (x_n,d_n)$
Prototipos: $(p_1,d_1), (p_2,d_2), \ldots, (p_m,d_m)$ / $m << n$

El algoritmo OISVM genera los prototipos a partir de los datos de entrenamiento mediante el algoritmo incremental *ILVQ* descrito en la sección previa. Detallamos aquí su funcionamiento y los parámetros implicados.

1. Cuando una nueva instancia (x_i, d_i) se extrae del buffer y se recibe como entrada, se consulta su etiqueta de clase d_i.
2. Si es nueva, se inserta la instancia directamente en el conjunto de prototipos LP. Si no lo es, se buscan en el conjunto LP
    a. el vecino más cercano llamado "p_{winner}" y
    b. el segundo vecino más cercano llamado "$p_{runner\text{-}up}$".
3. Se establece una conexión entre p_{winner} y $p_{runner\text{-}up}$ (se pone la 'edad' de la conexión a 0). Este valor de edad se verifica en una fase posterior para todas las conexiones, y si es mayor que determinado parámetro predefinido (*OldAge*), se retira dicha conexión.
4. Si el valor absoluto de la distancia entre x_i y w_{p1} es mayor que cierto umbral $T_{p1,}$ la nueva instancia se insertará en la lista de prototipos en forma de nuevo prototipo. Pasa lo mismo para $p_{runner\text{-}up}$.
5. Si el nuevo ejemplo no pasa a ser un nuevo prototipo, se actualiza el peso de los prototipos en función de la distancia con p_{winner} y $p_{runner\text{-}up,}$ y del valor de su clase.

Después de varias épocas de aprendizaje, los prototipos sin vecinos o con un solo vecino de la misma clase, son eliminados con el fin de eliminar prototipos causados por ruido.

Cálculo del umbral Ti

El umbral T_i de un prototipo se define como sigue por la distancia 'entre la misma clase' y la distancia 'entre clases'. La distancia 'entre la misma clase' ($T_{i\text{-}within}$) representa la distancia media entre *i* y otros prototipos de su entorno que tienen la misma etiqueta que *i*.

$$\frac{1}{N_{label_i}} \sum_{(i,j)} \in E \wedge label_k \neq label_i \left|\left|p_i - p_k\right|\right|$$

La distancia 'entre clases' ($T_{i\text{-}between}$) es la distancia media entre i y otros prototipos de su entorno que no tienen la misma etiqueta que i.

$$min_{(k,i)} \in E \wedge label_k \neq label_i \left|\left|p_i - p_k\right|\right|$$

El umbral será definido como el máximo entre la clase distancia que no es mayor que $T_{i\text{-}within}$, donde la distancia euclídea es utilizada como la métrica y *E* es la lista de conexiones entre prototipos. $N_{label\text{-}i}$ es el número de vecinos del prototipo con etiqueta *i*.

$$\max \left|\left|p_i - p_k\right|\right| \; label_k \neq label_i \leq \mathrm{T}_{i_{within}} : (k, i) \in E$$

Puede verse claramente como el algoritmo propuesto por (Jun Zheng 2010) para *LPs* corresponde exactamente al propuesto por los mismos autores para *ILVQ*. Cosa que, por los buenos resultados mostrados en su artículo, da a suponer el buen funcionamiento de la reducción por clustering del tamaño del problema y de los representantes utilizados en el entrenamiento en clasificación mediante máquinas de soporte vectorial, y el beneficio de las ventajas del concepto de edad y del concepto de regiones de *Voronoi* de *ILVQ* en estos modelos.

Generación de Vectores de Soporte (LSVs):

En el aprendizaje incremental, por lo general, las decisiones de los nuevos ejemplos tienen más peso que la de los más antiguos. Hay que aumentar el peso de prototipos antiguos para que un fallo de aprendizaje, en estos, tenga un coste equiparable al de uno reciente. Esto lo hacen en *OISVM* añadiendo los vectores de soporte de iteraciones anteriores como si fueran prototipos a la nueva formación de SVM.

Esta forma de combinar los SV y los prototipos del nuevo bloque se justifica en (Jun Zheng 2010), donde además se explica la forma en que la influencia de los patrones "pasados" se combina con la de los patrones que se están aprendiendo en este momento.

2.4.4 Técnica de agrupación Growing Neural Gas (GNG)

Un posible objetivo de aprendizaje en los entornos de aprendizaje no supervisado es la reducción de dimensionalidad. Es decir, encontrar un sub-espacio de baja dimensión del espacio vectorial de entrada que contenga la mayor parte o la totalidad de los datos de entrada. El Aprendizaje de Hebb Competitivo (CHL) es un método para contrucción de tales estructuras (de agrupamiento topológico) sobre el que Martinetz y Schulten utilizaron Neural Gas (NG) para encontrar los prototipos. Inspirado por estos, propuso Fritzke el algoritmo Growing Neural Gas (Fritzke 1995) para el agrupamiento y la adquisición de prototipos.

A continuación, como paso previo a la descripción de GNG y su algoritmo, se introducen brevemente los conceptos relacionados con el aprendizaje de Hebb competitivo que el algoritmo utiliza.

i. Conceptos previos: Aprendizaje de Hebb y Triangulación de Delaunay

El aprendizaje competitivo de Hebb (CHL) asume un número de puntos centros en $\mathbf{R}^n$, e inserta conexiones topológicas entre ellos sucesivamente por medio de los datos de entrada de una distribución $P(\xi)$. El principio de este método es:

Para cada señal de entrada 'x', conectar los dos centros más cercanos (medidos por la distancia euclídea) con una enlace.

Como resultado se obtiene el sub-grafo de la triangulación de Delaunay (**Figura 2.4.4.1a**) correspondiente al conjunto de puntos centrales. La **Figura 2.4.4.1b** muestra la "triangulación de Delaunay inducida". Esta se limita a las zonas de la entrada de espacio R^n donde $P(\xi) > 0$. En otras palabras, el sub-grafo no establece enlaces si no hay instancias de entrada entre sus puntos. La "triangulación de Delaunay inducida" ha demostrado conservar óptimamente la topología (la representación de la red) de una manera muy general.

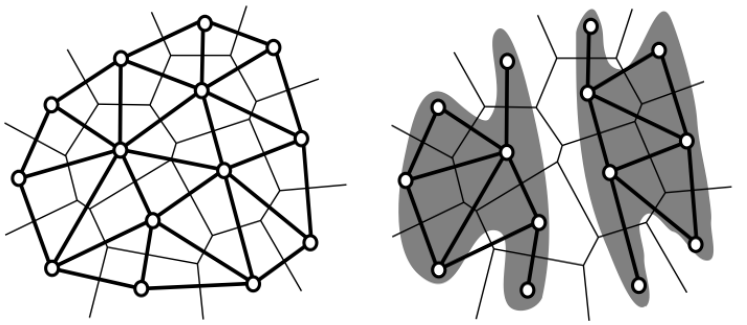


Figura 2.4.4.1a: *Dos formas de definir la cercanía entre un conjunto de puntos. a) La triangulación de Delaunay (líneas gruesas) conecta los puntos que tienen polígonos vecinos de Voronoi (líneas finas). Esto se reduce a los puntos que tienen una baja distancia euclídea al conjunto de puntos dado.* **Figura 2.4.4.1b***: La triangulación de Delaunay inducida (líneas gruesas) se obtiene mediante el enmascaramiento de la triangulación de Delaunay original con una distribución de datos $P(\xi)$ (sombreada). Dos centros están conectados sólo si la frontera común de sus polígonos de Voronoi se encuentran al menos parcialmente en una región en la que $P(\xi) > 0$ (adaptación cercana a Martinetz y Schulten 1994). Imágenes extraídas de* (Fritzke 1995).

Sólo los centros situados en el subconjunto de datos de entrada o en sus proximidades desarrollan las aristas (enlaces). Los otros son inútiles para el propósito de la topología de aprendizaje y a menudo se denominan unidades muertas. Para hacer uso de todos los centros, estos deben ser colocados en las regiones de R^n donde $P(\xi)$ es diferente de cero. Esto podría realizarse por cualquier procedimiento de cuantificación vectorial (VQ). Martinetz y Schulten propusieron el método VQ llamado Neural Gas, que dice así:

Para cada dato de entrada 'x', adaptar los k centros más cercanos donde k decrece a partir de un gran valor inicial hasta un valor final pequeño.

La combinación de NG y CHL descrita anteriormente es un procedimiento efectivo de aprendizaje de topologías. Incluso, para el caso de una combinación simultánea entre CHL y NG, Martinetz y Schulten ya utilizaron un esquema de envejecimiento de conexiones con el fin de eliminar las conexiones obsoletas por el movimiento de los centros. Sin embargo, un problema en las aplicaciones prácticas es la determinación a priori de un número adecuado de centros. La naturaleza del algoritmo de NG requiere una decisión de antemano y, si el resultado no es satisfactorio, las nuevas simulaciones tienen que realizarse a partir de cero.

ii. Descripción y funcionamiento de GNG

GNG es un prototipo basado en la combinación simultánea de CHL y NG de cuantización vectorial, donde los vectores (prototipos) se van añadiendo y borrando de forma dinámica en función de la edad de las conexiones. La topología generada por CHL no es una característica opcional de GNG como lo es para el modelo NG sino un componente esencial que se usa para dirigir la adaptación local (los vectores más próximos se ven "atraídos" hacia el patrón que se aprende), así como la inserción de los centros.

GNG es capaz de aprender la topología de los datos para un número arbitrario de dimensiones. Este nuevo conjunto de datos reducido, extraído como conocimiento topológico, es una aproximación a la distribución de datos original.

Algoritmo de Growing Neural Gas

GNG se basa en la actualización continua de las neuronas (o unidades). Es decir, en añadir sucesivamente nuevas neuronas, a una red inicialmente pequeña, mediante la evaluación de las medidas estadísticas locales recogidas durante los pasos previos de la adaptación. En el enfoque aquí descrito tiene una dimensionalidad que depende de los datos de entrada.

La red estudiada consiste en lo siguiente:

- un conjunto A de neuronas (o unidades). Cada neurona $c \in A$ tiene asociado un vector de referencia (posición) $w_c \in \mathbf{R}^n$. Los vectores de referencia, llamados así en la literatura de Growing Neural Gas, determinan la posición de las neuronas correspondientes en el espacio de entrada.
- un conjunto N de conexiones (o enlaces) no dirigidas entre pares de neuronas. Dichas conexiones no son ponderadas (sin pesos ni coste). Su único propósito es el de definir la estructura topológica. Así, los vecinos no están determinados por la distancia euclídea sino por una conexión directa mediante sus enlaces.

Se expone y se explica el algoritmo de Growing Neural Gas a continuación:

0. Se dispone de un número (posiblemente infinito) de datos de entrada n-dimensionales que obedecen a la función de densidad probabilística $P(\xi)$.
1. Se comienza con dos unidades *a* y *b* en posiciones aleatorias w_a y w_b de R^n.
2. Generamos un dato de entrada ξ de acuerdo con $P(\xi)$.
3. Se encuentra las unidad más cercana s_1 y la segunda unidad más cercana s_2.
4. Incrementamos la edad de todas las conexiones de s_1.
5. Se crea una variable local con el valor de la distancia al cuadrado entre el nuevo dato entrante y la unidad más cercana encontrada:

$$\Delta\text{error}(s_1) = \| w_{s1} - \xi \|^2$$

La acumulación de las distancias cuadráticas durante la adaptación, ayuda a identificar las unidades situados en zonas del espacio de entrada, donde la entrada de datos causa mucho error. Para reducir el error, se insertan en dichas regiones las nuevas unidades.

6. Movemos s_1 y sus vecinos topológicos directos hacia ξ en múltiplos de $\in_b$ y $\in_n$, (que son parámetros del algoritmo) respectivamente, con la distancia total:

$$\Delta w_{s1} = \epsilon_b(\xi - w_{s1})$$
$$\Delta w_n = \epsilon_n(\xi - w_n) \text{ for all direct neighbors } n \text{ of } s_1$$

Esto conduce a las neuronas a moverse hacia áreas del espacio de entrada donde los datos provienen de $(P(\xi) > 0)$.

7. Si s_1 y s_2 están conectados, se pone a cero la edad de dicha conexión. Si la conexión no existe, se crea. La inserción de las conexiones entre la unidad y la segunda unidad más cercanas, con respecto a dato de entrada, genera una conexión de "triangulación inducida de Delaunay" (**Figura 2.4.4.1b**) con respecto a la posición actual de todas las unidades.
8. Se eliminan las conexiones de edad mayor que a_{max}. Si esto ocasiona que alguna neurona se quede sin conexiones, se elimina esta. Este paso es necesario para deshacerse de las conexiones que ya no forman parte de la "triangulación inducida de Delaunay". Esto se consigue con la combinación entre el envejecimiento de las conexiones locales de la unidad más cercana (paso 3), y el reseteo de la edad de los enlaces que conectan s_1 y s_2 (paso 6). Esto hace que en caso de ruido, al cabo de cierto tiempo dicha información ruidosa se olvide al perder sus conexiones.
9. Si el número de datos de entrada generados hasta ahora es un múltiplo entero del parámetro λ, se inserta una nueva unidad:
   - Determinamos la neurona *q* con el error acumulado máximo.
   - Se inserta una nueva unidad *r* a medio camino entre *q* y su vecino *f*. Siendo *f* el vecino de *q* con mayor valor de error:

$$w_r = 0.5\ (w_q + w_f).$$

   - Se enlaza la nueva unidad *r* con las unidades *q* y *f*, y se elimina la conexión original entre *q* y *f*.

- Se disminuyen las variables de error de *q* y *f* multiplicándolas por una constante $\propto$. Se inicializa la variable de error de *r* con el nuevo valor de la variable de error de *q*.

10. Se reducen todas las variables de error multiplicándolas por la constante *d*.
11. Si el criterio de parada (por ejemplo: el tamaño de la red) no se ha cumplido, se vuelve al paso 1.

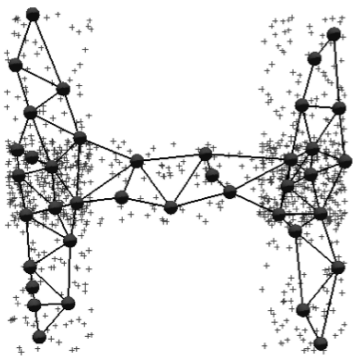
iv. Resultados en la literatura y ventajas de GNG

Las ventajas del algoritmo GNG respecto a otras técnicas de agrupación son las siguientes:

- No requiere una especificación por adelantado del tamaño de la red (número de neuronas) como es el caso de K-medias.
- Su robustez ante el ruido en los datos de entrenamiento.
- A diferencia de otros, el número de neuronas (unidades) de la red crece continuamente dentro de los datos de entrenamiento hasta un criterio de convergencia especificado por el usuario.
- Además, no hay que fijar un radio de clúster previo, como en el caso de la agrupación de vecino más cercano. El patrón se le asigna a la neurona más próxima, por lo que el radio se ajusta continuamente para cada neurona cuando el tamaño de la red crece.

La naturaleza iterativa del algoritmo GNG lo hace adecuado para la tarea de aprender la topología de grandes conjuntos de datos. Las **figuras 2.4.4.2** y **2.4.4.3** muestran ejemplos de la topología aprendido por el algoritmo GNG.

"Growing Neural Gas" es capaz de hacer explícitas las relaciones topológicas importantes de las señales de entrada en una distribución dada $P(\xi)$.



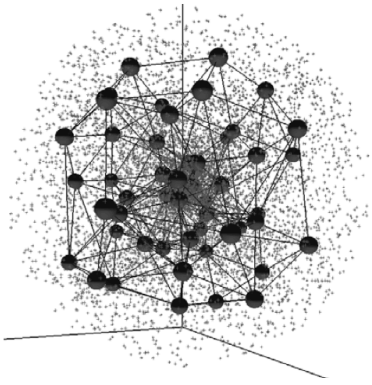


Figura 2.4.4.2. *Se muestra el resultado de una distribución aproximada en 2D con la estructura GNG. Imagen extraída de* (Ondrej Linda 2009).

Figura 2.4.4.3. *Se visualiza la topología de aprendizaje de una distribución de datos esférica en 3D. La esfera está compuesta por 10000 neuronas. La estructura GNG se ha aproximado con 50 neuronas. Imagen extraída de* (Ondrej Linda 2009).

En la **Figura 2.4.4.4** se ilustra las diferencias entre el modelo propuesto y de la red NG original. Aunque la topología final es bastante similar en ambos modelos, las etapas intermedias son bastante distintas. Ambos modelos son capaces de identificar los grupos en la distribución dada. Sin embargo, sólo el modelo "Growing Neural Gas" podría seguir creciendo para descubrir grupos aún más pequeños (que no están presentes en este ejemplo en particular).

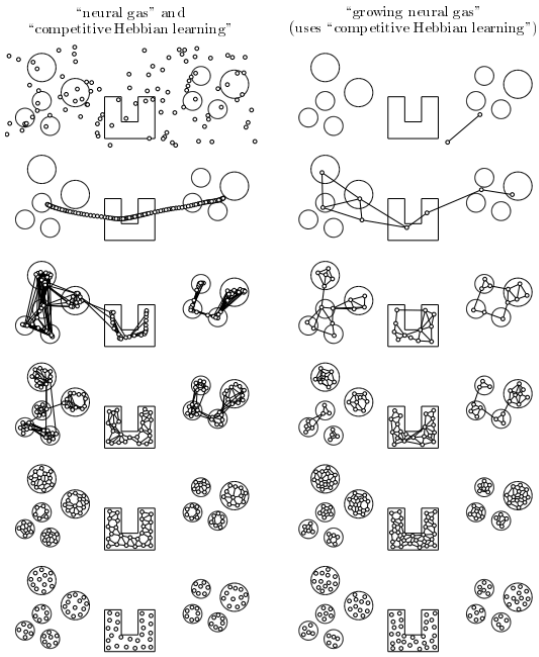


Figura 2.4.4.4: *La red NG / CHL de Martinetz y Schulten (1991) y el modelo "Growing Neural Gas" se adaptan a una distribución de probabilidad de clustering. Se muestran los respectivos estados iniciales (fila superior) y un número de etapas intermedias. Tanto el número de unidades en el modelo NG y el número final de unidades en el modelo "Growing Neural Gas" son 100. La fila inferior muestra distribución de los centros después de 10000 pasos de adaptación. El centro de distribución es bastante similar para ambos modelos, aunque las etapas intermedias difieren significativamente. Imagen extraída de* (Fritzke 1995).

2.4.5 Híbrido Incremental GNG-SVM

Una de las aportaciones de este trabajo es proponer a la comunidad una incrementalización de un algoritmo ofrecido en la literatura como solución a los problemas de clasificación de SVM con grandes conjuntos de datos. Esta incrementalización es paralela a la estudiada en *OISVM*, pero utilizando como base un algoritmo llamado GNG-SVM en vez de ILVQ.

Como se ha visto en el punto anterior, el algoritmo de Growing Neural Gas (*GNG*) es una herramienta robusta para el análisis de agrupaciones, capaz de aprender la topología de los datos.

El SVM por sí mismo, no es apto para el manejo de grandes conjuntos de datos, pues el proceso de entrenamiento de éste depende directamente del tamaño de estos. El objetivo es reducir el conjunto de datos de entrenamiento mediante *GNG*, cosa que también hará reducir el proceso de entrenamiento de SVM. En otras palabras, se trata una técnica que reduce el tamaño de los datos de entrenamiento a través de la topología de aprendizaje (aprendida por *GNG*). La información topológica describe la densidad, así como la forma de la distribución de los datos en el espacio de entrada. Dicho conocimiento es utilizado por las máquinas de soporte vectorial para construir una función de separación óptima.

i. Descripción y funcionamiento de GNG-SVM

GNG-SVM constituye un algoritmo de clasificación de datos híbrido desarrollado específicamente para la clasificación de conjuntos de datos de gran tamaño. El problema de decisión binario con dos clases disjuntas (en SVM) es descrito por (2.4.5.1).

$$S = \{(\vec{x}_i, l_i) \mid \vec{x}_i \in R^m, l_i \in \{-1,1\}, i = 1,2 \dots N\} \qquad \textbf{(Fórmula 2.4.5.1)}$$

Generalmente, un problema de la máquina de aprendizaje puede tener un número arbitrario de clases. Por lo tanto, (2.4.5.1) se puede generalizar para un problema de aprendizaje con *C* clases como sigue:

$$S = \{(\vec{x}_i, l_i) \mid \vec{x}_i \in R^m, l_i \in \{1, \dots C\}, i = 1,2 \dots N\} \qquad (\textbf{Fórmula 2.4.5.2})$$

Aquí m denota la dimensionalidad del problema, N es el número de instancias de entrenamiento y x_i y l_i son el vector de entrada y la etiqueta de clase respectivamente.

Algoritmo de GNG-SVM

GNG-SVM consta de dos fases. En la primera fase, se extrae la topología de los datos mediante la formación de varias instancias del algoritmo *GNG*. El conjunto de datos original se reducirá a esta información topológica extraída. En la segunda fase, se entrena la SVM con el conjunto de datos reducido nuevo. GNG-SVM se presenta en un problema de clasificación multiclase (ver **fórmula 2.4.5.1**).

Para un conjunto de datos de entrada de *S* con las clases *C*, el algoritmo es el siguiente:

1. Se divide el conjunto de datos de entrenamiento *S* en *C* subconjuntos *Si*, uno por cada clase.
$$\{S_1, S_2, \dots, S_c\} \quad \textbf{(Fórmula 2.4.5.3)}$$
2. Se construye un conjunto de *C* instancias del algoritmo *GNGi* del total de instancias de GNG.
$$\{GNG_1, GNG_2, \dots, GNG_c\} \quad \textbf{(Fórmula 2.4.5.4)}$$
3. Se asigna a cada subgrupo S_i su instancia GNG_i correspondiente (según clase).
4. Se entrena cada instancia GNG_i con el conjunto de entrenamiento de S_i hasta que se cumpla cierto criterio de convergencia para ese S_i (por ejemplo, número determinado de iteraciones, número de neuronas en la red, etc.).
5. Se extraen de cada instancia *GNGi* los vectores *W* correspondientes, como prototipos que representan su topología.
$$\{W_1, W_2, \dots, W_c\} \quad \textbf{(Fórmula 2.4.5.5)}$$
6. Se amplía cada vector w_i , asignándole una clase l_i, en cada vector de referencia d_j. Es decir, se añade a cada vector, la clase correspondiente de la instancia GNG_i de la que se ha extraído.
$$W_i^* = \{(\vec{w}_j, l_i)\}_{\,j=1,2\dots N_i} \quad \textbf{(Fórmula 2.4.5.6)}$$
Aquí N_i denota el número de vectores de referencia en el conjunto Wi.
7. Se crea un nuevo conjunto de datos *S** mediante la unión de todos los conjuntos de vectores de referencia etiquetados W_i^*. Este conjunto S*, representa la topología reducida de los datos de entrada.
$$S^* = \bigcup_{i=1}^{C} W_i^* \quad \textbf{(Fórmula 2.4.5.7)}$$
8. Entrenar una SVM con el nuevo conjunto de datos de entrenamiento reducido *S**.

ii. Resultados en la literatura de GNG-SVM

En la literatura de GNG-SVM, el conjunto de datos de entrada se procesa con un algoritmo de *GNG* implementado en C++. Tras ello, el nuevo conjunto se suministra a la librería *SMO* de *WEKA*, que utiliza un kernel polinómico para las pruebas. Los parámetros utilizados para SVM fueron establecidos a colación de las pruebas realizadas en el artículo (J. L. Balcázar 2001).

El rendimiento de GNG-SVM fue probado en test sintéticos, así como en conjuntos de datos del mundo real. Pruebas en los conjuntos de datos sintéticos linealmente separables y no separables demostraron que el algoritmo presentado puede reducir significativamente el tamaño del conjunto de datos de entrada, mientras que preserva la información topológica importante (ver **Figura 2.4.5.1**).

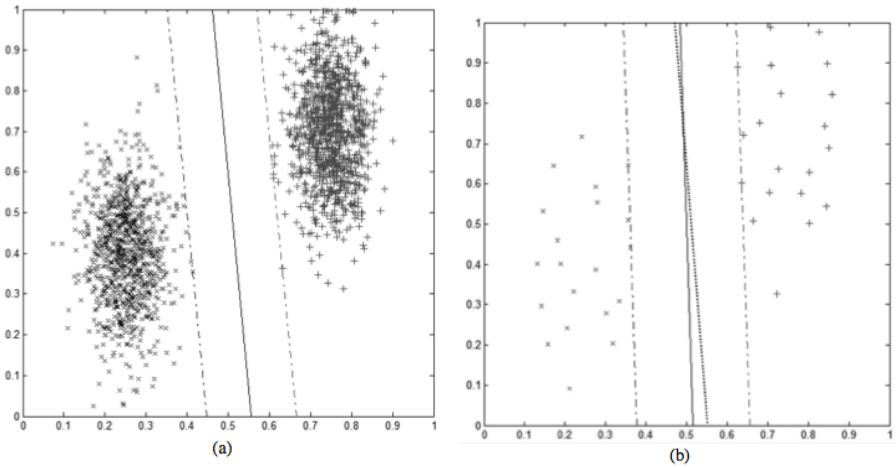


Figura 2.4.5.1: *Margen de separación entre dos clases en SVM, utilizando el conjunto de datos original (a), y utilizando el conjunto de datos reducido. La* ***Figura 2.4.5.1a*** *muestra el margen de separación calculado por SVM en el conjunto de datos de entrenamiento original, que consta de dos clases. El hiperplano de separación está marcado con una línea recta, mientras que los límites del margen se dibujan con líneas de trazos y de puntos. Para una comparación, la* ***Figura 2.4.5.1b*** *ilustra el margen de separación construido por SVM en el entrenamiento del nuevo conjunto de datos reducido. El conjunto de datos original se redujo en 20 neuronas por clase. La línea de puntos en la* ***Figura 2.4.5.1b*** *muestra el hiperplano de separación original a partir de la* ***Figura 2.4.5.1a****. Esta comparación demuestra que mientras que el tamaño del conjunto de datos de entrenamiento se redujo significativamente, el margen de separación construido fue alterado sólo ligeramente. Imágenes extraídas de* (Ondrej Linda 2009).

La precisión de la clasificación de GNG-SVM sobre el conjunto de datos Magia quedó casi igual en comparación con la formación SVM sobre el conjunto de datos original. En el caso del conjunto de datos Shuttle, la clase menos abundante experimentó una disminución significativa de rendimiento, que se atribuyó a la complejidad del problema. Esto demostró que GNG-SVM es adecuado para un determinado tipo de problemas, donde las clases forman cúmulos más separados en el espacio de atributos.

Además, en cuanto al balance entre el tiempo de entrenamiento y la precisión de la clasificación de GNG-SVM, se demuestra que cuanto más tiempo se asigna para la fase de entrenamiento *GNG*, mejor será el rendimiento de SVM (**Figura 2.4.5.2**).

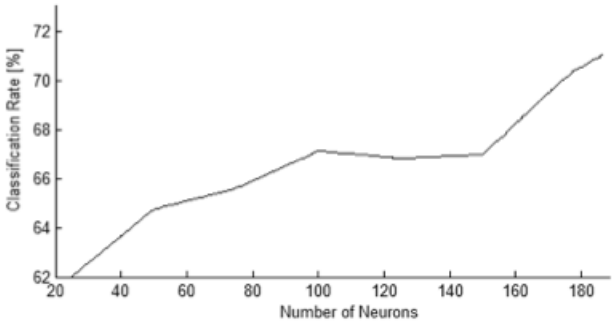


Figura 2.4.5.2: *Tasa positiva de acierto de la clase dos en función del número de neuronas de la red GNG. La tasa de acierto positiva de la clase 2 crece continuamente a medida que el número de neuronas en la red aumenta. En otras palabras, cuanto más tiempo se asigna para la formación del algoritmo GNG, mejor será el rendimiento de SVM. Imagen extraída de* (Ondrej Linda 2009).

iii. Algoritmo incremental de GNG-SVM (IGNGSVM)

El algoritmo mostrado antes prepara a la red SVM para una clasificación eficiente con conjuntos grandes de datos. Dicha técnica ofrece gran agilidad para conjuntos batch, pero presenta grandes inconvenientes si nos planteamos su desarrollo en un entorno de aprendizaje incremental.

Si .quisiéramos obtener la predicción del clasificador en cualquier momento, bastaría con realizar un tratamiento de las instancias de forma aislada y secuencial (de una en una), de forma que el modelo se construya a lo largo del tiempo, es decir, ampliándolo o corrigiéndolo a medida que se extrajese nuevo conocimiento (ver sección **2.1**). A pesar de que *GNG* es un algoritmo continuo, la traducción literal del algoritmo *GNG-SVM* para su incrementalización, supondría la unión de todos los subconjuntos de S, y su clasificación mediante SVM en cada momento en que se quiera obtener una predicción. Esto consume demasiados recursos computacionales como para plantearlo(ver sección **2.3).**

Nuestra solución para el tratamiento incremental, puede ser aplicar el mismo mecanismo visto para el ya estudiado OISVM, el cual utiliza también un método de cuantización vectorial (obtención de ejemplos basado en (Kohonen, 1990)) para crear un subconjunto característico de los datos. Al usar el mecanismo de división en bloques TS, los ejemplos que lleguen tienen que esperar a llenar el bloque de datos para poder ser incluidos en el modelo. Esto hace que sólo esté actualizado para la predicción antes de recibir la siguiente instancia (a la que llena el bloque TS).

A continuación, se exponen los pasos necesarios para la incrementalización del algoritmo, aplicando la metodología de *OISVM*.

1. Se carga un bloque de datos de tamaño |TS| sobre el que aplicar *GNG-SVM*.
2. Dicho bloque TS se subdivide en tantos sub-bloques TS_i como número de clases i haya. Cada bloque contendrá instancias GNG_i con determinado valor de clase i.
3. Los sub-bloques son procesados en instancias distintas de *GNG* y se obtiene el subconjunto topológico S de cada uno (tras añadir la información de clase a cada vector **W** extraído).
4. Se une en un único conjunto S* todos los subconjuntos S.
5. En caso de existir SV's de la pasada iteración (sólo si no es el primer), se mezclan las instancias SV's con las instancias del subconjunto topológico S*, en un nuevo conjunto T_{emp} (Véase cómo hemos tomado la notación utilizada en *OISVM* para nombrar éste nuevo conjunto, así como la utilizada para llamar a los bloques de datos).
6. El nuevo subconjunto T_{emp} es enviado a la máquina de soporte vectorial, que es entrenada con éste tras su vaciado (SVM se reinicializa para tratar cada TS por separado, las topologías anteriores se tienen en cuenta mediante el envío de los vectores SV's de la anterior ronda, que a su vez representa datos de las anteriores).
7. Se extraen los SV's de la SVM, y se almacenan para siguientes iteraciones. Si existe algún bloque más, se vuelve al paso 1. Si no, se termina aquí.

Se puede observar como el algoritmo de incrementalización es idéntico al de SVM con el pre-procesado de *ILVQ* para extraer la topología de los datos, pero usando IGNGSVM para reducir el tamaño de los conjuntos de datos iniciales. El conjunto S* es equivalente al conjunto de instancias G en *OISVM* en lo que a los pasos del algoritmo se refiere.

iv. Propósitos de la implementación de IGNGSVM

Si bien la propuesta de (Ondrej Linda 2009) permite ejecutar en un entorno batch, clasificación SVM con grandes conjuntos de datos mediante el uso de su topología, el método incremental propuesto aquí deberá hacerlo en un entorno incremental.

Las ventajas de utilizar un modelo incremental son numerosas y ya se han tratado en los apartados pertinentes de este estado del arte. Además la incrementalización de este algoritmo permitirá compararlo de forma directa a *OISVM*, algo que por su similar estructura será interesante.

3. HERRAMIENTAS PARA EL ESTUDIO EXPERIMENTAL DE ALGORITMOS DE APRENDIZAJE

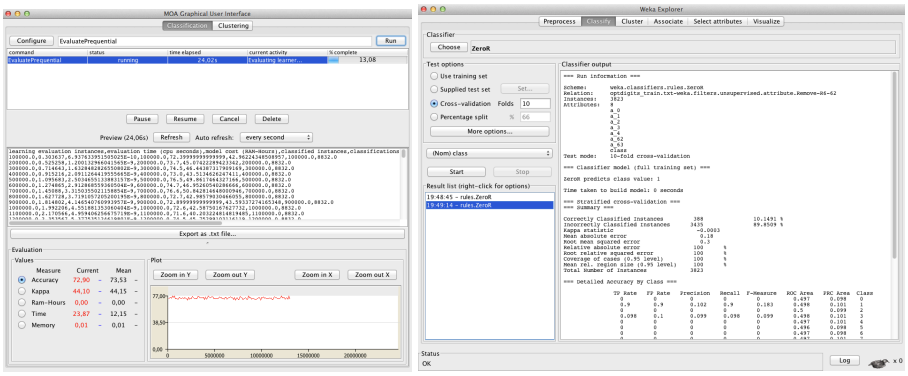
En este apartado se introduce la herramienta utilizada en el proyecto, MOA, para el estudio experimental de los algoritmos de aprendizaje. En primer lugar, se explican los objetivos de su uso, seguido de una breve introducción sobre su funcionamiento y utilización. También se habla brevemente del entorno de análisis WEKA, base a partir de la cual se programó MOA para abordar problemas de clasificación de flujos de datos (data stream).

3.1. OBJETIVOS

Como ya se ha explicado, el objetivo de este proyecto es el estudio comparativo de entre algoritmos novedosos de aprendizaje incremental, así como la codificación y prueba de algunos cuyo código fuente no se encontraba disponible. Dicho estudio se realiza mediante la integración de estos en MOA (Massive Online Analysis), un entorno de software para la implementación y experimentación mediante aprendizaje online en stream de datos.

De forma más detallada, a continuación se separan los fines perseguidos con la utilización de las herramientas de análisis MOA y WEKA:

- Integración de los algoritmos a las herramientas de análisis.
- Validar la correcta integración de dichos algoritmos mediante pruebas existentes en la literatura pertinente a cada uno.
- Realizar, entre los tres algoritmos, un estudio experimental comparativo aplicado sobre Varios conjuntos de datos referenciados en la literatura relativa a aprendizaje de conjuntos grandes de datos, y aprendizaje de datos no estacionarios.



Figuras 3.1 y 3.2: *Interfaz gráfica de MOA y WEKA, respectivamente, sobre un conjunto de prueba de ejemplo. A la derecha, se ejecuta Zero sobre una muestra truncada del train set de optdigits.*

3.2. PLATAFORMAS DE EXPERIMENTACIÓN: WEKA Y MOA

MOA incluye una colección de clasificadores online, así como herramientas para su evaluación. MOA se relaciona con el entorno de análisis del conocimiento WEKA, ambos escritos en Java. Los principales beneficios del uso de tal lenguaje son su portabilidad, y las bibliotecas de apoyo fuertes y bien desarrolladas.

WEKA da nombre a una extensa colección de algoritmos de Aprendizaje Automático útiles para ser aplicados sobre datos mediante las interfaces que ofrece, o para embeberlos dentro de cualquier aplicación. Además, contiene las herramientas necesarias para realizar transformaciones sobre datos, tareas de clasificación, regresión, clustering, asociación y visualización.

i. Aspectos generales sobre uso y funcionamiento de MOA.

El funcionamiento del análisis en MOA se basa en flujos de datos continuos. Un entorno de análisis que trabaje con stream de datos debe tener requisitos diferentes de la configuración batch tradicional (la soportada por WEKA). Los más significativos son los siguientes:

1. Procesar un ejemplo por vez y en una sola iteración.
2. Utilizar una cantidad limitada de memoria.
3. Trabajar en un periodo de tiempo limitado.
4. Estar preparado para predecir en cualquier momento.

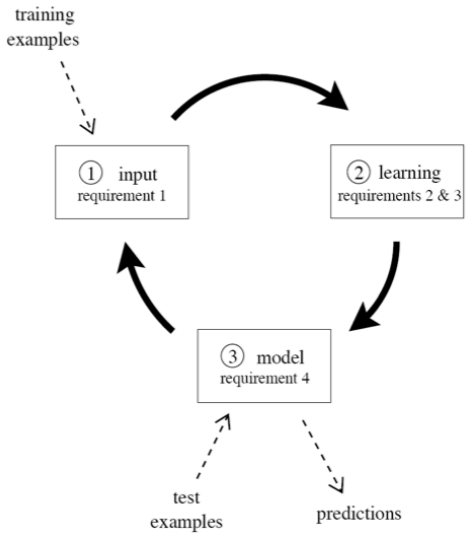


Figura 3.3: *Ciclo de clasificación de un stream de datos. Imagen extraída de la guía de usuario de MOA, a la que se hace referencia en el anexo de este documento.*

La **Figura 3.3** muestra el uso típico de un algoritmo de clasificación incremental en un entorno donde el flujo datos es continuo. En **(1)**, se pasa al clasificador el ejemplo siguiente disponible del stream (requisito 1). En **(2)**, el clasificador procesa el ejemplo y actualiza sus estructuras de datos. Lo hace sin rebasar los límites de memoria establecidos en el mismo (requisito 2), y lo más rápidamente

posible (requisito 3). En **(3)**, el algoritmo está listo para aceptar el siguiente ejemplo. A petición, es capaz de predecir la clase de los ejemplos que no constan en la entrada (requisito 4).

En el aprendizaje batch tradicional, el problema de la escasez de datos es superado analizando y promediando el resultado de distintos conjuntos de datos de train y test. Por otro lado, en el aprendizaje incremental, el problema de hacer preciso un análisis continuo, plantea retos distintos. Una solución consiste en hacer 'capturas' en diferentes momentos para ver hasta qué punto mejora el modelo.

El procedimiento de evaluación de un algoritmo de aprendizaje determina qué ejemplos son usados en el entrenamiento del clasificador, y cuáles son utilizados en las pruebas. El procedimiento utilizado históricamente en el aprendizaje batch, en parte ha dependido del tamaño de los datos de entrada. Con grandes sets de datos suele ser necesario reducir el número de repeticiones del algoritmo, a fin de completar los experimentos en un plazo razonable. Sin embargo, al intentar deducir qué procedimiento de evaluación utilizar sobre un entorno de aprendizaje incremental, uno de los problemas que surgen es cómo construir una buena medida de precisión en el tiempo. Surgen dos enfoques:

- Holdout (retención): Cuando el aprendizaje batch alcanza un tamaño en el que la validación cruzada consume demasiado tiempo, a menudo se retiene un conjunto de entrenamiento para medir su rendimiento. Esto es muy útil cuando la división entre los sets de entrenamiento y de prueba han sido predefinidas, de forma que los resultados de los diferentes estudios se pueden comparar directamente.

- Interleaved Test Then Train o Prequential: Cada ejemplo se puede utilizar individualmente para probar el modelo antes de ser usado en el entrenamiento. Desde aquí, la precisión se puede actualizar de forma incremental. Este esquema tiene la ventaja de no necesitar la retención de ejemplos, haciendo el máximo uso de los datos disponibles. También garantiza un gráfico temporal de precisión liso, ya que cada ejemplo será cada vez menos significativo en la media global.

Como la clasificación stream de datos es un campo relativamente nuevo, estas prácticas de evaluación no están tan bien investigadas y establecidas como lo están en la configuración batch. La mayoría de las evaluaciones experimentales utilizan menos de un millón de ejemplos de entrenamiento. En el contexto de los flujos de datos esto es decepcionante ya que, para ser verdaderamente útil, los algoritmos tienen que ser capaces de manejar flujos de ejemplos potencialmente infinitos.

MOA permite evaluar adecuadamente algoritmos de clasificación de stream de datos en flujos de grandes dimensiones, del orden de decenas de millones de

ejemplos siempre que sea posible, y bajo los límites de memoria explícita. Para ver en más profundidad las explicaciones sobre su uso, éste se detalla en la guía de usuario referenciada en el anexo del trabajo.

ii. Creación de un nuevo clasificador en MOA

Una vez explicado a grandes rasgos el funcionamiento del análisis de un flujo de datos continuo, puede profundizarse explicando la formación de un algoritmo de análisis incremental para MOA. Dicha explicación, ayudará no sólo en la tarea de comprender el proceso realizado por MOA, sino también en la labor de mostrar al lector el trabajo realizado en este proyecto, a nivel de programación, referente a la integración de los algoritmos en la herramienta utilizada.

Siguiendo la sintaxis de java, cada clasificador se encuentra ubicado en una clase separada. Cada clasificador en MOA debe contar con la siguiente serie de elementos y métodos:

- Elementos *Option*: Representan los datos de entrada al análisis, que son introducidos por interfaz o terminal. Por ejemplo, véase **Figura 3.4**.

```
public FloatOption lambdaOption = new FloatOption("lambda", 'l', "Lambda", 16);
public IntOption maxAgeOption = new IntOption("maxAge", 'a',"MaximumAge", 16);
public IntOption tsLength = new IntOption("TS", 't',"LongitudDeBloque",300);
```

Figura 3.4: *Tres entradas numéricas del clasificador integrado en este trabajo OISVM. De arriba abajo, representan los parámetros definidos en la literatura, λ, oldAge, y longitud de los bloques |TS|.*

- Método *resetLearningImpl*: prepara el aprendizaje restaurando las variables globales a su valor por defecto.

```
public void resetLearningImpl() {
    SV = new ArrayList<Prototipo>();
    G = new ArrayList<Prototipo>();

    numberInstances = 0;
    isClassificationEnabled = false;
    this.isBufferStoring = true;
```

Figura 3.5: *Fragmento de código del método resetLearningImpl en OISVM. Se inicializan las listas de prototipos G y SV vistas en* (Jun Zheng 2010)*, así como se pone el valor por defecto en otras variables globales.*

- Método *trainOnInstanceImpl*: Entrena los ejemplos uno por uno. No obstante, también se puede forzar un entrenamiento conjunto mediante la declaración de un buffer de instancias (como se realiza en la clase *WEKAClassifier*). Sin ir más lejos, dicho buffer se utiliza en OISVM para llenar el bloque de datos TS.

- Método *getVotesForInstance*: Dicho método se encarga de clasificar una instancia, recibida como parámetro, en la configuración de ese instante de

tiempo. En el caso de utilizar como clasificador algún tipo de algoritmo incrementalizado, como *OISVM*, se puede llamar directamente a la función de clasificación de instancias del clasificador utilizado de WEKA. En la clase *WEKAClassifier* se proporciona el código básico para realizar esto de forma correcta.

Como es razonable, cada clasificador posee una ruta propia según la jerarquía de carpetas establecida en MOA y WEKA. Dicha ruta deberá ser especificada como paquete de cada clasificador. Es también obvio, que cada clasificador debe pasar por el proceso de compilación antes de su utilización con las herramientas. La utilización de los distintos clasificadores es detallada en el anexo a este proyecto, en la guía de usuario. Por su parte, el proceso de compilación debe realizarse indiferentemente, mediante terminal o interfaz de programación, según comodidad del programador.

En el caso de este trabajo, el proceso de integración de los algoritmos ha sido efectuado mediante la interfaz de programación Eclipse. Por ello, la compilación de todas las clases ha sido de forma automática, efectuada por el programa utilizado.

iii. MOA a partir de WEKA

Una de las estructuras de datos clave utilizadas en MOA es la descripción de una instancia de un flujo de datos (esto vendría a ser el ejemplo de entrenamiento, en el estado del arte). Dicha estructura se toma prestada de WEKA, donde cada instancia es representada por una matriz de valores de coma flotante de doble precisión. Esto proporciona libertad para almacenar todos los tipos necesarios de valores numéricos. Los valores en coma flotante de doble precisión requieren un espacio de almacenamiento de 64 bits (8 bytes). Este detalle puede tener implicaciones en el uso de memoria.

Como se ha descrito, MOA utiliza las clases y el código de WEKA. MOA varía de WEKA en cuanto a su aplicación en streaming de datos, y su condición de clasificación sin necesidad de entrenamiento previo, gracias a las propiedades de la clasificación incremental (ver estado del arte). WEKA se encuentra más orientado a pequeños conjuntos de datos, cuyos valores no evolucionan con el tiempo. Sin embargo, el funcionamiento y la formación de sus clases internas siguen, como se ve a continuación, los mismos patrones.

Por un lado, el uso de WEKA se encuentra muy ligado al de MOA. Es detallado en la guía de usuario que se encuentra anexa en este trabajo. Por el otro, su funcionamiento puede deducirse del conjunto de métodos que forma cada clasificador. A continuación se expone un análisis de los métodos constituyentes de los clasificadores en WEKA, a partir del cual se explica el proceso de análisis de datos, así como se establece una comparativa con el mismo proceso en MOA.

a. Clasificadores en WEKA.

Para crear un clasificador en WEKA, es obvio que lo primero es crear una nueva clase y situarla en el paquete correspondiente de la jerarquía de carpetas (weka/classifiers/). Es importante señalar que cada clasificador deberá heredar ciertas clases de WEKA: *weka.classifiers.*, weka.core*.* y *weka.util*.* . Pese a no haberse descrito, en MOA esto se realiza de la misma forma, con sus respectivas librerías.

A su vez, todos los clasificadores extienden la clase *Classifier*. Si además, el clasificador calcula la distribución de clases también debe extender *DistributionClassifier*.

Todo clasificador tiene que tener los siguientes métodos:

- *buildClassifier*: Construye el clasificador con los parámetros recibidos como instancias. Inicializa el valor de las instancias. Podría establecerse una similitud entre este y *resetLearningImpl (de MOA)*, en que ambos son inicializadores del clasificador. No obstante, en ningún momento esto supone que desde *resetLearningImpl* se realizará una llamada a *buildClassifier*, pues para llamar al segundo necesitamos los datos previamente inicializado, así como instancias preparadas para enviar a la distribución que se creará. Por ejemplo, en OISVM, se llama a *buildClassifier* en cada iteración del entrenamiento (cuando TS alcanza la longitud que le corresponde).

- *classifyInstance*: Clasifica una instancia concreta una vez que el clasificador ya está construido. Devuelve la clase en la que se ha clasificado o *Instance.missingValue* si no se consigue clasificar.

- *distributionForInstance*: Devuelve la distribución p(C|E). Es decir, la probabilidad de que la clase sea C_i dada una instancia E. Cuando se realiza una llamada para predecir con un método de WEKA desde MOA, este es el método al que deberá de hacerse referencia. Por ejemplo, en OISVM se crea un objeto de LibSVM que llama, en primera instancia, a buildClasifier para realizar la distribución del buffer de instancias TS. Luego, cuando desea saber el estado de la clasificación, llama a *distributionForInstance.*

Además, los clasificadores deberán implementar la interfaz *UpdateableClassifier* si tienen un carácter incremental, y *WeightedInstanceHandler* si hacen uso de los pesos de cada instancia.

Por último, también es muy útil conocer las clases *Instance* e *Instances* (del paquete *weka.core*) que son aquellas que almacenan (siempre con doubles) todas las instancias.

iv. Uso de librerías de WEKA

Como se comenta en el estado del arte, no todos los clasificadores online son puramente incrementales.

- Existen clasificadores estáticos que son adaptados, o incrementalizados, para su uso con flujos de datos continuos. Algoritmos de WEKA utilizados desde MOA para procesar un stream mediante su división en varios bloques secuenciales (reentrenando en cada uno).

- Por el contrario, algoritmos incrementales usados en modo batch. Así mismo, pueden estudiarse clasificadores incrementales en WEKA, siempre que el flujo de datos no sea continuo (predefinido).

MOA da la posibilidad de utilizar la librería de WEKA para facilitar el uso de clasificadores estáticos en un entorno incremental. Existen dos clasificadores en MOA que facilitan el uso, y que sirven como ejemplo para utilizar los algoritmos de WEKA, *WekaClassifier* y *SingleClassifierDrift*. En este trabajo se ha utilizado *WekaClassifier* tanto en la fase de validación con *LibSVM* (ver Validación 1, en 5.1), como de ejemplo en el uso de librerías propias de WEKA. Por ejemplo, *OISVM* implica un uso de la librería *LibSVM* mediante llamadas desde el propio clasificador. La sintaxis de programación y varios métodos han sido extraídos aquí, de *WekaClassifier*, a modo de aprovechar recursos ofrecidos por el entorno WEKA. En el anexo, se brinda al usuario la referencia para descargar la guía de usuario de WEKA; ahí se documenta el uso de *WekaClassifier* de forma detallada.

v. Meta-clasificador CVParamaterSelection

El meta-clasificador *CVParameterSelection* sirve para automatizar y optimizar las pruebas en WEKA con cierto conjunto determinado de parámetros. En la **Figura 3.4** puede verse la interfaz de WEKA desde CVParameter Selection. Puede resumirse su utilización como clasificador 'iterativo' que elige la mejor prueba.

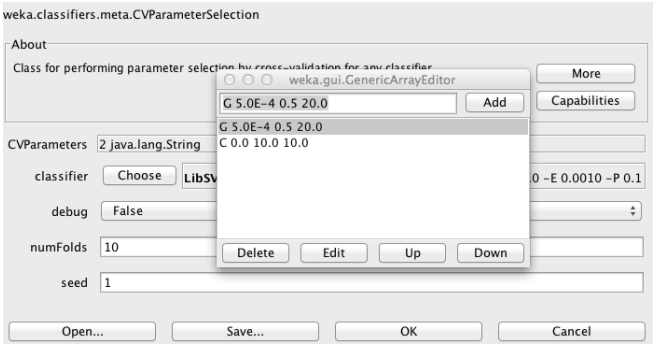


Figura 3.4: *En la primera línea, WEKA hará pruebas de un conjunto sobre LibSVM alternado valores en Gamma, en el intervalo de 0.0005 a 0.5, con 20 pasos. En la segunda, se probarán diez valores de 0 a 10 para el coste.*

4. CODIFICACIÓN Y VALIDACIÓN DE ALGORITMOS

En este apartado se explica el procedimiento seguido en el trabajo desde el paso posterior a la planificación, hasta la validación de los algoritmos. Todo ello obviando, evidentemente, el proceso de documentación que ha sido paralelo a todo el desarrollo del proyecto.

Como complemento, también se hablará a grandes rasgos de la selección de parámetros realizada en la fase de validación. Por último, se comentará brevemente el uso de algoritmos previos y el estudio de publicaciones anteriores a la correspondiente de los clasificadores programados.

4.1. Procedimiento seguido

Para la codificación de los clasificadores, se han utilizado los algoritmos propuestos en la literatura específica de cada una de las implementaciones. El procedimiento seguido consta de 7 fases, enumeradas a continuación:

1. En primer lugar, existió un periodo de adaptación a la herramienta de clasificación incremental *MOA*, así como un estudio previo de la programación de clasificadores en *WEKA* y *MOA*.
2. De forma paralela, se ha recopilado toda la información pertinente a los clasificadores a estudiar. Gracias a dicho proceso de recopilación y estudio se tomó la decisión de qué tercer clasificador estudiar.
3. Obviamente, el proceso de estudio a llevado consigo anclado un proceso de traducción, pues casi la totalidad de la literatura referente al tema tratado no se encuentra en español.
4. Búsqueda de clasificadores ya programados. En el caso de *SGD*, por ejemplo, supuso el estudio de los algoritmos de (Bottou, SGD 2012), así como los incluidos en las librerías de *WEKA* y *MOA*. También se necesito del estudio de un algoritmo previo de *ILVQ*, usado en el algoritmo de *OISVM*. Por otro lado, *OISVM* necesitó un estudio previo de la librería LibSVM de WEKA, así como de otras clases internas del programa de clasificación para la posterior programación.
5. Resumen y comprensión de los algoritmos. Este paso ha tomado cierto tiempo, en el que se entendió y se decidió la forma de adoptar la programación de los algoritmos.
6. Fase de codificación. En ella se codificaron los clasificadores teniendo en cuenta las decisiones tomadas en la fase previa.

7. Fase de validación. Aquí, se han probado los clasificadores con los parámetros de entrada escogidos en la fase de experimentación de la literatura. Esta fase ha servido de depuración, pues supuso la corrección de los clasificadores hasta aproximar al máximo los resultados.
8. Fase de aceptación. Una vez terminada la fase de validación, se elaboran las fichas de presentación y aprobación de los algoritmos propuestos que figuran en el anexo del trabajo. Dichas fichas constan del nombre del algoritmo, de sus referencias bibliográficas, y de los conjuntos de datos y parámetros seleccionados en la fase de validación.

4.2. Selección de parámetros

La selección de los parámetros ha sido fundamental en la fase de validación. Gracias a los parámetros escogidos para probar los conjuntos de datos en la literatura pertinente, ha podido verificarse que los algoritmos programados tienen un similar funcionamiento. Gracias a la obtención de resultados aproximados, ha podido validarse el correcto funcionamiento de lo programado en la fase 6. El hecho de que los resultados no sean idénticos no es discriminativo, ya que las publicaciones utilizadas no detallan completamente el entorno experimental. Es por ello, por lo que deben realizarse ciertas suposiciones sobre valores de los parámetros utilizados, el método de evaluación, pre-procesamiento de los datos, etc.

4.3. Comparación con publicaciones previas

Desde un principio, la fase de estudio ha supuesto la consulta de artículos e informes científicos previos y posteriores a los de los algoritmos a desarrollar. A parte, se han estudiado codificaciones existentes, y valorado su uso de cara a la experimentación.

- En *OISVM* (Jun Zheng 2010) se detectó el uso del algoritmo incremental *ILVQ* (S. F. Ye Xu 2009) (F. S. Ye Xu 2010) cuya codificación fue proporcionada, para crear el primer clasificador, debido a la existencia de una implementación realizada por un alumno de otro trabajo fin de grado. Por ello, el desarrollo del clasificador supuso el estudio previo de las clases proporcionadas para *ILVQ*, así como el estudio de *LibSVM*.

- En el caso de *SGD*, había una implementación previa en *MOA*, con la limitación de que sólo clasifica sobre kernel lineal (conjuntos de datos lineales). Hubo que verificar en los artículos referenciados que es una limitación inherente al algoritmo propuesto por Bottou (Bottou, SGD 2012).

- Por último, en relación al algoritmo creado, *IGNGSVM*, *GNG-SVM* se toma *de* (Ondrej Linda 2009), y se combina con la forma de abordar el problema incremental de (Jun Zheng 2010). El hecho de utilizar la misma filosofía ayuda a crear una comparación directa entre *GNG* (Fritzke 1995) e *ILVQ*. El planteamiento utilizado para el etiquetado de instancias se basa en guardar las clases de los prototipos, que serán necesarias en la posterior clasificación de los subconjuntos con Support Vector Machines. La diferencia fundamental con *ILVQ*, aquí, es que el segundo algoritmo sólo crea conexiones entre neuronas de la misma clase (**ver sección 2**).

A parte de las codificaciones estudiadas, se ha consultado literatura relacionada a los clasificadores propuestos para entender las bases de su funcionamiento. Por ejemplo, (Syed 1999) (Gert Cauwenberghs 2000) (Pavel Laskov 2006) (Stefan 2001) (Abril 2003) (Juan D. Velásquez 2010) fueron consultados en el estudio de las máquinas de soporte vectorial, (David W. Aha 1991) (Giraud-Carrier 2000) (S.M. Lee 2010) (H. L. Nadeem Ahmed Syed n.d.) como estudio previo de los conceptos y problemas del aprendizaje batch e incremental y (A.Tsymbal 2004) (G.Widmer 1996) (Francisco J. Ferrer 2005) (Jesús S. Aguilar 2005) para su aplicación en conjuntos no estacionarios.

5. ALGORITMOS INCREMENTALES ESTUDIADOS

En este apartado se describe el trabajo realizado a lo largo del TFG a nivel práctico. Detalla el proceso de comprensión y codificación de los distintos clasificadores implementados, explica la elección de los parámetros usados en este proceso, y los problemas encontrados.

En el caso de *SGD*, no ha sido necesario realizar proceso de validación del algoritmo, ya que la versión utilizada es una implementación integrada en MOA previa a este trabajo (ver sección **4.3**) .

La notación utilizada en la descripción de los algoritmos es la misma que la vista en el estado del arte, la cual, a su vez es copiada de las fuentes de la literatura.

5.1. DESCRIPCIÓN Y JUSTIFICACIÓN DE LA VALIDACIÓN REALIZADA

A continuación, se describe el tratamiento realizado sobre los conjuntos de ejemplos de entrenamiento, los modelos de evaluación de precisión utilizados, y la tarea de validación realizada en los distintos algoritmos cuya programación y funcionamiento debemos validar.

i. Conjuntos utilizados en la validación

A continuación se describen los cuatro conjuntos utilizados en el proceso de validación de los distintos algoritmos, se explicaran sus propiedades y el origen de los datos. Su utilización está ligada a su uso por los autores de los algoritmos a validar.

Optdigits:

El conjunto *Optdigits,* formado por un fichero de entrenamiento y otro de test, fue publicado por (E. Alpaydin, 1998). Se creó utilizando programas de pre-procesamiento para extraer mapas de bits normalizados de dígitos escritos a mano a partir de un formulario pre impreso. Los mapas de bits resultantes, de 32x32 se dividen en bloques de 4x4 no superpuestos. Se genera una matriz de entrada de 8x8 (64 atributos) donde cada elemento es un número entero en el rango de 0 .. 16. El último valor de cada fila es el dígito al que se refiere (rango 0 a 9) la matriz.

UPS:

La base de datos *USPS* proviene de un conjunto de dígitos escaneados de sobres por el Servicio Postal de Estados Unidos, y es presentada en (Hastie, 2009). Las

cifras originales fueron escaneadas con distintos tamaños y orientaciones. El procedimiento de segmentación realizado por el Servicio Postal causó que algunas cifras fueran mal segmentadas. Por ello, la base de datos fue generada con una tasa de error de un 2 a un 5 %. En la versión original, el conjunto de entrenamiento está compuesto por 7.291 imágenes y el conjunto de pruebas contiene 2007 imágenes, sin embargo la versión que usamos en validación consta de 9298 instancias repartidas uniformemente entre dos conjuntos (train y test). Está compuesto por 256 atributos y diez clases descritas por un número entero de 1 a 10.

Shuttle:

El conjunto de datos *Shuttle* es un conjunto de datos estacionario instancias usado para evaluar los efectos de clústeres irregulares en el proceso de etiquetado (ello lo hace valioso en el estudio y validación de *IGNGSVM* y *OISVM*). Fue creado por Jason Catlett (Catlett, 2006). Los datos proceden de la NASA y se refieren a la posición de los radiadores en el transbordador espacial. El problema parece estar libre de ruidos en el sentido de que las tasas de error son arbitrariamente pequeñas para en un gran conjunto de datos. Contiene 9 atributos numéricos con siete valores posibles de clase (Rad.Flow, Fpv.Close, Fpv.Open, Bypass Alta, Bpv.Close, Bpv.Open). Aproximadamente el 80% de los datos pertenece a la clase 1. Por tanto la precisión predeterminada es de aproximadamente 80%.

Stbal1:

Es un conjunto artificial de datos con dos atributos y una clase binaria. Es utilizado para estudiar el comportamiento de *GNG* a modo de 'capturas' en distintas fases de una iteración, con el objetivo de validar su funcionamiento. En él, los datos se generan a partir de dos distribuciones gaussianas, una por clase. Las varianzas y centros se definen de forma que las distribuciones tienen un área con fuerte solapamiento.

ii. Tratamiento de los conjuntos

Para validar *OISVM* con *Shuttle* y *Usps*, ambos conjuntos tuvieron que ser previamente acondicionados mediante filtros que convirtieran la clase de tipo numérico, a tipo etiqueta, debido a que su descarga de internet no estaba preparada para su procesamiento en *WEKA* (*WEKA Filter NumericToNominal - First*).

Además, para ejecutar las pruebas con *OISVM*, y posteriormente, también con *IGNGSVM*, es necesario partir ambos conjuntos en dos subconjuntos "train" y "test", de la misma forma que se encontraba *Optdigits (ello no presenta problemas al deberse de conjuntos no estacionarios sin concepto de cambio de*

contexto). Se valida el clasificador comprobando su comportamiento sobre el conjunto de test después de haber sido entrenado sobre el conjunto train.

La separación de *Shuttle* y *Usps* en dos conjuntos se ha realizado mediante la aleatorización previa de ambos (*WEKA Filter Randomize –S 42*) y la partición de cada uno en dos conjuntos de igual longitud.

Además, siguiendo los pasos seguidos en (Ondrej Linda 2009), en *IGNGSVM* se transforma el conjunto *Shuttle* unificando las clases de menor aparición en una sola. Como resultado obtenemos que el 80% de las instancias sean de una clase, y el 20% de la otra. Este nuevo conjunto será el usado como *Shuttle* en la fase de estudio de los algoritmos.

iii. Modelo de evaluación utilizado

Para evaluar la calidad de un clasificador usaremos el porcentaje de éxito sobre el conjunto de entrenamiento, obtenido cuando se ha procesado todo el conjunto de datos.

La tarea de evaluación utilizada en la fase de validación, tanto de *OISVM*, como de *IGNGSVM*, es el evaluador proporcionado por el tutor, y programado por otro compañero, *EvaluateChunksTwoFiles*. Este evaluador se encarga de primero entrenar, y luego clasificar, utilizando un fichero de entrenamiento y otro de test (por ello el tratamiento realizado sobre *Shuttle* y *Usps*).

iv. Validación realizada

La validación realizada tiene como objetivo verificar la correcta programación de los algoritmos integrados en este trabajo. En ningún momento se requiere debatir los resultados, por lo que no es necesaria la repetición de las mismas pruebas que, de todas formas, no se encuentran ampliamente explicadas en los artículos relevantes de la literatura. Por ello, precisiones cercanas para pruebas parecidas, así como comportamientos teóricos en procesos de clustering y clasificación observados, serán validados como muestra de un correcto funcionamiento de los algoritmos programados.

5.2. ONLINE INCREMENTAL SUPPORT VECTOR MACHINES

Como se comenta en el estado del arte, el clasificador *OISVM* (Jun Zheng 2010) adapta las máquinas de soporte vectorial a una versión online e incremental para hacer frente a los problemas de aprendizaje gradual y de gran escala.

En este apartado se analiza el algoritmo propuesto por los autores desde el punto de vista de su programación, se interpreta su funcionamiento, se explican las decisiones tomadas en la fase de codificación; los parámetros usados en la validación; y los problemas encontrados.

i. Análisis del algoritmo

Abajo se expone el algoritmo de *OISVM*, extraído de (Jun Zheng 2010) para proceder a su entendimiento.

Algorithm 1. Online incremental SVM (OI-SVM)

1: To simulate incremental learning for off-line data, we partition the whole data into n parts. For online data, the stream data is input to the system directly.
2: Initialize prototype set G, support vector set SV, and edge set E with $\emptyset$.
3: Load one block of data TS into memory. If G is empty, randomly choose two input data from the training set TS, p_1 and p_2, and insert the two data into G. I.e., $G = \{p_1, p_2\}$.
4: Input a new pattern $\xi = (x_i, d_i)$ to the system.
5: Find the winner p_1 and runner-up p_2 by in set G.
6: **if** $N_{label_{x_i}} < 2 or \|x_i - w_{p_1}\| > T_{p_1} or \|x_i - w_{p_2}\| > T_{p_2}$ **then**
7: Insert ξ into set G. Go to Step(4).
8: **end if**
9: **if** $(p_1, p_2) \notin E$ **then**
10: $E = E \cup \{(p_1, p_2)\}$
11: **end if**
12: $age_{(p_1,p_2)} = 0$.
13: **for** The neighbor p_i satisfies $(p_1, p_i) \in E$ **do**
14: Update $age_{(p_1,p_i)} \leftarrow age_{(p_1,p_i)} + 1$
15: **end for**
16: $M_{p_1} \leftarrow M_{p_1} + 1$, $\eta_1 = \frac{1}{M_{winner}}$, and $\eta_2 = \frac{1}{100 M_{winner}}$.
17: **if** $label_\xi = label_{p_1}$ **then**
18: Update $\mathbf{w_{p_1}} \leftarrow \mathbf{w_{p_1}} + \eta_1(\xi - \mathbf{w_{p_1}})$
19: **for** The neighbor p_i satisfies $(p_1, p_i) \in E$ and $label_{p_i} != label_\xi$ **do**
20: Update $\mathbf{w_{p_i}} \leftarrow \mathbf{w_{p_i}} - \eta_2(\xi - \mathbf{w_{p_i}})$
21: **end for**
22: **else**
23: Update $\mathbf{w_{p_1}} \leftarrow \mathbf{w_{p_1}} - \eta_1(\xi - \mathbf{w_{p_1}})$
24: **for** he neighbor p_i satisfies $(p_1, p_i) \in E$ and $label_{p_i} = label_\xi$ **do**
25: Update $\mathbf{w_{p_i}} \leftarrow \mathbf{w_{p_i}} + \eta_2(\xi - \mathbf{w_{p_i}})$
26: **end for**
27: **end if**
28: Delete those edges in set E whose age outstrips the parameter $OldAge$.
29: **if** The number of learned examples is the integer multiple of parameter λ. **then**
30: Delete the nodes p_i in set G that have no neighbor node and delete the nodes p_i who has only one neighbor.
31: **end if**
32: **if** All the data in the block TS has been processed. **then**
33: Combine the prototype set G and the support vector set SV in new set $Temp$: $Temp = G \bigcup SV$.
34: Train a new SVM with the set $Temp$, get the new support vector set SVp_{new}.
35: Update the old support vector set SV. Delete the old support vectors in SV and add the support vectors in SVp_{new} to SV, i.e., SV=$\emptyset$, $SV = SVp_{new}$.
36: Go to Step(3) to continue the learning process.
37: **else**
38: Go to Step(4) to continue the online learning process.
39: **end if**
40: **if** All the training data has been processed. **then**
41: **return** set SV.
42: **end if**

Figura 5.2.: *Algoritmo de OISVM, extraído de (Jun Zheng 2010).*

El algoritmo propuesto para *OISVM* (**Figura 5.2.**), se divide conceptualmente en dos partes. Una parte, denominada *LPs* (Learning Prototypes), encargada de generar y agrupar los prototipos. Y la otra, denominada *LSVs* (Learning Support Vectors), encargada de gestionar las máquinas de soporte vectorial.

Como se comenta en el estado del arte, la parte *LPs* contiene el algoritmo de *ILVQ* para el agrupamiento de prototipos. Los únicos cambios de la parte de *LPs* respecto a ILVQ son que recibe distintos bloques de datos (se usa un buffer) y que

debe interactuar con *LSVs*. Vemos como las líneas del algoritmo propuesto para OISVM, desde la 2 hasta la 38, si ponemos como falsa la condición de la línea 32, corresponden al algoritmo de *ILVQ* estudiado en (S. F. Ye Xu 2009) (F. S. Ye Xu 2010). El resto de líneas, incluida la condición de la línea 32, pertenecen a la parte de *LSVs*.

El razonamiento que sigue OISVM puede resumirse en lo siguiente:

1. Recibe un bloque de datos procedente del stream.
2. Se genera el conjunto G de prototipos
3. Se unen los conjuntos agrupados con *ILVQ* (G) a los vectores de soporte (SV) y se envían a clasificar a una máquina de soporte vectorial (en caso de ser la primera iteración, no habrá SVs, por lo que sólo se envían los conjuntos agrupados).
4. Se extraen los SVs resultantes de la clasificación para la siguiente iteración.
5. Se vuelve a esperar hasta que se recibe del datastream un nuevo bloque de datos (paso 1).

ii. Diseño e implementación del clasificador

Tras el estudio y la interpretación del algoritmo en el apartado anterior, en este punto se expone la fase de codificación del clasificador. Para ello, se enumeran los cinco pasos resumidos del funcionamiento de *OISVM* y se explica para cada uno las decisiones tomadas referentes al diseño y la implementación.

1. *MOA* se encarga de clasificar datos de forma incremental, por lo que envía un prototipo por vez. Para realizar el stream de datos, creamos un buffer que almacena el número de prototipos fijado por el usuario. Cuando el buffer llega al número indicado, se cumple la condición para el siguiente paso. Para la implementación de esta parte se utilizó como plantilla la clase *WEKAClassifier*, de la librería interna de *MOA*, ya que esta implementa un buffer de instancias para llamar a un clasificador de *WEKA*. En nuestro caso se llama primero a *ILVQ*, pero es necesario dicho buffer para marcar el tamaño del subconjunto de entrada. Además los conjuntos resultantes serán enviados a *LibSVM*, la cual es una implementación muy utilizada de una máquina de soporte vectorial que dispone de interfaz para *WEKA* y se puede utilizar como clasificador batch.
2. Para la programación de *ILVQ* se ha utilizado un conjunto de tres clases proporcionado por el tutor. Se llegó a la conclusión de utilizar una implementación de otro alumno para evitar varias codificaciones paralelas del mismo. Por ello, en el clasificador implementado se llama a las clases de *ILVQ* proporcionadas.

3. Puesto que el algoritmo *ILVQ* proporcionado devolvía los resultados de forma distinta (objeto prototipo) a la entrada de *LibSVM*, hubo que añadir antes de la parte *LSVs*, varios bucles que recorrieran los resultados y los almacenaran en una lista (arraylist de prototipos).

4. El clasificador SVM se inicializa en cada iteración para no guardar los datos de las anteriores clasificaciones. Una vez inicializado, se envía el objeto Instances creado por medio del método público *buildClassifier*. La entrada de *LibSVM* es la unión de la lista G resultante de ILVQ y la lista de SVs devuelta en la iteración anterior del clasificador en forma de objeto Instances. Para dicha unión, se crean las instancias respectivas a cada elemento de las listas, y se van insertando, una a una, a un nuevo objeto Instances. El objeto Instances es una colección de objetos Instance, cada uno de los cuales representa un patrón de entrenamiento para la SVM (atributos + clase).

5. Para obtener los vectores de soporte SVs resultantes de la forma más sencilla, se ha puesto la variable de LibSVM *m_Model* como publica. De este modo, se accede a ella desde la clase de *OISVM*. Luego, se llama al método *getField*, copiado de LibSVM, que obtiene la matriz de SVs de las clases internas de LibSVM, así como el vector con los valores de sus etiquetas. Una vez obtenido todo, se copia a una lista (arraylist) de cara a la siguiente iteración.

6. Se da por finalizada la iteración del algoritmo y el buffer de entrada se encuentra vacío, así como todas las variables globales en su estado inicial. Al recibir nuevas instancias por parte de *MOA*, se procederá a su almacenamiento (paso 1).

Como último aspecto del diseño, queda justificar las decisiones tomadas respecto a las variables de entrada al algoritmo. Estas son parámetros cuyo valor debe fijarse por el usuario a la hora de la ejecución.

Las cuatro variables de entrada de OISVM son las siguientes:

- `FloatOption lambdaOption`: Su valor se otorga mediante el comando `-l`. Representa el símbolo λ descrito en (Jun Zheng 2010), cuyo valor es divisor del número de conjuntos de entrenamiento en *ILVQ* (S. F. Ye Xu 2009) y debe ser fijado por el usuario.
- `IntOption maxAgeOption`: Su valor se otorga mediante el comando `-a`. Representa la edad máxima de las conexiones de *E*.
- `IntOption tsLength`: Su valor se otorga mediante el comando `-t`. Representa longitud de los bloques a procesar (ver primera línea del algoritmo en apartado anterior). En otras palabras, representa la longitud del buffer de Instance's (que está representado como objeto Instances).
- `WEKAClassOption baseLearnerOption`: Su valor se otorga mediante el comando `-b`. Representa el tipo de clasificador de *WEKA* utilizado. Gracias a este parámetro se pueden enviar parámetros para el uso de *LibSVM*.

iii. Parámetros usados

Para validar el algoritmo de *OISVM* implementado, se ha buscado la comparación resultados con la literatura. Para ello, se ha probado con varios sets de datos de (Jun Zheng 2010).

Selección de los parámetros gamma y coste:

Antes de la validación de *OISVM* con los sets *optdigits, Usps y Shuttle*, puesto que el artículo de referencia no provee los valores de gamma y coste exactos de pruebas, ha habido que buscar estos. En la literatura sólo consta el uso de un kernel Gaussiano, con C y G de valor 2^i y 2^j , donde i y j tienen valores en el rango [–5, 5].

Algoritmos	Resultados de la literatura con LibSVM	Resultados propios con LibSVM
Optdigits	98.37%	98.44%
Usps	99.53%	99.82%
Shuttle	99.83%	98.66%

Tabla 5.2.1: *Valores de precisión alcanzada en la literatura y en este proyecto, con los parámetros C y G indicados, en los tres algoritmos, sobre LibSVM.*

Se ha utilizado el meta-clasificador *CVParameterSelection* (ver en apartado 3), con el fin de automatizar y optimizar el tiempo de las pruebas con un conjunto determinado de parámetros. Se han buscado parámetros con coste de 0 a 10, y de Gamma de 0 a 0.5 en determinado número de pasos. A más grande el conjunto de datos, menos pasos (para evitar tiempo de procesamiento). A continuación, se ofrecen dos tablas de resultados que ofrecen los valores hallados, con que *LibSVM* más se acerca a la precisión encontrada en la literatura.

Algoritmos	Gamma	Coste
Optdigits	0.0008	8
Usps	0.013526315789473685	4
Shuttle	0.04999999999999998	10

Tabla 5.2.2: *Valores de Gamma y Coste utilizado en la fase de validación del proyecto sobre LibSVM de kernel radial.*

iv. Pruebas de validación de OISVM

Prueba de validación mediante Optdigits:

El primer set probado ha sido *Optdigits* (3823 instancias en entrenamiento y 1797 de test). Los valores mostrados en la literatura y los probados son los siguientes: λ = 450, maxAge = 450, gamma = 0.0008, coste = 8, N_TS (número de bloques de datos TS) = 10. También se ha probado otro número de bloques N_TS.

Algoritmo LibSVM de WEKA	Precisión	Número de SV's
Conjunto optdigits	98.44%	1108

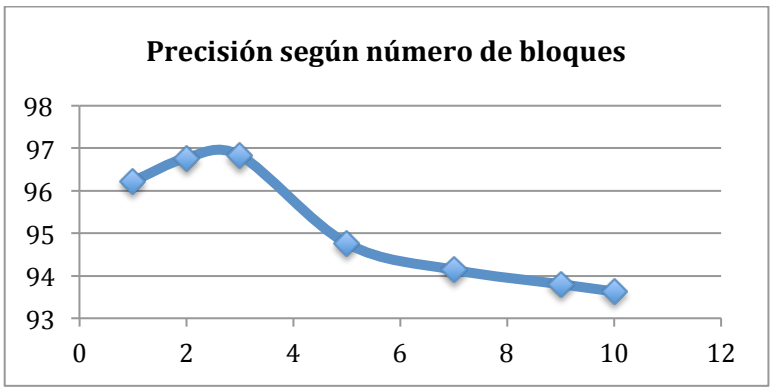
Tabla 5.2.3: *Precisión y número de SV's de optdigits en LibSVM con los parámetros utilizados.*

En primer lugar, se mira la correspondencia de las pruebas del conjunto en la ejecución en el algoritmo *LibSVM* de *WEKA*, y los resultados de la literatura pertinente (**Tabla 5.2.3**). Acto seguido, en la **Tabla 5.2.4** se expone una tabla de pruebas de optdigits sobre *OISVM*. Por último se ofrecen gráficas que ilustran el resultado de las tablas, para un análisis más ilustrativo.

N_TS	Precisión conseguida (%)	Número de SV's
1 (modo batch)	96,22	551
2	96,77	1176
3	96,83	1149
5	94,76	1685
7	94,14	1568
9	93,80	2099
10	93,63	2172

Tabla 5.2.4: *Precisión y número de SV's según el N_TS en la validación mediante el test y train set de Optdigits.*

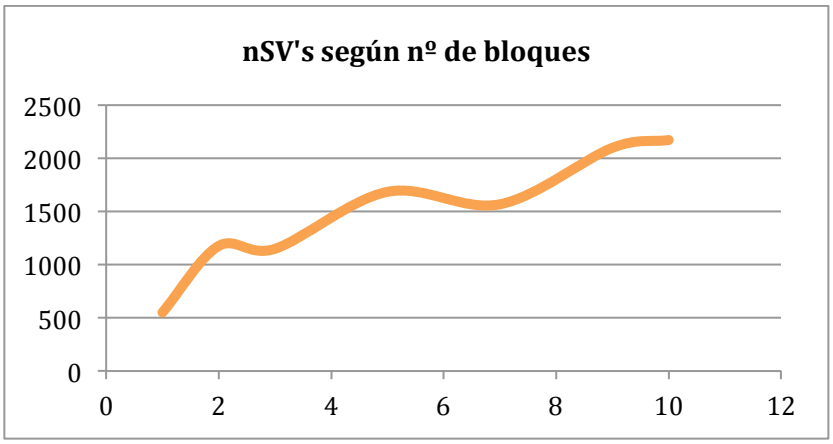
A partir de la **gráfica 5.2.1** se observa un decrecimiento de la precisión alcanzada al aumentar |*TS*| (tamaño de los bloques) . En primera instancia, el decrecimiento de la precisión no esperado respecto a las pruebas de los autores de *OISVM* puede deberse a que las pruebas de la literatura se realizan con 10 bloques de longitud arbitraria.



Gráfica 5.2.1: *Precisión según el N_TS en Optdigits.*

En la **gráfica 5.2.2** se observa el crecimiento casi lineal del número de *SV's* con el aumento de las iteraciones (particiones de *Optdigits*). Este número crece mucho respecto a las pruebas mostradas en (Jun Zheng 2010), cosa que podría tener algún efecto perjudicial en la precisión alcanzada.

Por otro lado, también se reitera la importancia de la longitud de los bloques. Bloques más grandes en ciertas zonas del conjunto podrían hacer que el clasificador fuera más resistente al ruido, mientras que bloques más pequeños en zonas carentes de ruido mejorarían la precisión y reducirían el número de clases, y con ello de *SV's*.



Gráfica 5.2.2: *nSV's según el N_TS en Optdigits.*

La validación del algoritmo mediante este conjunto de datos, se basa en la similaridad de la precisión y *nSV's* alcanzados en modo batch, con los de (Jun Zheng 2010), así como en la buena estabilidad alcanzada aumentando el número de iteraciones. Puesto a que el modelo se degrada más de lo esperado, convendrá hacer pruebas de validación con más conjuntos, a modo de buscar el patrón de distinción entre las pruebas de este proyecto y las de la literatura pertinente.

Prueba de validación mediante Usps:

Otro set de la literatura utilizado en la validación ha sido *Usps,* cuyo set inicial de 9297 instancias ha sido partido en dos conjuntos de longitud similar entre ellos. Los valores mostrados en la literatura, y los probados son los siguientes: λ = 500, maxAge = 400, gamma = 0.0135 y coste = 4 (*los dos últimos hallados mediante CVParameterSelection*).

Algoritmo LibSVM de WEKA	Precisión	Número de SV's
Conjunto Usps	99.83%	2091

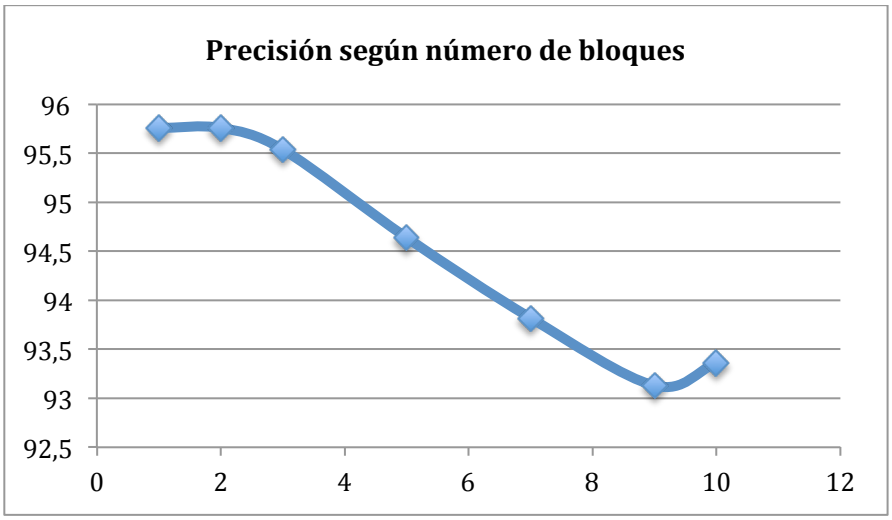
Tabla 5.2.3: *Precisión y número de SV's de Usps en LibSVM con los parámetros utilizados.*

En primer lugar, se mira la correspondencia de las pruebas del conjunto en la ejecución en el algoritmo *LibSVM* de *WEKA*, y los resultados de la literatura pertinente (**Tabla 5.2.3**). A continuación se expone una tabla de pruebas de *Usps* sobre OISVM (**Tabla 5.2.4)**.

N_TS	Precisión conseguida (%)	Número de SV's
1 (modo batch)	95,76	692
2	95,76	704
3	95,54	1169
5	94,64	1438
7	93,82	1566
9	93,13	1566
10	93,36	1718

Tabla 5.2.4: *Precisión y número de SV's según el N_TS en la validación mediante el train set de Usps con valores de C y Gamma óptimos.*

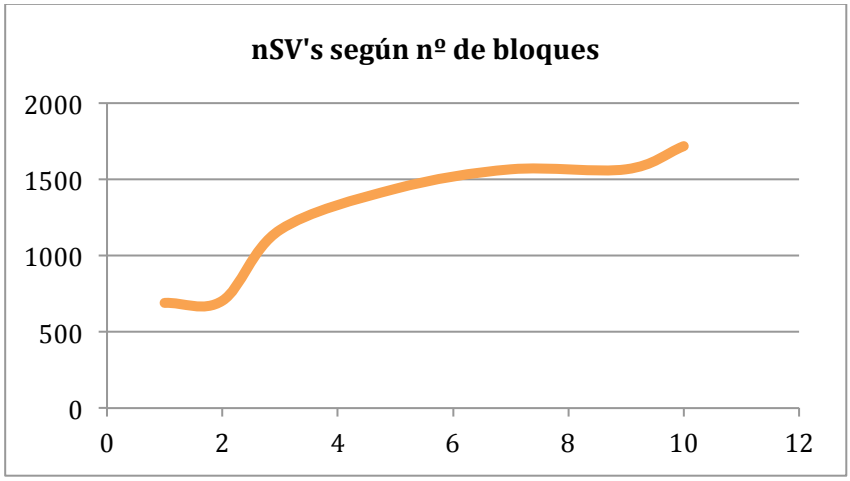
A partir de la **gráfica 5.2.3** se observa un decrecimiento de la precisión alcanzada al reducir el N_TS en los que se divide el conjunto. Al igual que sucede con *Optdigits*, el decrecimiento de la precisión puede explicarse debido a que las pruebas de la literatura se realizan con 10 bloques de datos de longitud arbitraria y que el resultado ofrecido en el documento de referencia debe de ser la media mediante la ejecución de dichos conjuntos por validación cruzada (como ya se ha comentado previamente).



Gráfica 5.2.3: *Precisión según el número de bloques de datos de tamaño* |TS| *en Usps.*

En la **gráfica 5.2.4** se observa el crecimiento del número de *SV's* con el aumento de las iteraciones, al igual que sucedía en el conjunto de validación anterior.

Puesto a que este modelo también se degrada más de lo esperado, se utilizará un último conjunto para la validación del algoritmo.



Gráfica 5.2.4: *nSV's según el N_TS en Usps.*

Prueba de validación mediante Shuttle:

Shuttle es el último conjunto, usado también en (Jun Zheng 2010), que utilizamos para validar *OISVM*. Los valores mostrados en la literatura, y los probados son los siguientes: λ = 15000, maxAge = 400, gamma = 0.04999999999999998 , coste = 10 (*los dos últimos hallados mediante CVParameterSelection*).

Algoritmo *LibSVM* de *WEKA*	Precisión	Número de *SV's*
Conjunto Shuttle	98.38	9435

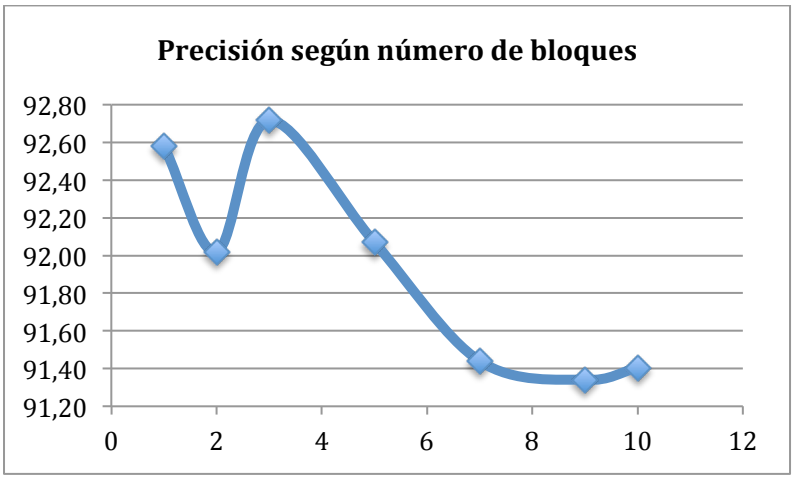
Tabla 5.2.5: *Precisión y número de SV's de Shuttle en LibSVM con los parámetros utilizados.*

Como en las anteriores pruebas de validación, primero se mira la correspondencia de las pruebas del conjunto en la ejecución en el algoritmo *LibSVM* de *WEKA*, y los resultados de la literatura pertinente (**Tabla 5.2.5**). Observamos una precisión cercana, aunque no similar a la mencionada en la literatura. Además, se obtienen 9435 *SVs* cuando la literatura obtiene 2112 *SVs* para dicha prueba. Se expone a continuación una tabla de pruebas de *Shuttle* sobre *OISVM* (**Tabla 5.2.6**).

N_TS	Precisión conseguida (%)	Número de SV's
1 (modo batch)	92,579	270
2	92,0197	651
3	92,71695	825
5	92,07	1646
7	91,44	1914
9	91,34	2530
10	91,4049	2753

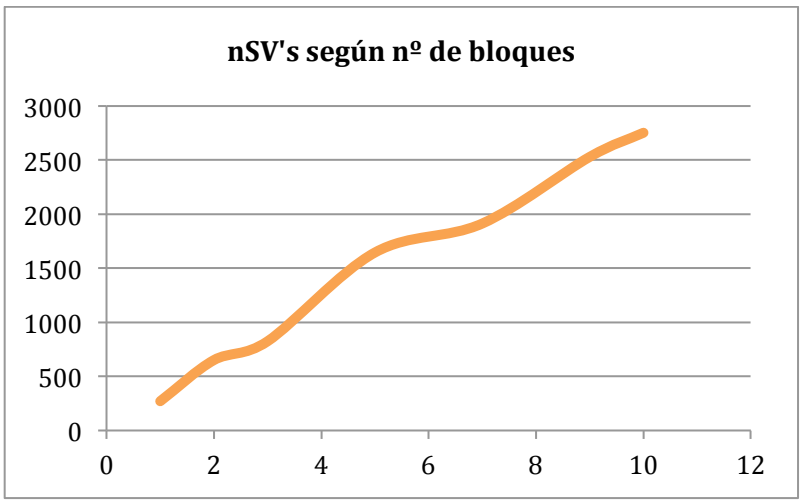
Tabla 5.2.6: *Precisión y número de SV's según el N_TS en la validación mediante el train set de Shuttle con valores de C y Gamma óptimos.*

Los resultados de la **tabla 5.2.6** se muestran representados en las siguientes gráficas, donde se observa un comportamiento similar al de los otros conjuntos de datos.



Gráfica 5.2.5. *Precisión según el N_TS en Shuttle.*

A partir de la **gráfica 5.2.5** se observa un decrecimiento de la precisión alcanzada al aumentar el tamaño |TS| al igual que en las anteriores dos pruebas de validación. La **gráfica 5.2.6** también se comporta como las dos pruebas anteriores. Se observa un crecimiento del número de *SV's* con el aumento de las iteraciones.



Gráfica 5.2.6: *nSV's según el N_TS en Shuttle.*

v. Conclusiones de validación de OISVM:

Un resumen comparativo entre los resultados obtenidos y los originales se muestra en la siguiente tabla:

	LIBSVM		OISVM (Batch)		OISVM (10 blocks)	
	Paper	Nuestros	Paper	Nuestros	Paper	Nuestros
Optdigits	98.37	98.44	97.33	96.22	97.33	93.63
Shuttle	99.83	98.38	99.68	92.57	99.77	91.40
Usps	99.53	99.83	98.96	95.76	99.04	93.36

Tabla 5.2.7: *Resumen de resultados de validación obtenidos y esperados con los distintos conjuntos.*

Las pruebas realizadas no se ajustan con exactitud a los resultados alcanzados en (Jun Zheng 2010). No obstante, como se ha comentado en el primer apartado de esta sección, nuestras pruebas de validación no se han realizado de forma exacta a la literatura. En parte por que en el artículo de referencia no habla con exactitud del procesamiento de los conjuntos de datos, ni del tipo de pruebas realizadas en el estudio, y también por que no estamos verificando que las pruebas de la literatura son correctas, sino validando la correcta programación e integración del algoritmo en *MOA*. Por ello, valores parecidos serán aceptados para la validación.

En cuanto a las gráficas mostradas, cada prueba sólo se ha procesado una vez. Si hubiéramos calculado la media de diez distintas pruebas (como en muchos artículos de la literatura), estas serían mucho más lineales.

Por otro lado, los resultados pueden haber variado respecto a los de la literatura debido a la existencia de ruido que haya sido eliminado en las pruebas de los artículos sin previo aviso; ya que en estas no se detalla el procesamiento hecho sobre los datos. Así mismo, se han usado los parámetros de *LibSVM* proporcionados en la literatura, que pueden no ser los óptimos si el tratamiento seguido no este.

Ya que los valores alcanzados que pueden considerarse cercanos a los presentados (Jun Zheng 2010), consideramos por las tres pruebas realizadas, un buen funcionamiento y por ende, una correcta programación del algoritmo.

vi. Problemas encontrados

Una vez comprendido el algoritmo *OISVM* (Jun Zheng 2010) y el funcionamiento de *LibSVM*, así como asimilado el conjunto de clases utilizado en los clasificadores de *MOA*, la implementación de *Online Incremental SVM* para *MOA* ha sido prácticamente directa. No obstante, algunos problemas e inconvenientes surgieron a la hora de interpretar el clasificador incremental. En primer lugar, varios parámetros del algoritmo propuesto no se encuentran explicados en profundidad, por lo que su interpretación podría ser confusa si no se ha leído otros artículos de los mismos autores (S. F. Ye Xu 2009) (F. S. Ye Xu 2010). A partir del estudio de estos, se descubre la utilización de *ILVQ* en el proceso de clustering de *LPs*, que facilitó la tarea de programación al reutilizar código existente.

En la fase de programación, surgieron dos problemas principales:

- En primer lugar, se necesitaba enviar los datos a *LibSVM* por medio de un objeto Instances. Para ello, debían definirse objetos Instance para cada instancia representada, en las clases de *ILVQ* proporcionadas a modo de objeto 'Prototipo'. El problema aquí es que no existe un constructor de Instance, por lo que la solución tomada finalmente fue la clonación del último objeto Instance metido en el buffer (aún en memoria intermedia). Una vez

clonado en otro objeto, se vacía su contenido y se llena con el del objeto prototipo correspondiente. De cara a clasificar con *LibSVM*, se añaden los objetos Instance, de uno en uno, al objeto Instances sin ningún tipo de inconveniente.

- El otro problema encontrado fue la recuperación de los vectores de soporte y de sus etiquetas de las clases internas de *LibSVM*. Para su recuperación, se necesitó la copia del método de *LibSVM getField(x,y),* que obtiene el valor del parámetro de nombre *y*, del objeto *x*. Gracias al método, se obtienen los campos *nSV* y *label* del objeto *m_Model* de *LibSVM*. El primer campo es una matriz de dos dimensiones que contiene la lista de vectores de soportes resultantes. El segundo, es un vector de tanta longitud como clases distintas existan, que contiene el número de prototipos para ese valor de etiqueta. La asignación de las etiquetas con los SV es trivial, pues se encuentran ordenados por valor de clase. Por ejemplo, si los valores de *label* son {3,4,2}, los tres primeros *SVs* tendrán etiqueta 0, los cuatro siguientes etiqueta 1, y los dos últimos etiqueta 3.

Otro inconveniente fue a la hora de encontrar los parámetros *C* y *Gamma* óptimos. Como no venían explícitos en la literatura, hubo que obtenerlas haciendo pruebas de forma iterativa, algo que consumió mucho tiempo.

Una vez solucionados estos problemas, a parte de algún error común de compilación, no hubo más inconvenientes que retrasaran ninguna fase referente a la implementación.

Sin embargo, la fase de pruebas surgió un inconveniente al usar evaluadores por defecto de *MOA*, cuyas instancias se pasaban de una en una, y en cada paso se ejecutaba la clasificación. Esto hacía que antes de terminar de cargar en buffer el primer conjunto de datos *TS*, se ejecutara la clasificación por defecto de *WEKAClassifier*. Además, las instancias que clasificaran tras cargar un bloque TS, lo harían con un bloque de retraso.

Por ejemplo, si cada bloque fuera de 100 patrones, los 100 primeros pasados para clasificar lo harían sin la máquina SVM entrenada, los siguientes 100 lo harían con la SVM con que se deberían haber clasificado los 100 anteriores (por motivos de '*concept drift*' si los conjuntos de datos fuesen no estacionarios explicados en el estado del arte, aunque este no sea el caso), los 100 siguientes con la de los 100 anteriores, y así sucesivamente. Esto tenía una pésima repercusión en los resultados de precisión de las pruebas. Tras detectar el error, hubo que cambiar los evaluadores utilizados por el programado por otro compañero (y mencionado en el apartado anterior) *EvaluateChunksTwoFiles*.

Un último inconveniente que surgió las fases finales de este proyecto (Estudio Comparativo), e hizo tener que volver a repetir toda la validación realizada para *OISVM* y para *IGNGSVM* (ya que su estructura incremental es similar al del

primero), fue un problema con las copias auxiliares para las instancias entrantes. Al crear nuevos objetos Instances (y también al definir nuevas SVMs) no se limpiaba correctamente su contenido y quedaban ejemplos de entrenamiento del conjunto de datos. Esto hacía que se mantuviese una copia del conjunto total siempre en memoria, cosa que hacía cambiar los resultados y ralentizaba el algoritmo. Al percatarnos de cierto funcionamiento sospechoso con algunos evaluadores similares a los de la literatura de cada algoritmo (usados en la primera fase del estudio comparativo), detectamos dicho fallo con trazas en los algoritmos y se solucionó limpiando correctamente estas copias auxiliares de los objetos Instances y *LibSVM*.

5.3. GROWING NEURAL GAS

La implementación del algoritmo propuesto, *IGNGSVM*, supone la previa integración en *MOA* del algoritmo incremental Growing Neural Gas (Fritzke, 1995).

En este apartado, se analiza el algoritmo propuesto por el autor, se interpreta su funcionamiento, se explican las decisiones tomadas en la fase de codificación; los parámetros usados en la validación; y los problemas encontrados.

i. Análisis del algoritmo

En el **apartado 2.4.4** se expone el algoritmo propuesto en la literatura para *GNG*. Este algoritmo de clustering basa su funcionamiento en la reducción de los datos de entrada a su conjunto topológico. Es decir, se asociarán unas neuronas con otras según su distancia euclídea, y sólo perdurarán aquellas conexiones que a lo largo del tiempo se consideren más importantes en el conjunto de datos.

El algoritmo a programar para Growing Neural Gas se resume en lo siguiente:

1. Comienza al haber recibido al menos dos prototipos de entrada, sin importar su etiqueta.
2. Se encuentra las unidad más cercana s_1 y la segunda unidad más cercana s_2. Tras ello, se incrementa la edad en las conexiones de s_1.
3. Se actualiza el valor del error acumulado de s_1 según la distancia euclídea con la última instancia entrante ξ.
4. Se mueve s_1 y sus vecinos topológicos hacia ξ en múltiplos de los dos parámetros de entrada $\in_b$ y $\in_n$, respectivamente.
5. Se reinicializa la conexión entre s_1 y s_2.
6. Se eliminan las conexiones de edad superior a a_{max}, y las neuronas que se queden sin conexiones.
7. Paso de interpolación. Se crea una nueva neurona r en el centro del punto con más error y su vecino con más error. Se crean nuevas conexiones, y se actualizan las antiguas y los valores de error acumulado.
8. Se reducen todas las variables de error multiplicándolas por la constante *d*.

ii. Diseño e implementación del algoritmo de clustering

Tras el estudio y la interpretación del algoritmo en el apartado anterior, en este punto se expone la fase de codificación. Para ello, se enumeran los pasos resumidos del funcionamiento de *GNG* y se explica para cada uno las decisiones tomadas referentes al diseño y la implementación.

1. Al ser un algoritmo incremental, recibe las instancias de una en una. En cuanto recibe las dos primeras (sin tener en cuenta su etiqueta), estas son detectadas como las unidades de entrada *a* y *b,* de vectores de referencia w_a y w_b. Se interpreta la instancia entrante en la llamada al método de entrenamiento actual como dato de entrada ξ, donde la función $P(\xi)$ será el conjunto de datos de entrenamiento del problema.

2. Se encuentra las unidad más cercana s_1 y la segunda unidad más cercana s_2 y se incrementa la edad en las conexiones de s_1. Puesto que el código se reutiliza de *ILVQ*, el cálculo de la distancia euclídea se realiza con una llamada al método '*obtenerCercanos*'. La edad de las conexiones que tenga el objeto prototipo con vector de referencia más cercano se ponen a cero. También se crea una conexión, con el método de *ILVQ* '*anadirVecino*', o se pone a cero su edad si esta ya existe.

3. Se actualiza el valor del error acumulado de s_1 según la distancia euclídea con la última instancia entrante ξ. Para ello, se define una nueva variable global sobre la clase de prototipos. Se llamará al método '*setError*' para cambiar dicha variable en el valor calculado con el método de prototipo '*dist*'.

4. Para mover s_1 y sus vecinos topológicos hacia ξ en múltiplos de los dos parámetros de entrada $\in_b$ y $\in_n$, respectivamente, se han modificado las funciones realizadas en el método de Prototipo "*actualizarW*", donde antes se actualizaba la posición siguiendo otra ecuación. A su vez, ahora el valor enviado es el parámetro de entrada $\in_b$ ó $\in_n$, según proceda.

5. Se reinicializa la conexión entre s_1 y s_2 utilizando el método 'buscarEnlace'. Si existe, se pone su edad a cero; si no, se llama a '*anadirVecino*'.

6. Se eliminan las conexiones de edad superior a a_{max}, llamando al método de Prototipo '*purgarConexiones*'. Tras ello, se recorre la lista S y se eliminan de esta los puntos que no tengan vecinos.

7. Si la iteración de *GNG* es múltiplo de lambda, se crea una nueva neurona r en el centro del punto q con más error, buscado iterativamente, y su vecino con más error (consultando iterativamente todos los vecinos de q). Se crean nuevas conexiones con '*anadirVecino*', y se elimina la conexión entre q y su vecina con más error con '*eliminarVecino*', y se actualizan los valores de error acumulad con '*setError*'.

8. Se reducen todas las variables de error multiplicándolas por el parámetro de entrada '*constantOption*' mediante '*setError*'.

Como último aspecto del diseño, queda justificar las decisiones tomadas respecto a las variables de entrada al algoritmo. Estas son parámetros cuyo valor debe fijarse por el usuario a la hora de la ejecución.

Las variables de entrada de GNG son las siguientes:

- `IntOption lambdaOption`: Su valor se otorga mediante el comando –l. Representa al símbolo λ, cuyo valor es divisor del número de conjuntos de entrenamiento.
- `IntOption maxAgeOption`: Su valor se otorga mediante el comando –m. Representa la edad máxima que pueda alcanzar cualquier conexión de una neurona.
- `FloatOption alfaOption`: Su valor se otorga mediante el comando –a. Representa el símbolo alfa, el parámetro que hace decrecer los valores de error acumulado de las neuronas q y f en el caso de interpolación.
- `FloatOption constantOption`: Su valor se otorga mediante el comando –d. Representa el símbolo d, encargado del decrecimiento de todas los valores de error acumulado de la red.
- `FloatOption BepsilonOption`: Su valor se otorga mediante el comando –e. Representa el parámetro $epsilon_b$, que mueve a la neurona más cercana s1.
- `FloatOption NepsilonOption`: Su valor se otorga mediante el comando –n. Representa el parámetro $epsilon_n$, que mueve a los vecinos de s1.

iii. Parámetros usados

Como la programación del algoritmo de agrupación Growing Neural Gas ha sido orientada exclusivamente a su uso en *IGNGSVM*, para su validación, se observará el funcionamiento del clustering en un algoritmo representable en dos dimensiones.

Para ello, se ha realizado una prueba sobre el conjunto de ejemplos stbal1 de 3000 ejemplos de entrenamiento, con 6 bloques de tamaño |TS| = 500. Otros parámetros usados, definidos por las iniciales de entrada al clasificador *IGNGSVM*, se muestran en la **Tabla 5.3.1**.

$\boldsymbol{\lambda}$	16
Edad máxima de conexión (a_{max})	16
$\boldsymbol{\alpha}$	0.5
$\boldsymbol{d}$	0.995
$\boldsymbol{\varepsilon_b}$	0.2
$\boldsymbol{\varepsilon_n}$	0.006
Criterio de parada (neuronas generadas por interpolación para pasar a reducir el siguiente subconjunto S)	12

Tabla 5.3.1. *Parámetros escogidos para la validación de GNG, sobre IGNGSVM, en stbal1*

iv. Prueba de validación de GNG y conclusiones

En la **Figura 5.3.1** se observa una distribución equivalente a un bloque *TS* de los seis en los que se divide el conjunto al aplicar un tamaño de 500 instancias. En dicha iteración, observamos que la clase 0 tiene un total de 255 instancias, y que la clase 1 tiene 245.

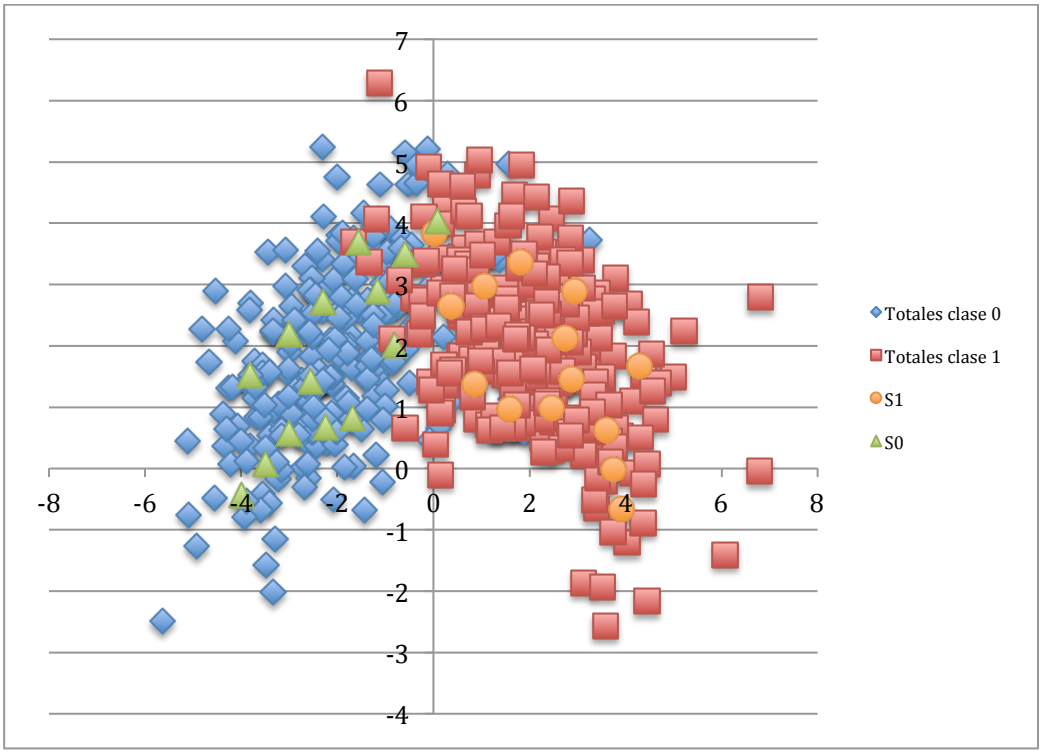


Figura 5.3.1: *Representación gráfica de la reducción realizada en el último bloque TS de la prueba realizada de IGNGSVM sobre stbal1.*

Se observa también como la reducción que se produce al tener un criterio de parada de doce puntos creados en fase de interpolación, es de catorce instancias para cada subconjunto S de S* (es decir, se pasa de 500 a 28 ejemplos de entrenamiento). En la **Figura 5.3.2** puede observarse con más facilidad la agrupación y distribución de los puntos que ha tenido lugar en la **Figura 5.3.1**.

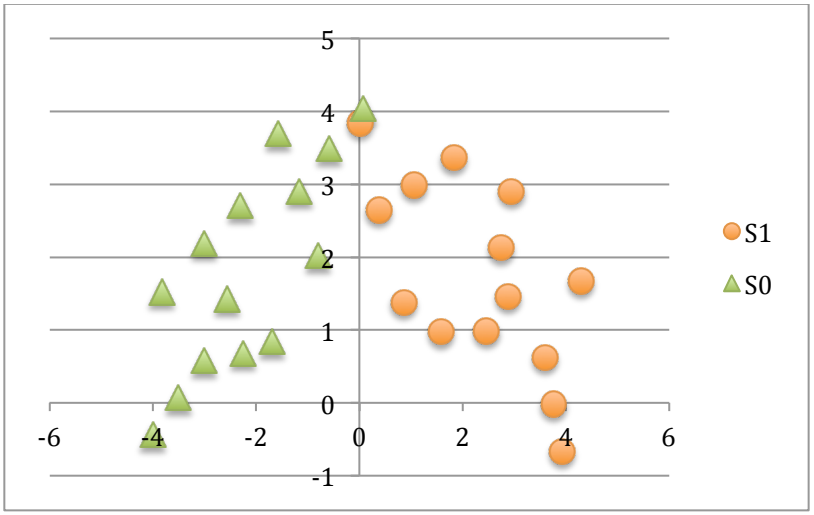


Figura 5.3.2: *Representación gráfica de los subconjuntos S0 y S1 obtenidos.*

Dado el nivel de generalización y representación alcanzado, podemos suponer válida la programación de *GNG*. Sin embargo, y también para asegurarnos de su correcto ensamblaje con SVM, podemos observar en la **Figura 5.3.3** el resultado de SVs totales representado sobre los subconjuntos S0 y S1 de la última iteración. Observamos como los SVs separan correctamente la distribución de datos estacionaria (cuya representación es aproximada en los seis bloques TS). Para dar mayor validez a la prueba realizada, el resultado alcanzado en precisión tras la reducción es del 91.68%, cuando el obtenido en un ciclo batch de *LibSVM*, sobre el conjunto completo y sin reducir para los mismos parámetros de SVM (gamma 0 y coste 8, con kernel radial) es del 92.37%. Esto es significativo, sobre todo si tenemos en cuenta la labor de reducción producida y su gran nivel de representación.

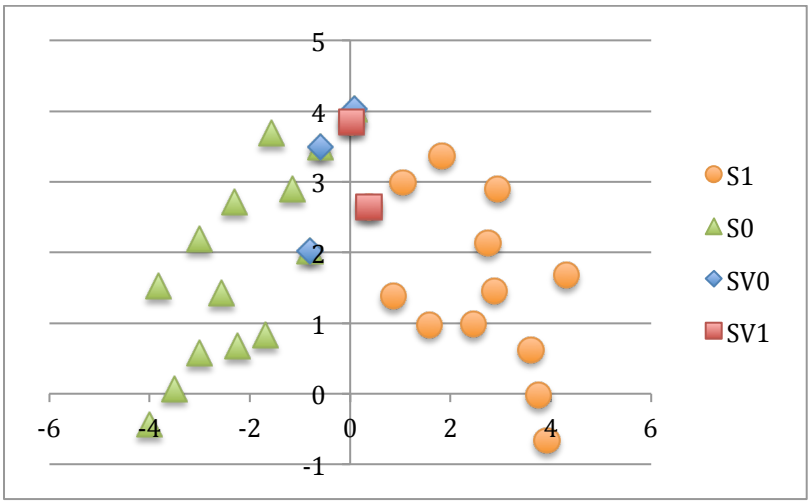


Figura 5.3.3: *Superposición de los SVs obtenidos al final del algoritmo, sobre los subconjuntos S0 y S1 obtenidos en la última iteración.*

iv. Problemas encontrados

Una vez comprendido el algoritmo incremental a integrar en *MOA*, la implementación de *GNG* ha sido directa reutilizando las clases existentes de *ILVQ*.

El principal problema ocurrió a la hora de interpretar la utilización de los datos que en un principio debían ser aleatorios, a y b, pero que por no poder determinar el espacio al que pertenecen R^n, deben pertenecer al espacio de entrada. Por consiguiente, serán las dos primeras instancias que entren en el algoritmo.

Una vez solucionados este y otros problemas ya en fase de implementación, no hubo más inconvenientes que retrasaran la parte de validación. Antes de proceder a esta, se volvió a comprobar mediante trazas el correcto funcionamiento del algoritmo.

5.4. INCREMENTAL GROWING NEURAL GAS - SUPPORT VECTOR MACHINES

Al igual que *OISVM*, el clasificador *GNGSVM* (Ondrej Linda, 2009) utiliza un algoritmo de cuantización vectorial para lograr la reducción de los datos a su subconjunto topológico y, con ello, facilitar el uso de grandes conjuntos de datos en máquinas de soporte vectorial. Como variante, en este trabajo se propone una versión incremental de dicho algoritmo, llamada *Incremental Growing Neural Gas - Support Vector Machine*.

En este apartado se analiza el algoritmo propuesto, se interpreta su funcionamiento, se explican las decisiones tomadas en la fase de codificación; los parámetros usados en la validación; y los problemas encontrados.

i. Análisis del algoritmo

En la **sección 2.4.5** del estado del arte, se expone el algoritmo propuesto para *IGNGSVM*. Este, al igual que *OISVM*, se divide conceptualmente en varias partes. En primer lugar, se hace utilización del algoritmo *GNG* para reducir los datos a su conjunto topológico (viene siendo la función de la parte *LPs* que hace *OISVM* mediante *ILVQ*). En segundo lugar, se etiquetan los distintos grupos topológicos y se entrenan en un conjunto en SVM. Por último, se realiza una modificación incremental sobre el algoritmo de (Ondrej Linda, 2009) para adaptarlo a un entorno incremental. Esta última parte copia la estructura de *LSVs* de *OISVM*.

El algoritmo a programar con *IGNGSVM* se resume en lo siguiente:

1. Recibe un conjunto de prototipos almacenado en un stream de datos.
2. Se divide los datos entrantes en sub-conjuntos por distintas clases.
3. Se obtiene la topología de cada subconjunto con *GNG*, de forma aislada.
4. Se unen todas las instancias que forman parte de cada una de las topologías de las distintas instancias de *GNG*.
5. Se unen las instancias de topologías (*S**) a los vectores de soporte (*SV*) y se envían a clasificar a una nueva máquina de soporte vectorial (en caso de ser la primera iteración, no habrá *SVs*, por lo que sólo se envían los conjuntos agrupados).
6. Se extraen los *SVs* resultantes de la clasificación para la siguiente iteración.
7. Se vuelve a esperar por un nuevo conjunto de prototipos (paso 1).

ii. Diseño e implementación del clasificador

Tras el estudio y la interpretación del algoritmo en el apartado anterior, en este punto se expone la fase de codificación del clasificador. Para ello, se enumeran los siete pasos resumidos del funcionamiento de *IGNGSVM* y se explica para cada uno las decisiones tomadas referentes al diseño y la implementación.

1. *MOA* es un entorno de clasificación incremental que envía un prototipo por vez. Para realizar el stream de datos, se crea un buffer que almacena el número de prototipos fijado por el usuario. Cuando el buffer llega al número indicado, se cumple la condición para el siguiente paso. Para la implementación de esta parte se utilizó como plantilla la clase *WEKAClassifier*, de la librería interna de *MOA*, ya que esta implementa un buffer de instancias para llamar a un clasificador de *WEKA*.
2. Puesto que *GNG* es un algoritmo de clustering, y no de clasificación, este extrae la topología sin tener en cuenta la clase si se lo permitimos; cosa que arruinaría el proceso de clasificación posterior con *SVM*. Por ello, en *GNGSVM* (y *IGNGSVM*) se divide el conjunto del buffer en varios subconjuntos por el valor de su clase (tantos subconjuntos como clases, y cada prototipo en el subconjunto que pertenece a su clase).
3. Para obtener la topología de cada subconjunto *GNGi*, se crea un objeto *GNG* por cada clase, y es entrenado dentro de un bucle, que extrae a una lista *S* los resultados de la topología. Al terminar el bucle, la lista de listas *S** contiene todos los vectores de referencia en un orden correspondiente a la clase a la que pertenecen (dicho orden está determinado por el vector 'label').
4. Para la programación de *GNG* se ha reutilizado el conjunto de clases proporcionado para *ILVQ.* Esta reutilización y variación del código original se hizo por la similitud de ambos algoritmos de agrupación, y por mantener un estándar de codificación dentro del proyecto.
5. Se utiliza *LibSVM* en vez de *SMO* como algoritmo de clasificación con SVMs, debido a la reutilización de parte del código programado para *OISVM*. Esto facilita la comparativa entre el último e *IGNGSVM*. La entrada de prototipos en *LibSVM* se realiza mediante el envío de un objeto Instances. Dicho objeto debe contener todos los conjuntos de prototipos y SVs que deban enviarse según el algoritmo de *IGNGSVM* en forma de objeto Instance (cada objeto Instance contiene todos los valores de cada instancia, así como el valor de su etiqueta).
6. Como *GNG* devuelve los resultados en forma de objeto prototipo, los *SVs* también son convertidos en objeto prototipo (al igual que en *OISVM*), de forma que la unión de ambos conjuntos para su entrenamiento con SVM sea más sencilla. Para dicha unión, se crean las instancias respectivas a cada elemento de las listas, y se van insertando, una a una, a un nuevo objeto *Instances*. En este paso se inicializa, en cada iteración, el clasificador SVM con

el objetivo de que no guarde los datos de las anteriores clasificaciones, de esta forma *IGNGSVM* podrá compararse directamente a *OISVM*, por seguir una estructura muy parecida.

7. Para obtener los vectores de soporte *SVs* resultantes de la forma más sencilla, se ha puesto la variable de *LibSVM m_Model* como publica. De este modo, se accede a ella desde la clase de OISVM. Luego, se llama al método *getField*, copiado de *LibSVM*, que obtiene la matriz de SVs de las clases internas de *LibSVM*, así como el vector con los valores de sus etiquetas. Una vez obtenido todo, se copia a una lista (arraylist) de cara a la siguiente iteración.
8. Se da por finalizada la iteración del algoritmo y el buffer de entrada se encuentra vacío, así como todas las variables globales en su estado inicial. Al recibir nuevas instancias por parte de *MOA*, se procederá a su almacenamiento (paso 1).

Como último aspecto del diseño, queda justificar las decisiones tomadas respecto a las variables de entrada al algoritmo. Estas son parámetros cuyo valor debe fijarse por el usuario a la hora de la ejecución.

Las variables de entrada que añade *IGNGSVM* sobre *GNG* son las mismas que *OISVM* sobre *ILVQ*. Estas son:

- `IntOption tsOption`: Su valor se otorga mediante el comando `–t`. Representa longitud de los bloques a procesar. En otras palabras, representa la longitud del buffer de Instance's (que está representado como objeto Instances).
- `WEKAClassOption baseLearnerOption`: Su valor se otorga mediante el comando –b. Representa el tipo de clasificador de *WEKA* utilizado. Gracias a este parámetro se pueden enviar parámetros para el uso de LibSVM.

El resto de variables son las que necesita *GNG* para su funcionamiento y que ya se encuentran descritas en el **apartado 5.3**: *lambdaOption, maxAgeOption, alfaOption, constantOption, BepsilonOption y NepsilonOption.*

iii. Parámetros usados

Para validar el algoritmo de *IGNGSVM* implementado, se ha buscado la comparación resultados con la literatura. Para ello, se ha probado con el set de datos *Shuttle* con un kernel polinómico, usando un solo bloque cuya longitud es la de todo el conjunto de entrenamiento (ya que la versión de *GNGSVM* de la literatura no está codificada para separar el conjunto de datos en varios bloques de tamaño |TS|).

Los valores mostrados en la literatura, y por tanto los probados, son los siguientes: λ = 100, a_{max} = 200 , ϵ_b= 0.2, ϵ_n= 0.006, α = 0.5, d = 0.995. El resumen de resultados se muestra en la **Tabla 5.4.1**.

	LIBSVM		GNGSVM (Batch)	
	Paper	Nuestros	Paper	Nuestros
Shuttle	96.57	97.55	91.34	93.66

Tabla 5.4.1: *Resultados de validación obtenidos y esperados.*

Selección de los parámetros gamma y coste:

Antes de la validación de *GNGSVM* con *Shuttle*, puesto que el artículo de referencia ejecutaba SMO (otro clasificador con SVM) con kernel polinómico y sus parámetros por defecto, hay que buscar los valores de gamma y coste de LibSVM.

Igual que en la validación de *OISVM,* se ha utilizado el meta-clasificador *CVParameterSelection* (ver en apartado 3), con el fin de automatizar y optimizar el tiempo de las pruebas con un conjunto determinado de parámetros. Se han buscado parámetros con coste de 0 a 10 en diez pasos, y de Gamma de 0 a 1 en doscientos pasos. A más grande el conjunto de datos, menos pasos (para evitar tiempo de procesamiento). **La tabla 5.4.2** muestra los valores de Coste y Gamma encontrados por *CVParameterSelection* que logran una mayor tasa de acierto.

Conjuntos	Gamma	Coste
Shuttle	0.8844798994974845	8

Tabla 5.4.2: *Valores de Gamma y Coste utilizado en la fase de validación del proyecto sobre LibSVM de kernel radial.*

iv. Prueba de validación de IGNGSVM

En primer lugar, se mira la correspondencia de las pruebas del conjunto en la ejecución en el algoritmo *LibSVM* de *WEKA*, y los resultados de la literatura pertinente (**Tabla 5.4.3**).

Algoritmo LibSVM de WEKA	Precisión (%)	Número de SV's
Conjunto Shuttle	97.55	5151

Tabla 5.4.3: *Precisión y SV's de Shuttle en LibSVM con los parámetros utilizados en kernel polinómico.*

Observamos una precisión mejor a la mencionada en la literatura. Esto es debido a que las pruebas se han hecho seleccionando distintos grupos de datos. Se expone a continuación una tabla de pruebas de *Shuttle* sobre *GNGSVM*.

Algoritmo IGNGSVM (batch)	Precisión (%)	Número de SV's
Conjunto shuttle	93.66	72

Tabla 5.4.4: *Precisión y número de SV's con |TS| = TAMAÑO DEL CONJUNTO en la validación mediante el train set de shuttle con kernel polinómico de grado 1.*

Se observa como el resultado es mejor que el mostrado en la literatura **(Tabla 5.4.4).** No obstante, en la literatura las pruebas realizadas han sido diez distintas con distintos subconjuntos del fichero, cuya media es la mostrada en los resultados (además de utilizar *SMO* con distintos parámetros, en lugar de *LibSVM*). Como ya se ha mencionado, puesto que sólo se desea validar la correcta programación del algoritmo propuesto, damos por sentada su correcta programación con esta prueba.

iv. Problemas encontrados

Una vez comprendido el algoritmo *GNGSVM*, la implementación de *IGNGSVM* ha sido directa reutilizando las clases existentes de *OISVM*. Quizá las variante que más trabajo costó implementar fue la separación de los bloques TS en *i* sub-conjuntos por clase. Concretamente en esta parte, los sub-bloques *TSi* con sólo una instancia entraban en un bucle infinito, ya que no existía error acumulado en el conjunto, y la red no llegaba a implementar nunca el número de neuronas que consta en el criterio de parada. Como solución a este problema, los sub-bloques con una sola instancia no entran en dicho bucle, si no son añadidos al conjunto S* directamente (no se reducen).

Una vez solucionados estos problemas de implementación en *IGNGSVM*, a parte de algún error común de compilación, no hubo más inconvenientes que retrasaran la fase de validación. Una vez en ella, se detectaron fallos de depuración en el algoritmo que no tardaron en corregirse.

6. ESTUDIO COMPARATIVO

Tras la correcta validación de los algoritmos programados, en este apartado se desarrolla el estudio comparativo entre *OISVM*, *IGNGSVM* y *SGD*. Éste último, pese a no haber sido implementado en este trabajo, nos ha parecido interesante añadirlo al análisis para evaluar su funcionamiento por ser un algoritmo de distinta naturaleza a los otros.

A continuación se exponen los modos de evaluación utilizados para la comparación de los algoritmos, los conjuntos de datos usados en las pruebas, y se describirá la forma de representar y comparar los resultados con la que se cierra la sección.

6.1. Modos de evaluación utilizados

Como se menciona brevemente en la fase de validación, en fase de estudio se desarrollan algunas pruebas con un mismo conjunto de entrenamiento y test. Este es el caso de las tareas de evaluación '*EvaluateInterleavedTestThenTrain*', '*EvaluateInterleavedChunks*' y *GenerateModel.* Para estos, se ignora el resultado obtenido en el primer bloque de tamaño |TS|, en el cual la máquina SVM aún no está entrenada.

El primer bloque de datos se ignora mediante el uso del modo de evaluación '*WindowClassificationPerformanceEvaluator'*, asignando un tamaño de ventana igual al tamaño sumado de todos los bloques menos el primero. *SEA* es evaluada con dos conjuntos (test y train) mediante '*EvaluateChunkTwoFiles*'. Finalmente, en la segunda fase de estudio los cuatro conjuntos son evaluados con '*EvaluateChunkTwoFiles*' para *OISVM* y *IGNGSVM*, de cara a fijar un modo de evaluación único y hacer los conjuntos lo más comparable posibles para el análisis de resultados.

i. EvaluateInterleavedTestThenTrain

Este método de evaluación se encuentra descrito en la sección 3 de este trabajo. Permite que se pueda usar individualmente cada ejemplo para probar el modelo antes de ser usado en el entrenamiento. El problema que tiene esta tarea de evaluación es que ofrece resultados antes de que se entrene la primera máquina SVM, por lo que, como ya se ha mencionado, en estas pruebas se utiliza un tamaño de ventana que haga ignorar el primer bloque, donde los resultados no son fieles al modelo. Utilizado en *Elec2*.

ii. EvaluateInterleavedChunks

La diferencia fundamental entre este método y el anterior es que aquí las instancias se envían en grupos de más de una (por ejemplo, en pruebas efectuadas con *Weather* se han enviado grupos de 300 en 300 instancias que se encargan de llenar un buffer de |TS| = 1815 [para N_TS = 10]). Como es lógico, en esta tarea se repite el método utilizado con el tamaño de ventana para ignorar el primer bloque.

iii. GenerateModel

El uso de este evaluador se presenta en la literatura estudiada para tratar con conjuntos de datos partidos en conjuntos de Train y test (ejecutando el fichero de test una vez entrenado el modelo completo). *GenerateModel* procesa las instancias de entrenamiento para generar un modelo. Luego, se evalúa todo el conjunto de evaluación con el modelo final. Se ha usado en primera fase de estudio para hallar los resultados de *Shuttle*.

iv. EvaluateChunksTwoFiles

Este evaluador se encarga de primero entrenar, y luego clasificar, utilizando un fichero de entrenamiento y otro de test. Como *EvaluateInterleavedChunks*, recibe los datos de entrenamiento y test por bloques; pero, a diferencia de este, genera primero un modelo con los patrones de entrenamiento, y luego evalúa dicho modelo con los patrones de evaluación. Evita el uso de tamaño de ventana, al asegurar entrenamiento antes de test. Es utilizado en los cuatro conjuntos en la segunda fase del estudio comparativo (**sección 6.5**).

Las pruebas bajo evaluación distinta a *EvaluateChunksTwoFiles*, para *OISVM* e *IGNGSVM*, se desarrollan con un único conjunto de train, pero obviando los resultados del primer bloque *TS*, mediante un tamaño de ventana (en *WindowClassificationPerformanceEvaluator)* igual al tamaño sumado del resto de bloques. Esto es necesario ya que, para las dos versiones incrementalizadas de *SVMs*, se trata de evaluar con el primer bloque antes de haber entrenado la máquina de soporte vectorial.

6.2. Conjuntos de datos usados

A continuación se describen los cuatro conjuntos utilizados en el estudio experimental de los distintos algoritmos, se explicaran sus propiedades y el origen de los datos, así como se nombrará el porqué de su uso.

Sea:

SEA fue desarrollado por *W. N. Street and Y. Kim* en *A streaming ensemble algorithm (SEA) for large-scale classification* (W.N. Street, 2001), y ha sido utilizado por varios algoritmos como prueba estándar del concepto de cambio. El conjunto de datos se caracteriza por largos períodos de tiempo sin ningún tipo de desviación, con ocasionales cambios bruscos en el límite de la clase. El conjunto consta de dos atributos útiles, un tercero referente al ruido del conjunto, y una clase binaria de valor 1 o 2. El conjunto de entrenamiento consta de 25000 puntos por clase relevante (50000 en total), donde el 10% de las instancias llevan la clase de ruido. En el fichero de test se utiliza un conjunto separado de 50000 puntos sin ruido.

Weather:

La administración nacional oceánica y atmosférica de EEUU ha recopilado mediciones de más de 9000 estaciones meteorológicas en todo el mundo (Administration U.S. National Oceanic and Atmospheric). Los registros se remontan a la década de 1930, ofreciendo un amplia gama de tendencias climáticas (temperatura, presión, velocidad del viento, …). Para su creación fueron seleccionadas ocho características en base a su disponibilidad, eliminando aquellos con una tasa de característica faltante por encima del 15%. Los demás valores faltantes fueron imputados por la media de las características en las instancias anteriores y siguientes. Las etiquetas de clase se basan en el indicador binario para cada lectura diaria de lluvia con 18159 lecturas (instancias): 5.698 (31%) positivos (lluvia) y 12.461 (69%) negativo (sin lluvia).

Elec2:

Este conjunto de datos se ha convertido en un referente en la clasificación streaming. Fue descrita por primera vez por (Harries, 1999)y utilizado después para varias comparaciones de rendimiento en (M. Baena-Garcia, 2006). El set contiene información para el mercado eléctrico de Nueva Gales del Sur: contiene 27.552 registros con fecha de mayo de 1996 a diciembre de 1998, haciendo referencia a periodos de 30 minutos sobre los que se añaden sub-muestras, haciendo un total de 45.312. Estos registros tienen siete atributos:, dos indicadores de marca de tiempo (día de la semana, hora), y cuatro aspectos co-variables de captura de la demanda de electricidad y suministro, y una etiqueta binaria de clase (UP, DOWN). Una de las características más atractivas de este data set es el concepto de deriva (concept drift) que se produce en los datos, ya que durante el periodo de registro, el mercado de la electricidad se expandió en áreas adyacentes (el contexto cambió).

Shuttle:

El conjunto de datos *Shuttle* es un conjunto de datos estacionario usado para evaluar los efectos de clústeres irregulares en el proceso de etiquetado (ello lo hace valioso en el estudio y validación de *IGNGSVM* y *OISVM*). La versión utilizada es un tratamiento del conjunto con dos clases, donde se siguen conservando los siete atributos, realizando un proceso similar al de (Ondrej Linda 2009) ('Tratamiento de los conjuntos **sección 5.3**). La definición original de este conjunto también se encuentra en fase de validación (ver **sección 5.1**).

6.3. Descripción del análisis y representación de los resultados

En este punto se describen la forma de representar las pruebas hechas, el objetivo que persiguen, y el análisis de estas, con la finalidad de facilitar la posterior lectura, estudio y entendimiento del trabajo.

i. Objetivo del estudio

El objetivo de la fase de estudio es probar los algoritmos con conjuntos de datos que no se han visto de forma detallada en la literatura.

Se desea comparar el nuevo algoritmo *Incremental Growing Neural Gas - Support Vector Machine* a los otros dos, cuya similitud se basa en su modo incremental y en el uso de máquinas de soporte vectorial. Además, la comparación de *IGNGSVM* con *OISVM* supondrá la comparación directa entre *ILVQ* y *GNG*, debido a la similitud del resto de la estructura de ambos.

Además, se desarrolla un estudio detallado del impacto de distintas regiones de los conjuntos sobre los algoritmos, con el propósito de explicar deficiencias y aptitudes de cada uno de los tres algoritmos.

ii. Descripción del análisis

El análisis realizado en este punto está orientado a medir las precisiones obtenidas para cada región o bloque de los conjuntos de prueba, a la par que observar los tiempos de cómputo consumidos por los algoritmos.

Se buscan los puntos débiles y fuertes de cada algoritmo a partir de la variedad de conjuntos de train y test seleccionados. Se analiza cada prueba por tiempos y resultados, deteniéndose en algunas para observar la evolución del algoritmo durante la fase de aprendizaje (crecimiento o decrecimiento de la precisión al recibir determinados bloques, distintos tiempos obtenidos en distintos bloques TS, etc.).

iii. Representación de las pruebas

Los resultados obtenidos son expresados, al igual que en fase de validación, en tablas y gráficas que muestran el cambio y la adaptación de distintos números N_TS y a los conjuntos en las distintas pruebas.

También se muestra la evolución del algoritmo durante las distintas iteraciones del aprendizaje, lo que facilita el análisis del impacto de las distintas regiones del fichero de entrenamiento y test sobre los algoritmos.

Por último, en la fase de estudio comparativo se detallan gráficas y tablas conjuntas que permiten una comparativa directa entre todas las pruebas realizadas.

6.4. Experimentación preliminar

A continuación, se detallan las pruebas previas hechas sobre cada algoritmo, de cara a la optimización de los parámetros utilizados en las pruebas. Se especifica también el tratamiento hecho sobre los conjuntos y la selección de tareas de evaluación en *MOA*.

i. Tratamiento de los conjuntos y selección de tareas de evaluación

Para la fase de estudio se recurrieron a los ya mencionados conjuntos: *Shuttle, Weather, Sea* y *Elec2*.

Debido a que Sea fue obtenido desde un principio en dos conjuntos (train y test), se usa el método de evaluación *EvaluateChunksTwoFiles*, cuyas propiedades corresponden al modo de evaluación utilizado en la literatura del conjunto. Dicha tarea de evaluación se ejecuta también para *Shuttle*, dividido en dos conjuntos como se detalla una sección atrás. La diferencia en las pruebas radica en que *Shuttle* es randomizado 10 veces distintas, y calculada la media de sus resultados y tiempos en el estudio. Esto en el resto de conjuntos elegidos no es posible por la no estacionalidad de sus ejemplos de entrenamiento. Ello rompería todo contexto de cambio existente.

Además de dividirse en la división en dos conjuntos (train y test), se ha decidid usar la versión de *Shuttle* de dos clases utilizada en la validación de *IGNGSVM* (**ver sección 5.4**). Esta última decisión ha sido tomada por la fragilidad latente en el algoritmo *GNGSVM* ante ejemplos de entrenamiento de las clases menos pobladas del conjunto; es decir, las que menos aparecen en el conjunto de entrenamiento (ver 'Híbrido *GNGSVM*' en **sección 2.5**). Por otro lado, el kernel utilizado para el estudio es el gaussiano. De este modo, podrán compararse los resultados de una forma más directa al resto de pruebas para *OISVM* e *IGNGSVM*, y apreciar el

efecto de no aplicar un kernel polinómico sobre este conjunto modificado para dos clases.

Por su parte, *Weather* y *Elec2* son conjuntos formados por un único fichero batch. En la fase de estudio se presentaron problemas al ejecutar la tarea *EvaluateInterleavedTestThenTrain* pues, como *Shuttle* en fase de validación, no respetaba la no estacionalidad de los datos. Por ello se ha realizado un tratamiento sobre ambos conjuntos, de forma que el fichero train está formado por las filas pares (empezando por la fila 0), y el fichero test las impares. Dicho tratamiento respeta el concept drift en ambos conjuntos no estacionarios, mediante su ejecución con la tarea de evaluación *EvaluateChunksTwoFiles*.

En último lugar, al comienzo de las pruebas se observó como el funcionamiento a modo continuo de *SGD* necesitaba de una tarea de evaluación que no mandase grupos de datos acumulados, sino ejemplos de entrenamiento de uno en uno. Por ello, las pruebas de *SGD* se realizan mediante la tarea *EvaluateInterleavedTestThenTrain*. En fase de estudio se hacen distintas pruebas y combinaciones entre los sets de prueba y entrenamiento de los conjuntos, de cara a determinar cual se debe añadir en las comparativas.

ii. Obtención de parámetros y elección de kernel en LibSVM y SGD

En primer lugar, de cara a la selección del kernel en las pruebas con *OISVM* e *IGNGSVM*, elegimos el kernel gaussiano ya que en la literatura se ha demostrado su mejor funcionamiento con los conjuntos usados en el estudio.

La selección de los parámetros Coste y Gamma para las pruebas de *OISVM* e *IGNGSVM* en cada conjunto, se realiza de la misma forma que se detalla en la sección de validación. Los valores obtenidos para C y Gamma en cada conjunto son mostrados en la **tabla 6.1.**

Conjuntos	Coste	Gamma
SEA	7.0	5.0E-4
WEATHER	1.0	0.0052726
ELEC2	2.0	5.0E-5
SHUTTLE	10.0	0.05

Tabla 6.1. *Los valores han sido obtenidos con CVParameterSelection, para rangos C=[0,10] y G=[5.0E-5,0].* Para las particiones de Weather y Elec2 en dos conjuntos se utilizaron los valores originales mostrados en la tabla, debido a buenos resultados tras su test.

Los valores de la **tabla 6.1.** son los obtenidos por *WEKA* para el conjunto completo. Ello hace que, sobre todo en los problemas cuyo contexto cambia, no supongan la mejor elección posible. Otra posibilidad hubiera sido obtener los parámetros para el primer bloque de datos (de *OISVM* o *IGNGSVM*), de cara a que comenzasen bien los conjuntos no estacionarios.

Centrándonos en *SGD*, las implementación utilizada de *MOA* (así como la de Bottou estudiada) sólo permite un kernel lineal. Esto debe tenerse muy en cuenta en la fase de estudio del **apartado 6.5**. Debido a que las primeras pruebas con los conjuntos tuvieron los resultados esperados, los parámetros utilizados se han dejado como venían por defecto.

iii. Proceso de reducción en OISVM y IGNGSVM

Como se ha visto en el estado del arte, así como mostrado en fase de validación, los dos algoritmos integrados a *MOA* usan algoritmos los algoritmos *ILVQ* y *GNG* con el fin de reducir la dimensionalidad de los datos de entrada y reducir el tiempo de proceso de los distintos bloques.

Obtención de los parámetros

El objetivo es encontrar parámetros para los algoritmos de clustering *ILVQ* y *GNG* (usados por *OISVM* e *IGNGSVM,* respectivamente) que obtengan en las pruebas un buen porcentaje de acierto, reduciendo también el número de ejemplos a un tamaño suficientemente pequeño (sin dejar de ser un conjunto representativo). En la **tabla 6.2** se muestran los parámetros utilizados en las pruebas para 10 y 20 bloques.

	OISVM 10		OISVM 20		IGNGSVM 10			IGNGSVM20		
	-l	-a	-l	-a	-l	-m	-c	-l	-m	-c
WEATHER	320	320	165	165	162	174	174	81	80	87
WEATHER_CHUNKS	320	321	165	166	165	175	175	81	80	175
ELEC2	400	50	75	50	1600	800	800	400	50	400
ELEC2_CHUNKS	1600	50	800	100	1600	400	400	400	50	400
SEA	130	130	65	65	130	100	100	70	50	100
SHUTTLE	800	15000	400	7500	400	100	100	200	7500	100

Tabla 6.2: Parámetros utilizados en los procesos de clustering de *ILVQ* y *GNG* en las pruebas. –l = lambda, -a = maxAge, -c = criterio de parada

Reducción obtenida

La **tabla 6.3** muestra la reducción relativa al tamaño de bloque |TS| en cada uno de los conjuntos completos estudiados, tras la aplicación de los parámetros vistos en el apartado anterior.

	OISVM 10	OISVM 20	IGNGSVM 10	IGNGSVM 20
WEATHER	0,40	0,41	0,10	0,10
WEATHER_CHUNKS	0,37	0,40	0,10	0,20
ELEC2	0,28	0,18	0,17	0,18
ELEC2_CHUNKS	0,47	0,48	0,09	0,18
SEA	0,06	0,06	0,02	0,04
SHUTTLE	0,30	0,29	0,04	0,02

Tabla 6.3: *Porcentaje de reducción relativa a |TS| medio entre todos los bloques TS procesados en cada uno de los conjuntos estudiados, con OISVM e IGNGSVM para diez y veinte bloques.*

Cabe aclarar que este es el porcentaje de reducción aplicado sobre la entrada del bloque TS. A la cantidad de datos de entrada para la máquina de soporte vectorial, hay que sumarle los vectores de soporte de la generación anterior. Por ello, y debido a la acumulación de SVs que se producen a modo de defecto de la estructura del algoritmo, esta cifra crecerá considerablemente cuanto mayor sea el N_TS (número de bloques TS en los que se divide el conjunto).

6.5. Comparación de los resultados obtenidos

A continuación, se exponen las pruebas hechas para la fase de estudio comparativo del trabajo. Tras dos secciones de pruebas, se exponen las conclusiones obtenidas.

6.5.1. Pruebas de la literatura

En este apartado se detallan y analizan los resultados obtenidos, tras haber explicado las decisiones tomadas en fase de estudio previo a las pruebas; donde se optimizan los parámetros de *ILVQ* y *GNG* buscando alcanzar el mejor porcentaje de acierto de cada prueba

A continuación, se muestran las pruebas realizadas en el primer apartado de la fase de estudio en cajas negras. Acto seguido, se exponen los resultados obtenidos en cuanto a porcentaje de acierto en la **Tabla 6.5.1**, con una batería de pruebas correspondiente a la literatura estudiada.

SEA: CHUNKTWOFILES CHUNK SIZE = 250	**WEATHER**: INTERLEAVED CHUNKS SIZE = 30	**ELEC2**: InterleavedTestThenTrain	**SHUTTLE**: (GenerateModel TRAIN) + TEST (ESTACIONARIO)

	Conjuntos usados y número de bloques (%)							
Algoritmos	SEA 10 N_TS	SEA 20 N_TS	WEATHER 10 N_TS	WEATHER 20 N_TS	ELEC2 10 N_TS	ELEC2 20 N_TS	SHUTTLE 10 N_TS	SHUTTLE 20 N_TS
LIBSVM	92,48		-		-		97,73	
OISVM	92.20	93.21	79,85	80,44	63,61	64,87	91,62	91,21
IGNGSVM	92.24	92.48	78,35	78,74	61,35	64,28	92,68	92,95
SGD	79.03		61,23		90,86		78,52	

Tabla 6.5.1: *Porcentajes de acierto obtenidos en los cuatro conjuntos extraídos de la literatura, en las pruebas con 10 y con 20 bloques. SEA y Shuttle están distribuidos en conjunto de test y otro entrenamiento, por lo que son evaluados con EvaluateChunksTwoFiles y GenerateModel* Train+ test, respectivamente. *Weather y Elec se evalúan con InterleavedChunks y InterleavedTestThenTrain, respectivamente. LibSVM y SGD no utilizan bloques en su evaluación. LIBSVM se evalúa mediante WEKA con 'Supplied test set' para SEA y Shuttle, pero no lo hace para Weather y Elec2, pues no hay distinción entre fichero de entrenamiento y fichero de test.*

En primer lugar, hay que aclarar que el hecho de no ejecutar pruebas para *Weather* y *Elec2* con *LibSVM* en la **Tabla 6.5.1**, se debe a que no tenemos un conjunto de test con el que probar el entrenamiento.

Respecto a los resultados, se observa como un aumento del número de bloques parece mejorar el porcentaje de éxito en los conjuntos no estacionarios (*Sea, Weather y Elec2).* Esto es especialmente consistente en *Elec2* donde, teóricamente, debido a sus bruscos cambios de contexto serán mejores bloques más pequeños de ejemplos de entrenamiento. Menor TS debería ayudar a que el modelo entrenado se encuentre menos desactualizado. Constancia de esto se aprecia en la prueba con *SGD* en *Elec2*, donde un modelo de que recuerde únicamente del anterior ejemplo consigue un 90,86% de precisión de acierto y tarda sólo medio segundo en ejecutarse (ver **Tabla 6.5.2**). *SGD* no obtiene buen resultado en ninguno de los otros tres conjuntos.

	Evaluation time (cpu seconds) en conjuntos usados y número de bloques							
Algoritmos	SEA 10 N_TS	SEA 20 N_TS	WEATHER 10 N_TS	WEATHER 20 N_TS	ELEC2 10 N_TS	ELEC2 20 N_TS	SHUTTLE 10 N_TS	SHUTTLE 20 N_TS
LIBSVM	257		-		-		68	
OISVM	9,58	7,84,	18,68	28,91	136,67	91,43	51,54	48,85
IGNGSVM	8,35	13,35	34,11	16,94	5403,98	1051,41	35,13	43,06
SGD	0,47		0,53		0,52		0,85	

Tabla 6.5.2: *Tiempos de cómputo (en segundos) obtenidos en cada una de las pruebas mostradas en la* ***tabla 6.5.1****.*

Puede visualizarse (en la **Tabla 6.5.1**) como el ajuste de los parámetros de *ILVQ* y *GNG* en busca de optimizar la precisión de acierto, merece la pena sobretodo en los conjuntos de prueba no estacionarios. Dicho ajuste hace que, para comentar la tabla de tiempos (**Tabla 6.5.2**), haya que observar de cerca la tabla de reducción relativa

(**Tabla 6.3,** del apartado anterior), donde la gran diferencia de compresión entre los dos algoritmos explica los resultados.

Los tiempos mostrados en la **Tabla 6.5.2** son la suma del tiempo demorado en la reducción de los conjuntos más el tiempo consumido por la ejecución de la SVM. Si bien, en teoría el gasto de tiempo de reducción debería favorecer a al tiempo total consumido (debido a que reduce el pesado procesamiento de la SVM), hay que tener especial cuidado con la excesiva compresión. Se observa como en *IGNGSVM* se obtienen tiempos de cómputo superiores para una reducción menor; a más alto criterio de parada, más instancias hay que generar. De hecho, especialmente en *GNG*, la condición de parada global (para todas las clases) hace que la fase de reducción tarde más tiempo del necesario (en las líneas futuras del trabajo se comenta este hecho y se ofrece una posible mejora a implementar sobre el algoritmo programado). También se observa el pésimo comportamiento de *Elec2* en *IGNGSVM* y *OISVM* respecto a *SGD*, pero se verifica cómo al ser divido en el doble de bloques no sólo aumenta su porcentaje de acierto, sino que también disminuye el tiempo de procesamiento de la SVM.

En resumen, vemos como los tiempos disminuyen al aplicar un menor tamaño de bloque TS, pero aumentan en *GNG* al disminuir la reducción aplicada (p.e. *Shuttle IGNGSVM 20* y *Sea IGNGSVM 20* demoran más que para diez bloques debido a una mayor reducción de los conjuntos). Para disminuir la reducción, hay que prefijar un criterio de parada (*-c*) mayor. Esto hace que *GNG* procese las instancias de cada subbloque S_i continuamente hasta generar, en rondas múltiplos de lambda, '*-c*' instancias de entrenamiento; cosa que hace aumentar el tiempo de cómputo. En resumen, en estas primeras pruebas *IGNGSVM* muestra una mejor relación porcentaje de acierto / reducción aplicada que *OISVM*, pero sus tiempos son peores y no llega a los resultados de *OISVM* en ninguno de los conjuntos no estacionarios (aunque parece superarlo en ambos en *Shuttle*). Por su parte, *SGD* muestra la necesidad de evaluarse de instancia en instancia, y no en bloques.

Las pruebas ejecutadas en *Sea* con *OISVM* e *IGNGSVM* mejoran los resultados de *LibSVM* tanto en tiempo como en porcentaje de acierto. Por otra parte, la mejoría de tiempos en *Shuttle* respecto a su ejecución con *LibSVM*, hace que la precisión baje de un 97 a un 92%.

Por su parte, *Weather* muestra un extraño comportamiento en *OISVM* al aumentar el tiempo consumido de 18 a 29 segundos. Este hecho podría deberse a una peculiaridad de los datos en el conjunto, para el que el clasificador demora más al aplicar el doble de bloques con similar porcentaje de reducción (reducción relativa al 40% y 41% de TS).

Una prueba con distinto evaluador de este conjunto nos ofrecería otro punto de vista del que poder extraer conclusiones. Siguiendo con este razonamiento, un evaluador que probase de 1 en 1 podría ser muy útil en pruebas que mostrasen el comportamiento de *SGD*. Para ello, se ofrece en las **Tablas 6.5.3 y 6.5.4** los resultados de aplicar *InterleavedTestandTrain* sobre *Weather*.

	WEATHER: InterleavedTestThenTrain (%)	
Algoritmos	**WEATHER 10 N_TS**	**WEATHER 20 N_TS**
OISVM	80,08	80,69
IGNGSVM	78,40	78,92
SGD	67,80	

Tabla 6.5.3.: *Resultados de Weather con InterleavedTestThenTrain.*

	WEATHER: InterleavedTestThenTrain (time)	
Algoritmos	**WEATHER 10 N_TS**	**WEATHER 20 N_TS**
OISVM	16,37	23,05
IGNGSVM	35,95	15,10
SGD	0,41	

Tabla 6.5.4.: *Tiempos de Weather con InterleavedTestThenTrain*

Se observan resultados aproximados a los mostrados con *InterleavedChunks*, por lo que el tamaño de 'chunk', así como el evaluador, no son los responsables del aumento de tiempo producido al reducir TS con *OISVM*. Volviendo a *SGD*, este no obtiene aquí los buenos resultados mostrados con *Elec2*. El resultado mejora algo, verificando que *SGD* funciona mejor con un evaluador 'de uno en uno' (pues su corta memoria sólo recuerda la última instancia entrenada). Esto puede deberse a que el kernel lineal de *SGD* no sea suficiente para representar los datos.

A modo de resumen, en este apartado se ha observado como, en general, la reducción del tamaño de bloque |TS| y la compresión de los bloques en *OISVM* (con efecto adverso en *ILVQ* que en *GNG*) han reducido el tiempo de cómputo total de las pruebas. *IGNGSVM* ha obtenido mejor acierto para conjuntos más pequeños. Sin embargo, los resultados obtenidos este algoritmo son inferiores a los de *OISVM* en los conjuntos no estacionarios, y quedan lejos de *LibSVM* en cuanto a precisión para *Shuttle*.

Por otro lado, los resultados de *Elec2* justifican el impacto que tiene el modo de evaluación en cada conjunto. Esto hace que para poder mejorar las conclusiones del estudio, así como para verificar las obtenidas en este apartado, sea preciso definir una forma única de evaluación para los cuatro algoritmos (salvo *LibSVM*) que nos permita comparar los resultados entre estos de la forma más precisa posible.

6.5.2. Pruebas usando una misma tarea de evaluación

El evaluador escogido para los cuatro sets de datos es *InterleavedChunksTwoFiles*, que requiere que cada conjunto de datos esté dividido en dos partes (conjunto de entrenamiento y conjunto de test). En las pruebas, se procesa primero un bloque de un tamaño dado |TS| del conjunto de entrenamiento, y a continuación se evalúa sobre un bloque del mismo tamaño del conjunto de test, que pertenece al mismo intervalo temporal. Dicha separación en conjunto de entrenamiento y test permite también usar una *Máquina de soporte vectorial* simple como medida de comparación. Con esta última, se genera el modelo sobre todo el conjunto de entrenamiento, y luego se evalúa sobre el de test.

Sea y *Shuttle* ya se encontraban divididos en dos conjuntos. Sin embargo, para los conjuntos de datos *Weather* y *Elec2* ha habido que generar un conjunto de test independiente del de entrenamiento. Para ello, se ha dividido el conjunto original en dos partes, alternando los datos de entrenamiento (líneas impares) con los de test (líneas pares). El orden de los datos no se ha altera para conservar así los momentos de los cambios de contexto, aunque esto no ha evitado tener que optimizar nuevamente los parámetros de *ILVQ* y *GNG*. Se recuerda que, en el caso de *Shuttle*, desde la fase de validación los resultados son calculados con la media de diez experimentos en los que se ha aleatorizado y dividido en dos conjuntos del mismo tamaño. Esto no ha podido realizarse en el resto de conjuntos al ser no estacionarios y ello impedirnos 'desordenar' los ejemplos de entrenamiento.

Hecho esto, hemos decidido evaluar el algoritmo según la predicción sobre cada elemento del conjunto de test, tras entrenar con la respectiva línea del conjunto de entrenamiento (*CHUNK SIZE=TS*). Nótese que, en realidad, el entrenamiento sigue realizándose en bloques para los algoritmos *OISVM* y *IGNGSVM*, ya que estos acumulan los datos de entrenamiento hasta disponer de un número TS de datos, y sólo entonces actualizan lo aprendido. De cara al análisis de resultados de *Elec2CHUNKS* y *WeatherCHUNKS*, hay que prestar atención a que la longitud de los conjuntos es ahora de la mitad, ya que ello reduce el tiempo estimado de los algoritmos.

A continuación se ofrecen las cajas negras (como en el primer apartado), que describen las pruebas realizadas para *IGNGSVM* y *OISVM*; y la **Tabla 6.5.7**, que ofrece los porcentajes de acierto obtenidos. *LibSVM* se ejecuta desde la herramienta *WEKA* para dos conjuntos (entrenamiento y test), y *SGD* se evalúa con *ChunksTwoFiles Chunks Size* = 1, para obtener mejores resultados (ver *SGD* en apartado 1 de estudio).

SEA: CHUNKTWOFILES CHUNK SIZE = TS	**WEATHER**: CHUNKTWOFILES CHUNK SIZE = TS	**ELEC2**: CHUNKTWOFILES CHUNK SIZE = TS	**SHUTTLE**: CHUNKTWOFILES CHUNK SIZE = TS

	Conjuntos usados y número de bloques (%)							
Algoritmos	**SEA 10 N_TS**	**SEA 20 N_TS**	**WEATHER CHUNKS 10 N_TS**	**WEATHER CHUNKS 20 N_TS**	**ELEC2 CHUNKS 10 N_TS**	**ELEC2 CHUNKS 20 N_TS**	**SHUTTLE 10 N_TS**	**SHUTTLE 20 N_TS**
LIBSVM	92,48		81,982		78,343		97,730	
OISVM	94,458	94,886	80,749	81,301	77,073	77,311	92,051	90,960
IGNGSVM	92,572	92,722	78,138	78,655	74,522	76,609	92,687	93,070
SGD	78,926		68,300		91,33		78,50	

Tabla 6.5.7: *Porcentajes de acierto obtenidos en los cuatro conjuntos extraídos de la literatura, en las pruebas con 10 y con 20 bloques. Todas las pruebas a excepción de LibSVM se han evaluado mediante EvaluateChunksTwoFiles. Elec2 y Weather han sido distribuidos en conjunto de test y otro entrenamiento. LibSVM y SGD no utilizan bloques en su evaluación. En LibSVM, las cuatro pruebas han sido efectuadas con 'Supplied test set' de WEKA.*

Verificamos aquí el comportamiento del apartado anterior. Aumentar el número de bloques mejora el porcentaje de éxito en los conjuntos no estacionarios (*Sea, Weather y Elec2).* En los resultados se muestra cómo también lo hace en *Shuttle IGNGSVM.* Se aprecia como el cambio de evaluación ha ayudado a mejorar los resultados. Una de las causas es la elección hecha del parámetro *CHUNKSIZE*. Al elegir un tamaño de TS para ambos parámetros del *Chunk*, no se evalúa el modelo hasta enviar el bloque de datos completo. Por ejemplo, en el apartado anterior, *Sea* hacia test con 250 ejemplos cada 250 entrenados. Debido a que *Sea* tiene 50000 ejemplos por conjunto, esto hace que el modelo se actualice en *OISVM* e *IGNGSVM* (deje de estar en buffer) cada veinte test para |TS| = 5000 (10 bloques). Es decir, la mayoría ejemplos se probarán con una máquina que aún no ha procesado los últimos ejemplos enviados por el evaluador. Esto tiene un grave impacto sobre entornos no estacionarios. Sobretodo en aquellos en que más rápido sea el cambio. Vemos un aumento significativo de un trece por ciento en precisión para *Elec2*; y una gran reducción en los tiempos de ejecución de la **Tabla 6.5.8** (aún teniendo en cuenta que el set tiene la mitad de tamaño).

	Evaluation time (cpu seconds) en conjuntos usados y número de bloques							
Algoritmos	**SEA 10 N_TS**	**SEA 20 N_TS**	**WEATHER CHUNKS 10 N_TS**	**WEATHER CHUNKS 20 N_TS**	**ELEC2 CHUNKS 10 N_TS**	**ELEC2 CHUNKS 20 N_TS**	**SHUTTLE 10 N_TS**	**SHUTTLE 20 N_TS**
LIBSVM	257		14		346		68	
OISVM	9,509	8,140	6,517	6,726	71,868	92,821	48,720	44,517
IGNGSVM	8,489	13,324	18,106	20,196	650,958	382,071	33,833	41,715
SGD	0,934		0,417		0,763		1,31	

Tabla 6.5.8.: *Tiempos de cómputo (en segundos) obtenidos en cada una de las pruebas mostradas en la* ***tabla 6.5.1****.*

En los tiempos vemos un comportamiento similar al del apartado anterior. Las pruebas que aumentan el tiempo de ejecución con 20 bloques respecto a las de 10, lo hacen influenciados por el nivel de compresión de *ILVQ* y *GNG* (ver **Tabla 6.3.**), excepto *Weather* en *OISVM* (como en las pruebas hechas previamente) y esta vez *Elec2 OISVM.* Weather Chunks sigue manteniendo la reducción relativa de las pruebas anteriores en *OISVM*, y es en *Elec2Chunks* donde varía. Complementando el apartado anterior, con los resultados obtenidos puede suponerse que este comportamiento se debe a una peculiaridad de los datos enviados a la SVM. Esto se demuestra en las pruebas de la **Tabla 6.5.9,** donde se busca una compresión similar en ambas pruebas dando valor proporcional a *lambda* y *maxAge*. Se observa como disminuye el porcentaje de acierto, pero el tiempo que demora la prueba para veinte bloques es menor que para diez.

Weather Chunks	OISVM 10 N_TS	OISVM 20 N_TS
Porcentaje	80.69	80.77
Tiempo	6.43	6.14

Tabla 6.5.9.: *Resultados de porcentaje de acierto y tiempo de la ejecución de Weather Chunks con OISVM para 10 y 20 bloques. Los parámetros lambda y maxAge utilizados fueron (600, 600) para 10 bloques y (300, 300) para 20 bloques.*

A continuación se detallan las conclusiones alcanzadas con cada set de prueba, así como para cada algoritmo.

Análisis de las pruebas sobre SEA:

Aunque su evaluador sea el mismo que en el primer apartado, la diferencia aquí es que en el segundo *CHUNKSIZE* = *TS*, cosa que hace que los resultados de test no se vean perjudicados por un modelo 'desactualizado'. Sus resultados en *OISVM* e *IGNGSVM* consiguen reducir los tiempos de *LibSVM* en un 3000% (de 257 se reduce a 8 y 9 segundos).

Puede diferirse, de sus resultados, la demostración cómo muchos conjuntos necesitan de un clasificador con memoria pese a tener un paisaje de cambio abrupto. Este conjunto de datos tiene cambios de contexto bruscos garantizados al haberse generado artificialmente con este propósito, aunque luego tiene zonas en las que estos permanecen igual. Creemos que es por ello que *SGD* no obtiene resultados competentes.

Análisis de las pruebas sobre Weather:

OISVM consigue mejorar el tiempo de *LibSVM*, pero se mantiene en cuanto al porcentaje. *IGNGSVM* empeora en porcentaje y tiempo (debido al tiempo de reducción). *SGD* se comporta de la misma forma que en *Sea*; como ya se ha mencionado, esto puede deberse a la incapacidad de representar el conjunto con un kernel lineal. Otra posibilidad es que la clasificación se vea perjudicada por la ausencia de memoria de *SGD*.

Análisis de las pruebas sobre ELEC2:

A diferencia de *OISVM*, que aporta una buena reducción de tiempo con respecto a *LibSVM* y un porcentaje parecido, el resultado obtenido para *IGNGSVM* empeora en un 2% la tasa de acierto y duplica los tiempos (esto es debido a la lenta compresión antes mencionada). Una apreciación de las ejecuciones del conjunto es cómo, a medida que avanza en el entrenamiento, crece el tiempo necesitado por la *SVM* para procesar los datos. Ya que el contexto de *Elec2* varía muy bruscamente, este hecho puede deberse a que la acumulación de los *SVs* y nuevas instancias distintas entre sí hagan cada vez más compleja la obtención de nuevos vectores de soporte. El resultado obtenido en *SGD*, de 91,33% de acierto en 0,763 segundos, demuestra como este conjunto no requiere memoria y es separable linealmente.

En las pruebas con *EvaluateChunkTwoFiles*, se observa una mejora en porcentaje y tiempo, respecto a las pruebas con *InterleavedTestAndTrain*. Debido a que en la primera también existen bloques (se deben a la estructura de *OISVM* e *IGNGSVM*), la mejora en la sincronización de los datos de test y train; con *EvaluateChunkTwoFiles* no se clasifica con modelos atrasados.

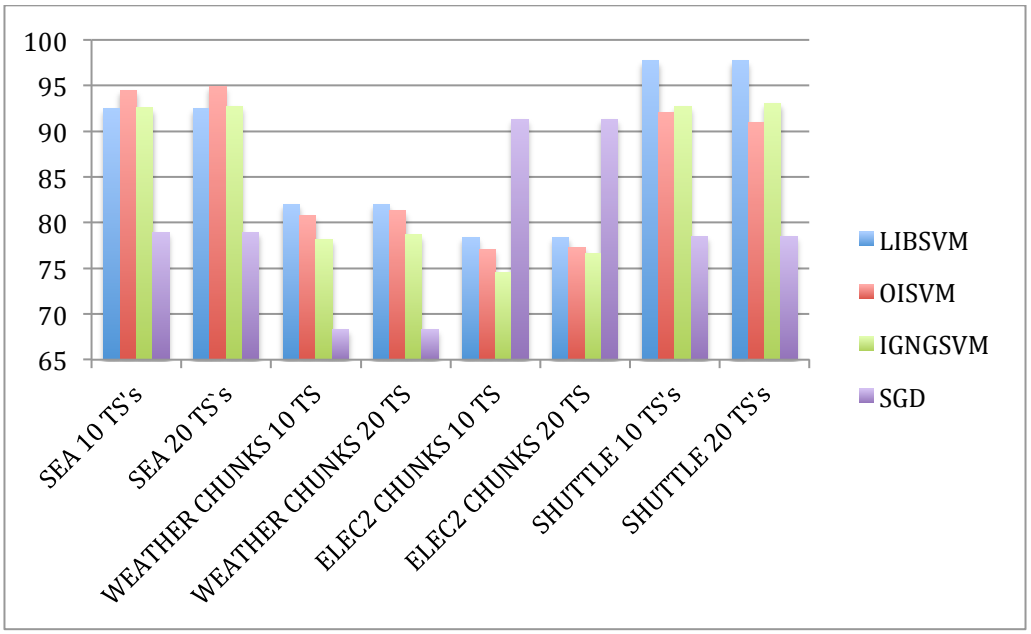
Esto coincide con lo mostrado en la literatura, donde se dice que en *Elec2* hay una fuerte correlación entre el dato presente y el dato anterior. *Elec2* demuestra por sus resultados en *SGD* que es simple de clasificar (kernel lineal vale) pero que los ejemplos pasados es mejor olvidarlos.

Análisis de las pruebas sobre Shuttle:

OISVM e *IGNGSVM* consiguen reducir el tiempo de las pruebas respecto a *LibSVM*, pero pierden alrededor de un 4% de acierto en el mejor de los casos; algo que para un conjunto como *Shuttle* (en el que el acierto medio es de 80%) es una reducción de porcentaje demasiado alta. Por su parte, *SGD* obtiene unos resultados pésimos, quedando por debajo del 80% de acierto. Esto podría deberse a que, para un conjunto de datos estacionario de gran tamaño como *Shuttle,* no sea conveniente el uso de un algoritmo sin memoria. También podría ocurrir que *SGD* no fuera capaz de representar *Shuttle* con un kernel lineal.

Comparación entre los distintos algoritmos

Tras la analizar los resultados, la primera conclusión es que, para todos los conjuntos, *OISVM* e *IGNGSVM* mejoran los tiempos de cómputo respecto a *LibSVM*. Esto puede explicarse gracias a la reducción recibida sobre cada bloque de datos. En concreto, para grandes conjuntos de datos el tiempo consumido por *IGNGSVM* será mejor que el de *OISVM*, obteniendo una mejor relación porcentaje de acierto/ reducción aplicada. (ver **Gráficas 6.5.1 y 6.5.3**).



Gráfica 6.5.1: *Porcentajes de acierto obtenidos en las distintas pruebas ejecutadas en cada conjunto. Se muestran gráficamente los resultados mostrados en la* ***tabla 6.5.7****. El eje representa la precisión obtenida de cada conjunto; cada punto del eje x es un conjunto sobre los cuatro algoritmos.*

	SEA 10 N_TS	SEA 20 N_TS
LIBSVM	7,52	
OISVM	5,542	5,114
IGNGSVM	7,428	7,278

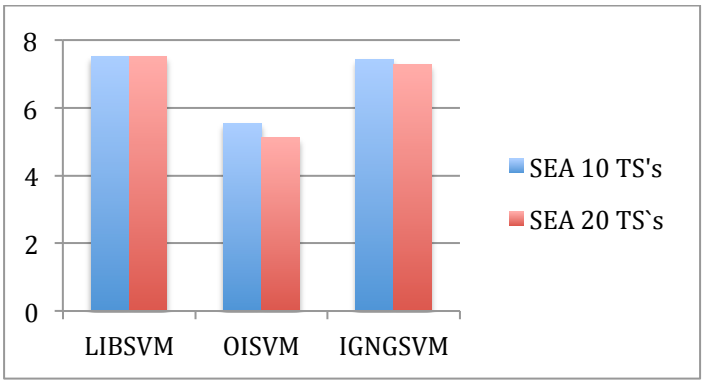
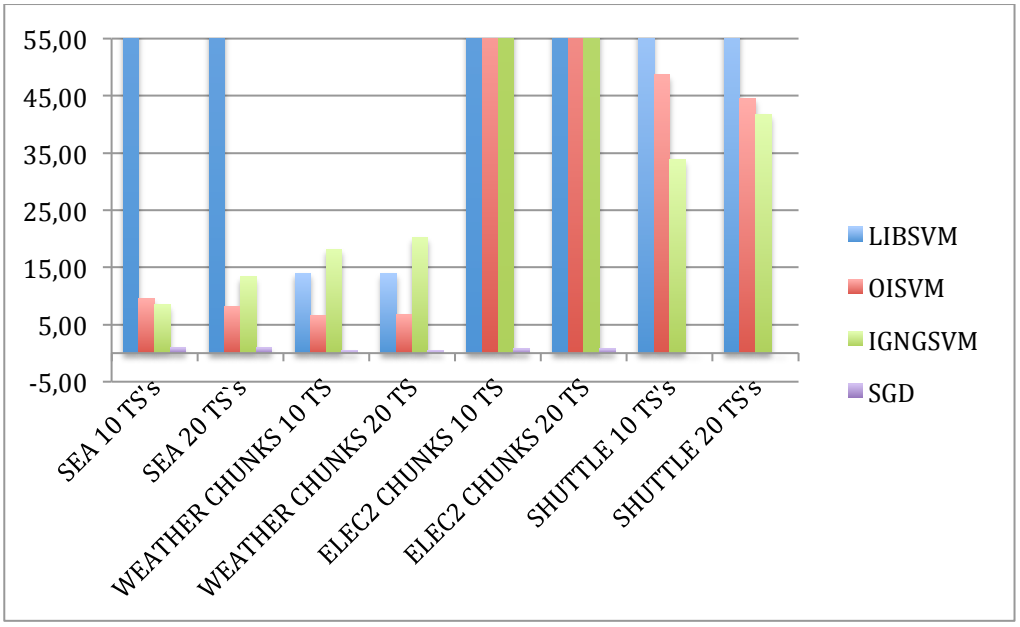


Tabla 6.5.10 y gráfica 6.5.2: *Resultados de la tabla 1.7 en porcentaje de error de la ejecución de ChunksTwoFiles en SEA para OISVM e IGNGSVM, frente a los resultados para LibSVM.*

Se observa como excepto para *Sea*, cuyos resultados mejoran tanto en tiempo (ver **Gráfica 6.5.2**) como en tasa de acierto de forma significativa, en el resto de sets sólo se igualan o empeoran los resultados de *LibSVM*. Pensamos que esto es bastante lógico, ya que toda reducción de un conjunto hace que el modelo pierda ejemplos que pese a tener menos peso según el algoritmo, pueden hacer variar el resultado de clasificación. Los resultados obtenidos con *Shuttle* (ver **Gráfica 6.5.3**) nos hacen ver como para conjuntos de gran tamaño se puede reducir el tiempo de cómputo de las SVM's, pero que en ciertos conjuntos pueden suponer un coste demasiado alto. Concretamente, aquí la pérdida de precisión relativa a *LibSVM* se acerca al 7%. Según (Catlett, 2006), teniendo en cuenta que el conjunto probado cuenta de dos clases con repartición de 80 y 20% de instancias respectivamente, se busca una precisión que tienda al 99%.



Gráfica 6.5.3: *Tiempo de cómputo consumido en las distintas pruebas ejecutadas en cada conjunto. Se muestran gráficamente los resultados mostrados en la **tabla 6.5.8**.*

Respecto a *Weather* y *Elec2,* sus resultados nos hacen pensar que los dos algoritmos propuestos para la incrementalización de *LibSVM*, podrían no capturar el momento de cambio de los conjuntos no estacionarios. Los dos algoritmos 'incrementales' deberían obtener mejores resultados que un algoritmo batch como es *LibSVM*, pero esto no es así. Una de las explicaciones para estos resultados puede ser que en *Weather*, a diferencia de *SEA*, el cambio de contexto no sea tan importante como para compensar el hecho de que se parte el conjunto y se entrena con menos. En *Elec2* podría darse un efecto distinto, debido a que dicho conjunto está diseñado (según la literatura) de forma que un ejemplo de entrenamiento suele depender directamente del anterior.

6.5.3. Conclusiones del estudio

Las pruebas hechas sobre *SEA*, en *OISVM* e *IGNGSVM* consiguen reducir los tiempos de *LibSVM* en un 3000% (de 257 se reduce a 8 y 9 segundos), así como mejorar sus resultados en precisión de acierto. Se muestra en la precisión y los tiempos obtenidos, cómo muchos conjuntos necesitan de un clasificador con memoria pese a tener cambios de contexto bruscos.

SGD se muestra necesario en conjuntos como *Elec2*. Aquí los cambios bruscos de contexto hacen que un algoritmo de kernel lineal que sólo mantenga en memoria la última instancia sea ideal para la clasificación. Se aprecia como el hecho de olvidar lo previo en conjuntos estacionarios, o no estacionarios cuyo paisaje de cambio no es tan abrupto, perjudica la clasificación. Además, no hay que olvidar los resultados de tiempo de *SGD* frente a los de una *SVM*. El hecho de hacer separables los datos de forma lineal y procesarlos en un tiempo menor a un segundo, hace que ninguno de los demás algoritmos estudiados pueda hacerle competencia en sets con propiedades similares.

Los resultados de *Weather* y *Elec2* nos hacen temer que los dos algoritmos propuestos para la incrementalización de *LibSVM*, podrían no estar capturando de forma correcta el momento de cambio. Podría deberse a que en *Weather*, a diferencia de *SEA*, el cambio de contexto no sea tan importante como para compensar el hecho de que se parte el conjunto y se entrena con menos. En *Elec2* el efecto podría ser incluso diferente. Los dos algoritmos 'incrementales' deberían obtener mejores resultados que un algoritmo batch como es *LibSVM*, pero esto no es así. Estos factores resaltan la importancia de un estudio previo antes de escoger un algoritmo u otro para distintos conjuntos.

Otro problema que podría afectar es que se han optimizado los parámetros de *LibSVM* (gamma y coste) para el conjunto completo en modo batch. Pese a optimizar la reducción de los conjuntos efectuada por *G* y *S* en *ILVQ* y *GNG*, respectivamente en busca de la mejor tasa de acierto, esta podría verse condicionada por unos parámetros internos de la SVM que no funcionan tan bien sobre conjuntos reducidos, o sobre cada bloque de datos (donde el único recuerdo de las iteraciones pasadas son los SVs). Esto también puede hacer pensar que quizá el guardado de los SVs del bloque anterior no sea suficiente para mantener la precisión del set en modo batch, sobretodo en conjuntos de gran tamaño

como *Shuttle*. Obviamente, aún perdiendo precisión por éste último factor, en caso de conjuntos no estacionarios se debería ganar acierto por asumir el *concept drift*.
En la literatura, no siempre se encuentra esta comparación con los resultados "base" de un algoritmo batch como LibSVM. Nos ha parecido interesante incluirla ya que podría sugerir que algunos de los conjuntos de datos no son especialmente buenos para observar el cambio de contexto o evaluar los algoritmos diseñados para capturarlo.

Las pruebas demuestran que aumentando el número de bloques en los que se divide el set, se aumenta la precisión obtenida en los casos no estacionarios, mientras que se mantiene en conjuntos estacionarios. Es ello lo que hace entender que, aún valorando la posibilidad de mejorar los parámetros de la SVM para el procesamiento por bloques, un mejor resultado de los algoritmos llegaría optimizando el número de bloques en que se va a dividir cada conjunto de ejemplos.

Por otro lado, se ha mostrado cómo en el caso de *IGNGSVM*, el algoritmo consume más tiempo cuanto menos se reduce el conjunto (mayor criterio de parada). Por lo que una mayor reducción del conjunto, mejora tanto el tiempo de reducción así como el tiempo gastado por *LibSVM* para establecer los SVs. También es destacable que en *GNG* este hecho se ve aumentado al haber programado una condición de parada global para todas las clases. Lo cual perjudica en tiempo los criterios altos de parada sobretodo en las clases con presencia minoritaria en la topología de *GNG*.

En cuanto a la comparación de *OISVM* e *IGNGSVM*, en *Shuttle* los resultados de *IGNGSVM* son algo mejores, pero siguen lejos de *LibSVM* en cuanto a precisión. En apariencia, *OISVM* muestra un mejor funcionamiento en los tres casos no estacionarios. Creemos que los resultados dependen de cómo se realiza la reducción del bloque de entrenamiento TS a un conjunto de prototipos representativos. En *OISVM* el algoritmo que realiza esto es *ILVQ*, y en *IGNGSVM* es *GNG*. No hay que pasar por alto, que *ILVQ* es mucho más sensible a los parámetros que *GNG*. Esto ha requerido especial atención en fase de estudio previo y, por ello, el uso de *IGNGSVM* podría convenir antes que *OISVM* en estudios en que el tiempo de optimización de parámetros de *ILVQ* no fuese asumible (siempre y cuando la pérdida de precisión no fuese considerable).

Por otra parte, los resultados obtenidos con *IGNGSVM* pese a ser menores en precisión obtenida que *OISVM*, son esperados, debido a que *GNG* es un conjunto diseñado para funcionar con millones de datos. Esto se verifica al mostrar en las pruebas mejor relación porcentaje de acierto / reducción aplicada que *OISVM*. El único conjunto suficientemente grande de los usados para *IGNGSVM* es *Shuttle*, y en él se ve la mejora respecto a *OISVM*. Por lo que, tanto para casos cuyo tiempo de optimización de parámetros se requiere que sea menor, y probablemente para conjuntos grandes (pues no hemos hecho suficientes pruebas como para afirmarlo), convendrá el uso de *IGNGSVM* antes que *OISVM*.

7. PLANIFICACIÓN

En secciones previas se han explicado los objetivos del proyecto, mostrados los aspectos teóricos relacionados y una introducción a las herramientas de análisis utilizadas. También se ha hablado del proceso de integración y validación de los algoritmos, comentando problemas encontrados y, finalmente, se ha realizado un estudio comparativo entre los clasificadores.

Aquí, se muestra la planificación seguida desde el comienzo del proyecto, que contiene la asignación de tiempo tanto en horas de trabajo como en días, el orden de realización de las tareas, y la comparación al trabajo final realizado.

La planificación se ha realizado en días de trabajo y el tiempo de trabajo durante cada día se ha medido en horas. Las tablas del trabajo hecho y planificado cada diez días se encuentran divididas en dos, para que quepan en el tamaño de las hojas del documento.

El trabajo se ha dividido en las siguientes partes:

- **Planificación.** Etapa mostrada en este apartado.

- **Preparación del entorno.** Esta fase incluye: instalación del programa, preparación del entorno de programación, realización de tutoriales y búsqueda de librerías a utilizar.

- **Estudio del entorno y del estado del arte.** Aquí se realizó: el estudio sobre algoritmos de aprendizaje incremental, el estudio del funcionamiento de SVM, el estudio de *ILVQ*, el estudio de la creación de clasificadores en *MOA*, y la familiarización con el entorno de *LibSVM*.

- **Estudio y desarrollo de los clasificadores**. En esta etapa figura, para cada clasificador: el estudio de los clasificadores, el diseño de los algoritmos, o estudio de algoritmos integrados previos, y la validación del clasificador programado.

- **Pruebas:** Es el estudio comparativo de los algoritmos integrados.

- **Documentación:** Proceso de documentación de todo el proyecto para elaborar el presente documento.

7.1. Planificación inicial semanal de las tareas (Diagrama de GANTT)

Se puede ver la planificación diaria inicial realizada en la imagen de abajo.

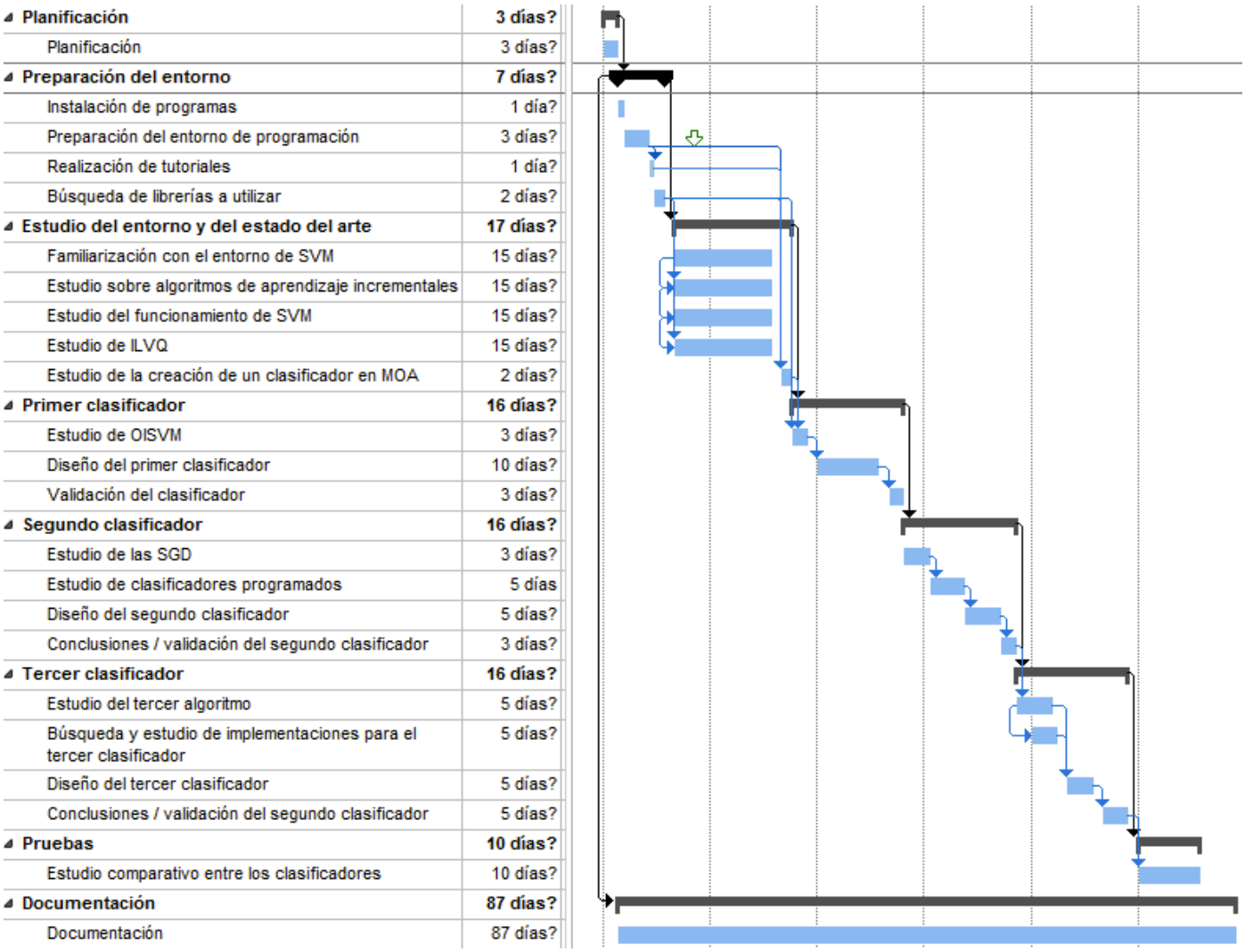


Figura 7.1: *Diagrama de Gantt del proyecto*

Respecto a la **figura 7.1**, se observa como la planificación sólo dura tres días. Esto en las tablas de tiempo por horas se representa de forma distribuida durante todo el proyecto, ya que cada semana ha habido que ir apuntando los tiempos consumidos. Esto he decidido no estimarlo en el diagrama de Gantt, ya que la mayor parte del tiempo dedicado a este punto es al comienzo del proyecto.

7.2. Ampliación del plazo del proyecto y re-planificación

Desde un principio el proyecto fue planificado para ser entregado en octubre de 2012 en convocatoria extraordinaria. Tras no celebrarse dicha convocatoria, el plazo del proyecto se amplió hasta febrero de 2013. Para aprovechar esta ampliación, realizamos una re-planificación a partir del estado del proyecto.

Puesto que el orden de las tareas no varía, siendo el tiempo dedicado el único cambio, no es necesario repetir en este punto el diagrama de Gantt descrito inicialmente en la **figura**

7.1. La nueva distribución del tiempo planificado se aprecia en los dos siguientes apartados.

7.3. Horas planificadas y realizadas

En un comienzo se ha planificado el número de horas trabajadas cada día, como figura en las tablas de abajo.

Planificación	Día 1	Día 2	Día 3															Horas totales
Planificación	2 horas	2 horas	2 horas															6
Preparación del entorno	**Día 1**	**Día 2**	**Día 3**	**Día 4**	**Día 5**	**Día 6**	**Día 7**											
Instalación de programas	2 horas																	2
Preparación del entorno de programación		3 horas	3 horas	3 horas														9
Realización de tutoriales					3 horas													3
Búsqueda de librerías a utilizar						3 horas	3 horas											6
Estudio del entorno y del estado del arte	**Día 1**	**Día 2**	**Día 3**	**Día 4**	**Día 5**	**Día 6**	**Día 7**	**Día 8**	**Día 9**	**Día 10**	**Día 11**	**Día 12**	**Día 13**	**Día 14**	**Día 15**	**Día 16**	**Día 17**	
Familiarización con el entorno de SVM	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora			15
Estudio sobre algoritmos de aprendizaje incrementales	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora			15
Estudio del funcionamiento de SVM	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora			15
Estudio de ILVQ	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora	1 hora			15
Estudio de la creación de un clasificador en MOA																2 horas	2 horas	4
Primer clasificador	**Día 1**	**Día 2**	**Día 3**	**Día 4**	**Día 5**	**Día 6**	**Día 7**	**Día 8**	**Día 9**	**Día 10**	**Día 11**	**Día 12**	**Día 13**	**Día 14**	**Día 15**	**Día 16**		
Estudio de OISVM	2 horas	2 horas	2 horas															6
Diseño del primer clasificador				2 horas	2 horas	2 horas	2 horas	2 horas	2 horas	2 horas	2 horas	2 horas	2 horas					20
Validación del clasificador														2 horas	2 horas	2 horas		6
Segundo clasificador	**Día 1**	**Día 2**	**Día 3**	**Día 4**	**Día 5**	**Día 6**	**Día 7**	**Día 8**	**Día 9**	**Día 10**	**Día 11**	**Día 12**	**Día 13**	**Día 14**	**Día 15**	**Día 16**		
Estudio de las SGD	2 horas	2 horas	2 horas															6
Estudio de clasificadores programados				2 horas	2 horas	2 horas	2 horas	2 horas										10
Diseño del segundo clasificador									3 horas	3 horas	3 horas	3 horas	3 horas					15
Conclusiones / validación del segundo clasificador														2 horas	2 horas	2 horas		6
Tercer clasificador	**Día 1**	**Día 2**	**Día 3**	**Día 4**	**Día 5**	**Día 6**	**Día 7**	**Día 8**	**Día 9**	**Día 10**	**Día 11**	**Día 12**	**Día 13**	**Día 14**	**Día 15**	**Día 16**		
Estudio del tercer algoritmo	2 horas	2 horas	2 horas	2 horas	2 horas													10
Búsqueda y estudio de implementaciones para el tercer clasificador		2 horas	2 horas	2 horas	2 horas	2 horas												10
Diseño del tercer clasificador							2 horas	2 horas	2 horas	2 horas	2 horas							10
Conclusiones / validación del segundo clasificador												2 horas	2 horas	2 horas	2 horas	2 horas		10
Pruebas	**Día 1**	**Día 2**	**Día 3**	**Día 4**	**Día 5**	**Día 6**	**Día 7**	**Día 8**	**Día 9**	**Día 10**								
Estudio comparativo entre los clasificadores	4 horas	4 horas	4 horas	4 horas	4 horas	4 horas	4 horas	4 horas	4 horas	4 horas								40
Documentación *(Proceso paralelo al resto)*	**Días 1 al 10**	**Días 11 al 20**	**Días 21 al 30**	**Días 31 al 40**	**Días 41 al 50**	**Días 51 al 60**	**Días 61 al 70**	**Días 71 al 80**	**Días 81 al 87**									
Documentación	5 horas	15 horas	20 horas	10 horas	10 horas	10 horas	10 horas	10 horas	10 horas									100
																	Total Global	339

Tabla 7.1: *Tabla de trabajo estimado en horas, de los días planificados.*

Planificado total	Días 1 al 10	Días 11 al 20	Días 21 al 30	Días 31 al 40	Días 41 al 50	Días 51 al 60	Días 61 al 70	Días 71 al 80	Días 81 al 87	Total
Planificación	6	0	0	0	0	0	0	0	0	6
Preparación del entorno	20	0	0	0	0	0	0	0	0	20
Estudio del entorno y del estado del arte	0	40	24	0	0	0	0	0	0	64
Primer clasificador	0	0	6	20	6	0	0	0	0	32
Segundo clasificador	0	0	0	0	14	23	0	0	0	37
Tercer clasificador	0	0	0	0	0	2	28	10	0	40
Pruebas	0	0	0	0	0	0	0	20	20	20
Documentación *(Proceso paralelo al resto)*	5	15	20	10	10	10	10	10	10	100
Horas planificadas para los días	31	55	50	30	30	35	38	40	30	
Planificado total acumulado	31	86	136	166	196	231	269	309	339	

Tabla 7.2: *Tabla de trabajo estimado inicial en horas, en intervalos de diez días.*

	1 a 10 de junio	11 a 20 de junio	21 a 30 de junio	1 a 10 de julio	11 a 20 de julio	21 a 30 de julio	6 al 15 de agosto	24 de agosto al 3 de septiembre	4 al 14 de septiembre	15 al 24 de septiembre	25 de septiembre al 4 de octubre	5 al 14 de octubre
Planificado total	Días 1 al 10	Días 11 al 20	Días 21 al 30	Días 31 al 40	Días 41 al 50	Días 51 al 60	Días 61 al 70	Días 71 al 80	Días 81 al 90	Días 90 al 101	Días 101 al 110	Días 111 al 120
Planificación	3	0,5	0,5	0,5	0,5	0,5	0,5	0,5	0,5	2	0,25	0,25
Preparación del entorno	8	3	0	0	0	0	0	0	0	0	0	0
Estudio del entorno y del estado del arte	8	20	15	15	15	15	0	0	10	0	0	0
Primer clasificador	0	0	0	13	10	2	1	1	2	0	0	0
Segundo clasificador	0	0	0	0	5	0	0	0	0	0	0	0
Tercer clasificador	0	0	0	0	0	0	0	0	10	5	5	5
Pruebas	0	0	0	0	0	0	0	0	0	0	5	5
Documentación *(Proceso paralelo al resto)*	3	8	8	10	10	25	25	8	10	5	3	3
Horas trabajadas en dichos días	22	31,5	23,5	38,5	40,5	42,5	26,5	9,5	32,5	12	13,25	13,25
Trabajado total acumulado	22	53,5	77	115,5	156	198,5	225	234,5	267	279	292,25	305,5

	15 al 24 de octubre	24 de octubre al 2 de noviembre	3 al 12 de noviembre	13 al 22 de noviembre	1 al 10 de diciembre	11 al 20 de diciembre	21 al 30 de diciembre	31 de diciembre al 9 de enero	10 al 19 de enero	20 al 29 de enero	
Planificado total	**Días 121 al 130**	**Días 131 al 140**	**Días 141 al 150**	**Días 151 al 160**	**Días 161 al 170**	**Días 171 al 180**	**Días 181 al 190**	**Días 191 al 200**	**Días 201 al 210**	**Días 211 al 220**	Total
Planificación	0,25	0,25	0,25	0,25	0,25	0,25	0,25	0,25	0,25	0,25	12
Preparación del entorno	0	0	0	0	0	0	0	0	0	0	11
Estudio del entorno y del estado del arte	0	0	0	0	0	0	0	0	0	0	98
Primer clasificador	0	0	0	0	0	0	0	0	0	0	29
Segundo clasificador	0	0	0	0	0	0	0	0	0	0	5
Tercer clasificador	0	0	0	0	0	0	0	0	0	0	25
Pruebas	5	5	5	5	5	5	0	0	2	0	42
Documentación *(Proceso paralelo al resto)*	3	3	2	2	2	2	2	2	2	2	140
Horas trabajadas en dichos días	8,25	8,25	7,25	7,25	7,25	7,25	2,25	2,25	4,25	2,25	
Trabajado total acumulado	313,8	322	329	336,5	343,75	351	353,25	355,5	360	362	

Tabla 7.3: *Tabla de trabajo planificado final en horas (**ver sección 7.2**), en intervalos de diez días. Las columnas coloreadas en colores más fuertes representan el trabajo ya realizado en el momento de la re-planificación.*

Finalmente, se ha hecho una tabla con las horas aproximadas trabajadas a lo largo del proyecto, a modo comparativo:

	1 a 10 de junio	11 a 20 de junio	21 a 30 de junio	1 a 10 de julio	11 a 20 de julio	21 a 30 de julio	6 al 15 de agosto	24 de agosto al 3 de septiembre	4 al 14 de septiembre	15 al 24 de septiembre	25 de septiembre al 4 de octubre	5 al 14 de octubre
Trabajado total	**Días 1 al 10**	**Días 11 al 20**	**Días 21 al 30**	**Días 31 al 40**	**Días 41 al 50**	**Días 51 al 60**	**Días 61 al 70**	**Días 71 al 80**	**Días 81 al 90**	**Días 91 al 101**	**101 al 110**	**Días 111 al 120**
Planificación	3	0,25	0,25	0,25	0,25	0,25	0,25	0,25	0,25	2	0,25	0,25
Preparación del entorno	8	3	0	0	0	0	0	0	0	0	0	0
Estudio del entorno y del estado del arte	8	20	15	15	15	15	0	0	10	0	0	0
Primer clasificador	0	0	0	13	10	2	1	1	2	0	0	0
Segundo clasificador	0	0	0	0	5	0	0	0	0	0	0	0
Tercer clasificador	0	0	0	0	0	0	0	0	10	8	15	0
Pruebas	0	0	0	0	0	0	0	0	0	0	5	0
Documentación *(Proceso paralelo al resto)*	3	8	8	10	10	18	18	8	5	10	10	5
Horas trabajadas en dichos días	22	31,25	23,25	38,25	40,25	35,25	19,25	9,25	27,25	20	30,25	5,25
Trabajado total acumulado	22	53,25	76,5	114,75	155	190,3	209,5	218,75	246	266	296,25	301,5

	15 al 24 de octubre	24 de octubre al 2 de noviembre	3 al 12 de noviembre	13 al 22 de noviembre	1 al 10 de diciembre	11 al 20 de diciembre	21 al 30 de diciembre	31 de diciembre al 9 de enero	10 al 19 de enero	20 al 29 de enero	
Trabajado total	**Días 121 al 130**	**Días 131 al 140**	**Días 141 al 150**	**Días 151 al 160**	**Días 161 al 170**	**Días 171 al 180**	**Días 181 al 190**	**Días 191 al 200**	**Días 201 al 210**	**Días 211 al 220**	Total
Planificación	0,25	0,25	0,25	0,25	0,25	0,25	0,25	0,25	0,25	0,25	10
Preparación del entorno	0	0	0	0	0	0	0	0	0	0	11
Estudio del entorno y del estado del arte	0	0	0	0	0	0	0	0	0	0	98
Primer clasificador	0	0	0	2,5	6	0	0	0	0	0	37,5
Segundo clasificador	0	0	0	3	0	0	0	0	0	0	8
Tercer clasificador	0	0	0	2,5	1	0	0	0	0	0	36,5
Pruebas	1,5	4,5	7,5	5	0	0	0	0	2	0	25,5
Documentación *(Proceso paralelo al resto)*	6	4	5,5	4,5	9,5	10	2,5	4	8,5	7,5	175
Horas trabajadas en dichos días	7,75	8,75	13,3	17,75	16,75	10,25	2,75	4,25	10,8	7,75	
Trabajado total acumulado	301,5	310,25	324	341,3	365,75	376	378,75	383	394	401,5	

Tabla 7.4: *Tabla de trabajo realizado en horas, en intervalos de diez días.*

7.4. Seguimiento del proyecto

Además de la planificación previa, también se han ido contabilizando las horas acumuladas de trabajo a lo largo del desempeño del proyecto. Abajo puede verse una comparativa en dos tablas en una gráfica que entrelaza ambas.

Planificado total acumulado	Días 1 al 10	Días 11 al 20	Días 21 al 30	Días 31 al 40	Días 41 al 50	Días 51 al 60	Días 61 al 70	Días 71 al 80	Días 81 al 87	Total
Planificación	6	6	6	6	6	6	6	6	6	6
Preparación del entorno	20	20	20	20	20	20	20	20	20	20
Estudio del entorno y del estado del arte	0	40	64	64	64	64	64	64	64	64
Primer clasificador	0	0	6	26	32	32	32	32	32	32
Segundo clasificador	0	0	0	0	14	37	37	37	37	37
Tercer clasificador	0	0	0	0	0	2	30	40	40	40
Pruebas	0	0	0	0	0	0	0	20	40	40
Documentación *(Proceso paralelo al resto)*	5	20	40	50	60	70	80	90	100	100

Tabla 7.5: *Tabla inicial de trabajo estimado acumulado en horas, en intervalos de diez días.*

Planificado total acumulado	Días 1 al 10	Días 11 al 20	Días 21 al 30	Días 31 al 40	Días 41 al 50	Días 51 al 60	Días 61 al 70	Días 71 al 80	Días 81 al 90	Días 90 al 101	Días 101 al 110	Días 111 al 120
Planificación	3	3,5	4	4,5	5	5,5	6	6,5	7	9	9,25	9,5
Preparación del entorno	8	11	11	11	11	11	11	11	11	11	11	11
Estudio del entorno y del estado del arte	8	28	43	58	73	88	88	88	98	98	98	98
Primer clasificador	0	0	0	13	23	25	26	27	29	29	29	29
Segundo clasificador	0	0	0	0	5	5	5	5	5	5	5	5
Tercer clasificador	0	0	0	0	0	0	0	0	10	15	20	25
Pruebas	0	0	0	0	0	0	0	0	0	0	5	10
Documentación *(Proceso paralelo al resto)*	3	11	19	29	39	64	89	97	107	112	115	118

Planificado total acumulado	Días 121 al 130	Días 131 al 140	Días 141 al 150	Días 151 al 160	Días 161 al 170	Días 171 al 180	Días 181 al 190	Días 191 al 200	Días 201 al 210	Días 211 al 220	Total
Planificación	9,75	10	10,3	10,5	10,75	11	11,25	11,5	11,8	12	12
Preparación del entorno	11	11	11	11	11	11	11	11	11	11	11
Estudio del entorno y del estado del arte	98	98	98	98	98	98	98	98	98	98	98
Primer clasificador	29	29	29	29	29	29	29	29	29	29	29
Segundo clasificador	5	5	5	5	5	5	5	5	5	5	5
Tercer clasificador	25	25	25	25	25	25	25	25	25	25	25
Pruebas	15	20	25	30	35	40	40	40	42	42	42
Documentación *(Proceso paralelo al resto)*	121	124	126	128	130	132	134	136	138	140	140

Tabla 7.6: *Tabla final (**ver sección 7.2**) de trabajo planificado acumulado en horas, en intervalos de diez días. Las celdas oscuras representan el trabajo realizado acumulado en el momento de la re-planificación.*

Trabajado acumulado	Días 1 al 10	Días 11 al 20	Días 21 al 30	Días 31 al 40	Días 41 al 50	Días 51 al 60	Días 61 al 70	Días 71 al 80	Días 81 al 90	Días 91 al 101	Días 101 al 110	Días 111 al 120
Planificación	3	3,25	3,5	3,75	4	4,25	4,5	4,75	5	7	7,25	7,5
Preparación del entorno	8	11	11	11	11	11	11	11	11	11	11	11
Estudio del entorno y del estado del arte	8	28	43	58	73	88	88	88	98	98	98	98
Primer clasificador	0	0	0	13	23	25	26	27	29	29	29	29
Segundo clasificador	0	0	0	0	5	5	5	5	5	5	5	5
Tercer clasificador	0	0	0	0	0	0	0	0	10	18	33	33
Pruebas	0	0	0	0	0	0	0	0	0	0	5	5
Documentación *(Proceso paralelo al resto)*	3	11	19	29	39	57	75	83	88	98	108	113

Trabajado acumulado	Días 121 al 130	Días 131 al 140	Días 141 al 150	Días 151 al 160	Días 161 al 170	Días 171 al 180	Días 181 al 190	Días 191 al 200	Días 201 al 210	Días 211 al 220	Total
Planificación	7,75	8	8,25	8,5	8,75	9	9,25	9,5	9,75	10	10
Preparación del entorno	11	11	11	11	11	11	11	11	11	11	11
Estudio del entorno y del estado del arte	98	98	98	98	98	98	98	98	98	98	98
Primer clasificador	29	29	29	31,5	37,5	37,5	37,5	37,5	37,5	37,5	37,5
Segundo clasificador	5	5	5	8	8	8	8	8	8	8	8
Tercer clasificador	33	33	33	35,5	36,5	36,5	36,5	36,5	36,5	36,5	36,5
Pruebas	6,5	11	18,5	23,5	23,5	23,5	23,5	23,5	25,5	25,5	25,5
Documentación *(Proceso paralelo al resto)*	119	123	129	133	142,5	152,5	155	159	168	175	175

Tabla 7.7: *Tabla de trabajo total realizado en horas, en intervalos de diez días.*

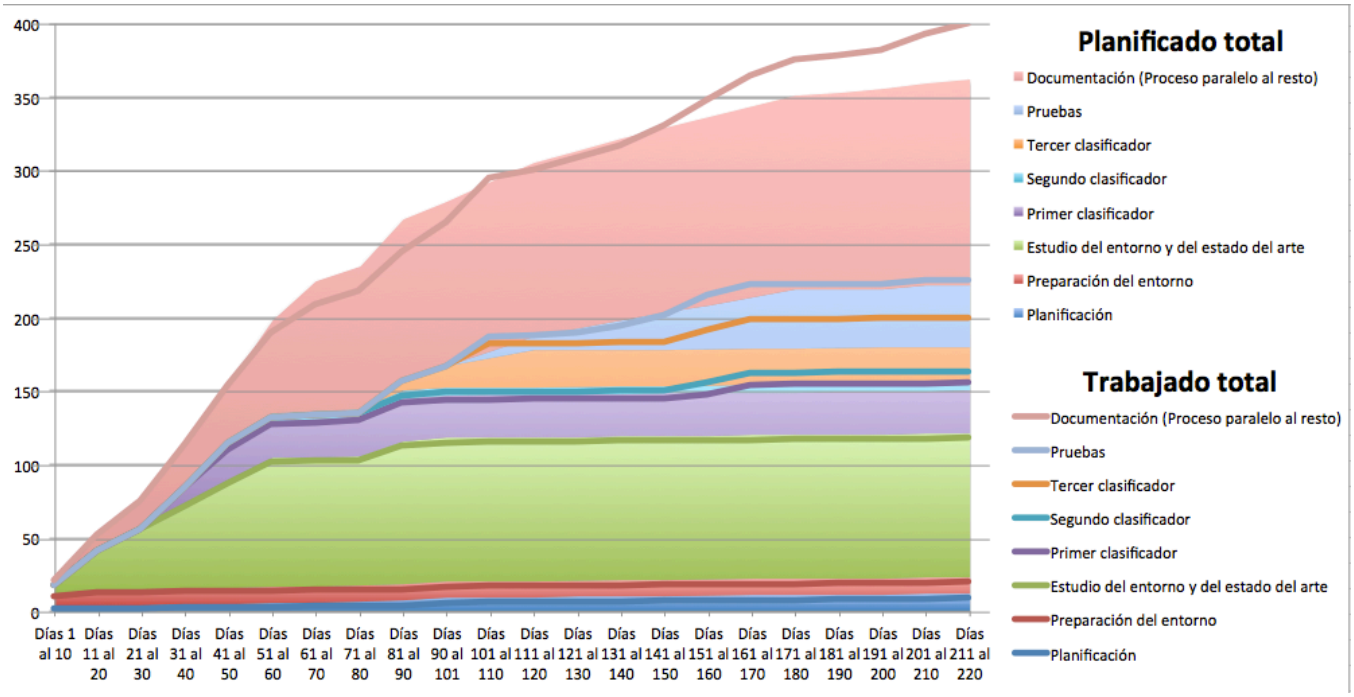


Gráfico 7.1: *Comparativa entre el trabajo estimado acumulado y el trabajo total realizado.*

7.5. Detalles sobre el proceso

A continuación tienen lugar otros detalles del proceso de planificación, como las dificultades encontradas durante el desarrollo del proyecto para cumplir la planificación inicial, o el uso de conocimientos previos, adquiridos en otras asignaturas, que ayudó a agilizar algunas tareas de estudio.

i. Dificultades encontradas

A lo largo del desarrollo del proyecto, han surgido varios inconvenientes que hicieron cambiar la predicción inicial de horas prevista (imprevistos, retrasos en la realización de tareas, fallos en desarrollo…). Sin embargo, todo ello pudo solucionarse cambiando la organización de horas por días, sin tener porqué variar el calendario propuesto en el diagrama de Gantt anterior.

Por ejemplo, los tiempos de estudio de cara al estado del arte llevaron más de lo debido. Esto se debe a que en la planificación inicial no se tuvo en cuenta la complejidad de los textos que debían estudiarse. No sólo por el idioma, si no por la profundidad teórica de los conceptos abarcados.

En la etapa de implementación se mantuvieron los tiempos estimados. Fue la validación la que llevó algo más de tiempo, por inconvenientes surgidos, como descubrir que las pruebas de OISVM fueron desarrolladas en la literatura para bloques de datos arbitrarios. Por ello se tuvieron que hacer cantidad de pruebas y deducciones, con el fin de validar el correcto funcionamiento del clasificador implementado. En esta etapa, un fallo de programación hizo volver a repetir todo el proceso de validación (tanto de OISVM como de IGNGSVM, por su estructura de bloques de datos) una vez ya nos encontrábamos en fase de estudio. El estudio comparativo no se excedió del tiempo planificado para las pruebas.

Por último, no debe pasar desapercibida la desviación que sufre la etapa de documentación en el proceso de planificación. Debido a las correcciones que sufrió el trabajo, la fase de documentación tardó 35 horas más de las previstas inicialmente. Esto se debe a que dichas correcciones se efectuaron meses después de escribir los apartados, lo que supuso un reestudio de la materia del estado del arte de la bibliografía.

Por otro lado, la ampliación del plazo del proyecto hizo re-planificar las tareas que quedaban por realizar en septiembre (**ver sección 7.2**). La re-planificación supuso un aumento de 23 horas en el tiempo planificado para la realización trabajo. Se aprecia en el **gráfico 7.1** cómo dicho aumento de horas planificadas no es tiempo suficiente para su finalización, llegando a necesitar 40 horas más en la práctica.

ii. Aplicación de conocimientos de otras asignaturas

Durante la realización de este proyecto, existieron factores que ayudaron a agilizar algunas de las tareas. Tal es el caso las etapas de estudio. Conocimientos previos como el funcionamiento, a grandes rasgos, de las máquinas de soporte vectorial, dadas en *Aprendizaje Automático,* el manejo en la misma asignatura de clasificadores online, y el conocimiento de las redes de neuronas gracias a la asignatura *Redes de Neuronas Artificiales*, ayudaron a entender más fácilmente los conceptos estudiados en este trabajo.

Todo conocimiento relacionado a la asignaturas de *Inteligencia Artificial*, como el aprendizaje basado en instancias (*Aprendizaje Automático*), ayudó en la realización del trabajo. No sólo para agilizar el entendimiento en el estudio, si no también como material de consulta en la etapa de documentación (Daniel Borrajo Millán, 2006).

En cuanto a trabajos relacionados, en el último curso desarrollé con dos compañeros un predictor de quinielas para la asignatura *Inteligencia Artificial en las Organizaciones*. Dicho predictor funcionaba mediante un conversor desarrollado en JAVA, que transformaba un fichero con los resultados de los partidos previos en instancias de SNNS (*Stuttgart Neural Network Simulator*). En SNNS se pasaba dicho fichero de salida (conjunto de datos entrenados), y sólo había que definir una red para poder predecir los próximos encuentros. Debido que sólo existía un bloque de ficheros de la base de datos (es decir, no entraban más con el tiempo), dicha técnica era batch. Obviamente el predictor no funcionaba perfectamente. No podía hacerlo cuando, no sólo las bases de datos son incompletas en cuanto a jugadores, forma física, estado emocional, meteorología, actitud del arbitro, etc., sino por que no existía información de contexto (*ver problema del Concept Drift en Estado del Arte*). Tras la realización de este trabajo, he visto como una posible mejora que podríamos de tener en cuenta es la utilización de una red incremental, que varíe según entren las nuevas instancias, jornada a jornada. Así podríamos almacenar datos de contexto, como lesiones, rachas, juego del equipo en las diferentes estaciones, etc., de una forma más sencilla que aplicando más datos de entrada (algo que podría ocasionar ruido en los datos de encuentro).

8. PRESUPUESTO

En el apartado anterior se detalla la organización y el tiempo dedicado en días y horas a cada tarea. A continuación se dividen las partes a presupuestar, se estima el coste de cada una, y por último se refleja el presupuesto final.

Coste del personal

La **figura 7.5** del apartado de planificación, muestra el total de horas estimadas en el proyecto para cada actividad. A continuación se divide el coste de las distintas tareas, con un salario de 15 €/hora para el ingeniero (2400 €/mes trabajando 160 horas).

Fases	Horas trabajadas	Coste
Planificación	6	90,00 €
Preparación del entorno	20	300,00 €
Estudio de entorno y estado del arte	64	960,00 €
Primer clasificador	32	480,00 €
Segundo clasificador	37	555,00 €
Tercer clasificador	40	600,00 €
Estudio comparativo	40	600,00 €
Documentación	100	1.500,00 €
TOTAL TRABAJADO	**339**	**5.085,00 €**

Tabla 8.1: *Coste de personal dividido en coste por tareas*

Coste de los equipos

Artículo	Coste	Vida útil	Coste mensual	Meses de uso
Macbook Pro 13" i5	1100,00 €	6 años	15,30 €	3
TOTAL COSTE DE EQUIPO				**45,90 €**

Tabla 8.2: *Coste del equipo en el periodo de utilización*

Coste software y herramientas

Software	Coste
Mac OSX	0 € (Incluido en la compra del equipo)
Eclipse	0 €
WEKA	0 €
MOA	0 €
TOTAL COSTE SOFTWARE	**0 €**

Tabla 8.3: *Coste del software necesario para el proyecto*

Coste material fungible

Concepto	Coste	Unds.	Coste total
CDs	1,00 €	5	5,00 €
Papel DIN A4 x500	3,99 €	1	3,99 €
Encuadernación	10,00 €	1	10,00 €
TOTAL MATERIAL FUNGIBLE			**18,99 €**

Tabla 8.4: *Coste del material fungible utilizado en el proyecto*

Total de costes directos

Software	Coste
Personal	5.085,00 €
Equipos	45,90 €
Software	0,00 €
Material Fungible	18,99 €
TOTAL DIRECTO	**5.149,89 €**

Tabla 8.5: *Costes directos del proyecto*

Presupuesto final

Se consideran además, un 8% de gastos indirectos (internet y luz en meses de verano), y se espera obtener un margen de 9% de beneficios.

Concepto	Coste
COSTES DIRECTOS	5.149,89 €
COSTES INDIRECTOS	411,99 €
COSTES TOTALES	5.591,88 €
BENEFICIO	503,27 €
TOTAL PROYECTO	**6095,15 €**

Tabla 8.6: *Presupuesto del proyecto*

PRECIO TOTAL: 6.095,15€ (IVA no incluido).

Seis mil noventa y cinco euros y quince céntimos (IVA no incluido).

9. CONCLUSIONES Y LÍNEAS FUTURAS

En este apartado, se exponen conclusiones que resumen las ya presentadas al analizar los datos en el **apartado 6**. Además, se añaden otras de carácter más cualitativo y general, así como posibles mejoras y líneas futuras, de forma que podamos orientar posibles investigaciones a partir de este proyecto.

En este trabajo se ha evaluado el funcionamiento de tres algoritmos de clasificación incremental basados en SVMs. Para ello se ha utilizado la herramienta de análisis incremental *MOA*, en la que ha habido que integrar dos algoritmos programados en este proyecto.

En la fase de estudio comparativo, se ha visto cómo *OISVM* e *IGNGSVM* consiguen reducir en gran medida los tiempos de ejecución de los conjuntos de datos con *LibSVM*. Sin embargo, vemos en los resultados como su estructura de partir en bloques los conjuntos hace difícil respetar los momentos de cambio en conjuntos no estacionarios y parecen funcionar mejor orientados a tratar con conjuntos de datos grandes de forma online.

Cada día crece más la necesidad de utilizar de técnicas que compriman y clasifiquen la cantidad de datos existentes. El tamaño de los conjuntos de datos manejados, hace imposible reunir y filtrar, en una sola iteración, el número de ejemplos requeridos en el proceso de aprendizaje. Cosa que hace necesario, tanto operar en línea, de forma continua, como procesar cada ejemplo a tiempo real.

Sin embargo, a pesar de los grandes beneficios del aprendizaje incremental, el diseño de un algoritmo que aprenda a lo largo del tiempo supone muchos retos en cuanto a fijar los límites de memoria, así como de asignar el peso idóneo en la clasificación a los nuevos y antiguos prototipos. Al programar un algoritmo tipo batch sólo es necesario preocuparse de clasificar correctamente los datos de una iteración, mientras que en un algoritmo incremental dichos datos pueden replicarse a lo largo del tiempo, en distintas iteraciones. Habrá que detectar cada momento de cambio de contexto o analizar si los datos que tratamos son estacionarios para garantizar una buena precisión de acierto en el menor tiempo de respuesta posible.

Por otra parte, la librería *LibSVM* utilizada no está pensada para su incrementalización, pese a haber sido utilizada por los autores de *OISVM*. Su limitación se hace notable en cuanto a complejidad temporal, lo que hace que no sea posible clasificar de inmediato conjuntos cuyo paisaje de cambio es muy abrupto, o tienen un gran coste en memoria.

Por ello, en *IGNGSVM* seguimos la misma estructura de división en bloques el conjunto de datos. Cosa que nos hace tener que prestar especial atención en clasificar siempre justo después de crear la SVM, pues la entrada de nuevas instancias va desactualizando el modelo hasta el fin de la iteración; donde se genera una nueva SVM con los datos de ese bloque y los SVs anteriores.

En definitiva, la librería *LibSVM* nos hace tener que olvidar la idea preconcebida de algoritmo incremental como clasificador en cualquier momento de tiempo, y nos obliga a

hacerlo en tandas si queremos lograr una clasificación que no consuma demasiado tiempo.

A continuación, se muestran algunas propuestas para buscar mejoras en cuanto a los resultados obtenidos en este proyecto, y facilitar una posible continuación de esta investigación.

Si nos referimos a la precisión obtenida, en primer lugar, *OISVM* muestra una buena reducción en la tasa de error respecto a *LibSVM* en *SEA*; aunque el resto de conjuntos tiene una pérdida de acierto considerable.

- Sería conveniente seguir haciendo pruebas con otros conjuntos, o procesar de distinta forma estos, buscando una mejoría para la precisión obtenida.

- También convendría revisar a fondo la integración de *ILVQ* utilizada con *OISVM*, por si no procesase correctamente los ejemplos con ruido o por alguna razón los parámetros usados no funcionasen de la forma que sí lo hacen en la literatura.

Respecto a *IGNGSVM*, este muestra un resultado esperado al empeorar cuando reducimos el tamaño de bloque de un conjunto no suficientemente grande (*GNG* suele usarse con conjuntos de millones de datos). Tampoco hay que olvidar el fallo de representación de *GNG* que surge en nuestra implementación de *IGNGSVM* cuando el número de ejemplos de cada clase no está nivelado; que es debido a que el criterio de parada en de cada sub-bloque TSi sea un número fijo de prototipos generados.

- Para arreglar el fallo cuando las clases no están niveladas, proponemos cambiar la implementación de forma que el criterio de parada de cada sub-bloque TSi sea un porcentaje de prototipos generados respecto al número de prototipos contenidos en dicho sub-bloque; es decir, del número de ejemplos que sean de esa clase i. De esta forma no generaríamos más prototipos de una clase de los que ya contiene, si no un porcentaje predefinido por el usuario de los que ya contenga.

- Aparte, una propuesta sería probar su funcionamiento con conjuntos de tamaño mayor a *Shuttle*, en los que se pudiese explotar el potencial de *GNG* para representar las topologías en conjuntos reducidos.

Como último punto, se ve como un algoritmo incremental puro como *SGD* puede ser muy útil en conjuntos de la naturaleza de *Elec2*. Sin embargo, en los otros tres conjuntos nos surge la duda de si los malos resultados se deben a la ausencia de memoria de *SGD*, o a que los conjuntos no pueden ser correctamente representados en un modelo lineal. En el caso de la representación, sería interesante buscar un modelo de *SGD* que permitiese trabajar con kernels no lineales.

10. OPINIÓN PERSONAL DEL TRABAJO Y AGRADECIMIENTOS

Tras la realización de este trabajo, puedo decir que he descubierto que la investigación es una de mis vocaciones. Llegue hasta este proyecto buscando algo que fuese rápido de hacer, y que fuese de la especialidad de computación (que elegí en su día por considerarla la rama más interesante), y encontré en él una de las actividades que mayor satisfacción personal me ha aportado.

A modo de valoración personal, en este trabajo he entendido la importancia de los algoritmos de clasificación incrementales en el uso de procesar los grandes conjuntos de datos que existen en la actualidad.

Hasta hacer este proyecto no me había percatado de las dimensiones en las que se aplican las tareas de minería de datos, y tenía una versión más simple de lo que era un clasificador, desconociendo las técnicas de aprendizaje incremental. Este trabajo me ha llevado a entender algunos de los razonamientos tomados para el diseño de los algoritmos estudiados .

10.1 Realización y planificación del trabajo

La realización de este proyecto fue un reto personal desde un principio. Hablé con varios profesores sobre mi deseo de realizar un proyecto basado en aprendizaje automático cuya longitud no fuese muy grande, pues quería exponerlo tres meses después. Todo ello me llevo a dar con Alejandro Cervantes, que ofertaba un proyecto de integración de nuevas técnicas de aprendizaje incremental en *MOA*.

En un principio, no existía un límite fijado sobre el número de algoritmos a integrar. Tras hacerlo con el primero, *OISVM*, y realizar un estudio sobre *SGD*, vimos como el tiempo se nos venía encima y establecimos dicho límite a tres algoritmos. En ese momento hubo que hacer una profunda búsqueda en la literatura pertinente a métodos incrementales para buscar el tercer algoritmo con el que realizar el estudio.

Tras la negativa de exponer en convocatoria extraordinaria este proyecto, pudimos prolongarlo cuatro meses más. Ello nos llevó a proponer como tercer clasificador, un nuevo algoritmo incremental, el cual llamamos *IGNGSVM* (debido al tiempo consumido, de otra forma hubiese habido que escoger algún algoritmo ya probado en la literatura). A partir de aquí, lo que comenzó siendo un reto en cuanto a fechas, paso a ser un reto de conocimientos a adquirir.

A partir de la planificación, superó en horas la estimación inicial, teniendo especial relevancia en ese aspecto el retraso del plazo de presentación, que nos permitió mejorar y elevar la complejidad del proyecto. El presupuesto inicial finalmente ha sido bastante barato debido a la ampliación del trabajo. Aunque, a éste, podría ahora descontársele el precio de los CDs e impresión del documento que, por la nueva normativa de entrega mediante Aulaglobal, no son necesarios.

10.2 Conocimientos adquiridos y mejorados

Desde el comienzo, la realización de este proyecto supuso entender muchos conceptos teóricos no trabajados, o vistos sólo por fuera, a lo largo de la carrera. Además de entender el funcionamiento incremental de las clases de la herramienta *MOA* y de cada uno de los algoritmos estudiados, tuve que entender conceptos teóricos como el kernel de una máquina de soporte vectorial, lo que es un cambio de contexto en un entorno no estacionario, etc.

Este trabajo me ayudó también a mejorar partes trabajadas en las distintas asignaturas relacionadas a la Inteligencia Artificial. Algunos puntos son los siguientes:

- Entender los conjuntos de datos como una serie de puntos (puntos con clase definida en el caso de aprendizaje supervisado) en un espacio, cuya dimensión depende del número de atributos que tengan los vectores o ejemplos de entrenamiento.
- Entender la necesidad de un kernel como herramienta para trasladar los ejemplos de cualquier dimensión a una en la que sea posible definir un plano separador (en el caso de SVMs) de clases.

- Mejorar mi manejo de la herramienta *WEKA* a partir de '.sh' ejecutados desde terminal, aprender el uso de herramientas propias de filtrado (NumericToNominal, Randomize, …) y de optimización de parámetros (CVParameterSelection), y aprender el uso de *MOA*.

- Ver como funciona un algoritmo de clasificación de *WEKA* desde dentro e interactuar con él (*LibSVM*). Programar tres algoritmos para *MOA* (*OISVM, IGNGSVM* y *GNG*).

- Adquirir conocimiento acerca de redactar informes de investigación, y de cómo buscar trabajos de investigación y saber citarlos en el documento.

- Entender las diferencias entre los distintos algoritmos de clasificación abarcados, y establecer un estudio acerca de los resultados alcanzados con cuatro conjuntos de datos de distintas características.

10.3 Agradecimientos

En primer lugar debo reconocer todo el trabajo realizado por Alejandro Cervantes como tutor de este trabajo. Quiero alabar su preocupación y consejos, además de sus procedimientos de trabajo.

Dos personas que no han tomado parte en la realización de este proyecto, pero si han sido parte activa antes o durante el proceso, y merecen un reconocimiento, son Pedro Isasi y Belén Ruiz.

El primero, después de varias tutorías como alumno y varias clases soportándome, fue a quien acudí para hacer este trabajo y me encomendó a Alejandro.

La segunda, por su especial atención y preocupación intentando gestionar la forma de exponer este proyecto en una posible convocatoria extraordinaria, aunque finalmente no pudiese ser.

Por último, agradecer a mi familia y a Alicia, por ser el mejor apoyo, y a mis tres compañeros de piso; Alejandro, Alexandre y Daniel; por aguantarme durante todo este proceso.

11. BIBLIOGRAFÍA Y REFERENCIAS

A continuación, se ofrecen las referencias bibliográficas utilizadas a lo largo de la realización del proyecto o citadas en el documento por su relación o relevancia con el contenido del trabajo.

[1] A.Tsymbal. (2004). *The problem of concept drift: Definition and related work.* Tech. Rep., Trinity College, Department of Computer Science, Dublin.
[2] Abril, L. G. (2003). *Modelos de Clasificación basados en Máquinas de Vectores Soporte.* Universidad de Sevilla .
[3] Administration U.S. National Oceanic and Atmospheric. (s.f.). *ncdc.noaa.gov*. Obtenido de Federal Climate Complex Global Surface Summary of Day Data: FTP: ftp.ncdc.noaa.gov/pub/data/gsod
[4] B. Jin, Y. Q. (2006). Classifying Very Large Data Setswith Minimun Enclosing all Based Support Vector Machine. *Proc. IEEE Int. Conf. on Fuzzy Systems.* Vancouver.
[5] Bordes A., B. L. (2009). Careful Quasi-Newton Stochastic Gradient Descent. *Journal of Machine Learning Research* , 1737-1754.
[6] Boris T. Polyak, A. B. (1992). Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization* (30), 838-855.
[7]Bottou, L. (2010). Large-Scale Machine Learning with Stochastic Gradient Descent. En Y. L. Saporta (Ed.), *Proceedings of the 19th International Conference on Computacional Statistics* (págs. 177-187). Paris: Springer.
[8] Bottou, L. (1998). Online Algorithms and Stochastic Approximations. (D. Saad, Ed.) *Online Learning and Neural Networks* .
Bottou, L. (27 de 04 de 2012). *Stochastic Gradient Descent*, 2. Obtenido de http://leon.bottou.org/projects/sgd
[9] C. W. Hsu, C. J. (1998). A comparison of methods for multi-class support vector machines. En *IEEE Transaction on Neural Networks* (Vol. 13, págs. 415-425).
[10] C.Giraud-Carrier. *A note on the utility of incremental learning.*
Catlett, J. (2006). *Basser Department of Computer Science.* University of Sydney . Sydney: N.S.W. .
[11] Daniel Borrajo Millán, J. G. (2006). *Aprendizaje Automático.* Madrid: Sanz y Torres.
[12] David W. Aha, D. K. (1991). Instance-Based Learning Algorithms. En *Machine Learning* (págs. 36-66). Boston: Kluwer Academic Publishers.
[13] Dennis J.E., J. S. (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall.
[14] E. Alpaydin, C. K. (1998). *Cascading Classifiers.* Bogazici University, Department of Computer Engineering . Istanbul: Kybernetika.
[15] Francisco J. Ferrer, J. S. (2005). Aprendizaje Incremental de Reglas en Data Streams. *Actas del III Taller Nacional de Minería de Datos y Aprendizaje* , (págs. 261-270).
[16] Fritzke, B. (1995). A growing neural gas network learns topologies. En D. D. G. Tesauro, *Advances in neural information processing systems* (págs. 625-632). Cambridge: MIT Press.

[17] G.Widmer, M. (1996). Learning in presence of concept drift and hidden context. *Machine Learning* , 69-101.
[18] Gert Cauwenberghs, T. P. (2000). Incremental and Decremental Support Vector Machine Learning. En *NIPS* (págs. 409-415).
[19] Giraud-Carrier, C. (2000). A Note on the Utility of Incremental Learning. En *AI Communications* (Vol. 12, págs. 215-223).
[20] H. Yu, J. Y. (2003). Classifying large data sets using SVM with hierarchical clusters. *Proceedings of the SIGKDD 2003*, (págs. 306-315). Washington DC.
[21] Harries, M. (1999). *Splice-2 comparative evaluation: Electricity pricing.* Technical report, The University of South Wales.
[22] Hastie, T. R. (2009). he Elements of Statistical Learning: Data Mining, Inference, and Prediction.
[23] J. L. Balcázar, Y. D. (2001). A random sampling technique for training support vector machines. *Proc 13th Int. Conf. Algorithmic Learning Theory.* Washington DC.
[24] Jesús S. Aguilar, F. J. (2005). *Minería de Data Streams: Conceptos y Principales Técnicas.*
[25] Juan D. Velásquez, Y. O. (2010). Predicción de series temporales usando máquinas de soporte. *Revista chilena de ingeniería , 18* (1), 64-75.
[26] Jun Zheng, H. Y. (2010). An Online Incremental Learning Support Vector Machine for Large-scale Data . (S.-V. B. 2010, Ed.) *ICANN 2010, Part II, LNCS 6353* , 76-81.
[27] Kohonen, T. (1990). Improved versions of learning vector quantization. 545-550.
[28] Laskov, P. G. (2006). Incremental support vector learning: Analysis implementation and applications. *Journal of Machine Learning Research 7* , 1909–1936.
[29] LeCun, L. B. (2004). Large Scale Online Learning. (L. S. Sebastian Thrun, Ed.) *Advances in Neural Information Processing Systems* (16).
[30] M. Baena-Garcia, J. d.-A.-B. (2006). Early drift detection method. En *ECML PKDD 2006 Workshop on Knowledge Discovery from Data Streams* (págs. 77-86).
[31] Moulines, F. B. (2011). Non-Asymptotic Analysis of Stochastic Approximation Algorithms for Machine Learning. *Advances in Neural Information Processing Systems* .
[32] Nadeem Ahmed Syed, H. L. (s.f.). Handling Concept Drifts in Incremental Learning with Support Vector Machines. *Program for Research in Intelligent Systems*
[33] Nadeem Ahmed Syed, H. L. *Handling Concept Drifts in Incremental Learning with Support Vectors Machines.* National University of Singapure, School of Computing.
[34] Olivier Bousquet, L. B. (2008). The Tradeoffs of Large Scale Learning. *Advances in Neural Information Processing Systems* (20).
[35] Ondrej Linda, M. M. (2009). GNG-SVM Framework - Classifying Large Datasets with Support Vector Machines Using Neural Gas. *Neural Networks, 2009. IJCNN 2009. International Joint Conference.*
[36] Pavel Laskov, C. G.-R. (2006). Incremental support vector learning: Analysis, implementation and applications. (E. P. Kristin P. Bennett, Ed.) *Journal of Machine Learning Research , 7*, 1909–1936 .
[37] Pingdom. (12 de 01 de 2011). *Royal Pingdom*. Recuperado el 31 de 08 de 2012, de http://royal.pingdom.com/2011/01/12/internet-2010-in-numbers/
[38] Rumelhart D.E., H. G. (1986). Learning internal representations by error propagation. En B. Books (Ed.), *Parallel distributed processing: Explorations in the microstructure of cognition,* (Vol. 1, págs. 318-362).

[39] Ruppert, D. (1988). *Efficient estimations from a slowly convergent Robbins-Monro process.* Técnical Report 781, Cornel University Operations Research and Industrial Engineering.
[40] S. Tong, D. K. (2000). Support vector machine active learning with applications to text classification. *In Proc. 17th Int. Conf. Machine Learning.* Stanford.
[41] S. W. Xiong, X. X. (2005). Support vector machines based on substractive clustering. *Proc 14th Int. Conf. Machine Learning and Cybernetics.* Guangzhou, China.
[42] S.M. Lee, S. R. (2010). Sequencial dynamic classification using latent variable models. En *Comput. J.* (Vol. 53, págs. 1415-1429).
[43] Shen F, H. O. (2008). A fast nearest neighbor classifier based on a self-organizing incremental neural network. *Neural Networks* , 1537-1547.
[44] Stefan, R. (2001). Incremental Learning with Support Vector Machines . *First IEEE International Conference on Data Mining .* ICDM.
[45] Syed, N. L. (1999). Incremental learning with support vector machines. *Workshop on Support Vector Machines at the International Joint Conference on Articial Intelligence .* Stockholm, Sweden: IJCAI.
[46] T.Poggio, G. C. (2000). Incremental and decremental Support Vector Machine Learning. *NIPS* , 409-415.
[47] Villegas M, P. R. (2008). *Simultaneous learning of a discriminative projection and prototypes for nearest-neighbor classification.*
[48] W.N. Street, a. Y. (2001). A streaming ensemble algorithm (SEA) for large-scale classification.
[49] Wasan, M. (1969). *Stochastic Approximation.* Cambridge: Cambridge University Press.
[50] Xu, W. (2010). *Towards Optimal One Pass Large Scale Learning with Averaged Stochastic Gradient Descent.* Technical report.
[51] Ye Xu, F. S. (2010). An incremental learning vector quantization algorithm. *Neural Comput & Applic* .
[52] Ye Xu, S. F. (2009). *An Online Incremental Learning Vector Quantization.* Nanjing University, Tokio Institute of Technology. Springer-Verlag.

ANEXOS

A GUÍA DE INSTALACIÓN Y EJECUCIÓN DEL SOFTWARE

A continuación se proporciona una referencia a los repositorios donde pueden consultarse las guías de instalación, ejecución y uso de *WEKA* y *MOA*.

- La guía de usuario de *WEKA* se encuentra disponible en la web de Waikato, donde pueden obtenerse varias versiones, así como acceso a la wiki para desarrollo: http://www.cs.waikato.ac.nz/ml/weka/documentation.html

- La guía de *MOA* también se puede conseguir fácilmente desde su web oficial, en la parte de documentación; también se ofrece acceso al API de *MOA*: http://moa.cms.waikato.ac.nz/downloads/#documentation

B INFORMACIÓN DE SALIDA

Los ficheros de salida de las fases de estudio de los algoritmos se encuentran en el archivo ***./Estudio Experimental/Resultados.zip*** .

La información de salida, además de trazas que muestran la reducción del bloque, es la que suelen mostrar las salidas en *MOA* (**ver guías en el Anexo A**).

C SOFTWARE DESARROLLADO

CÓDIGO FUENTE

El código fuente de los tres algoritmos implementados se encuentra en la carpeta ***./Código fuente/**** .

Para la ejecución de *OISVM*, se necesita de un algoritmo de *ILVQ*, que por no haberse desarrollado en este trabajo y ser de otro compañero, no se encuentra anexo en la entrega.

INSTALACIÓN DE LOS CLASIFICADORES

La instalación de los algoritmos integrados en este trabajo se realiza en paralelo a la instalación de MOA y WEKA. Para hacerlo, descomprimimos los algoritmos adjuntos en la carpeta cuya ruta se define en el apartado anterior, y se configura el proyecto de forma que los *classpath* permitan encontrar las librerías: *moa.jar, weka.jar, libsvm.jar* y los .class de los .java de este trabajo compilados.

Para ver un ejemplo de esto, se brindan los scripts .sh utilizados para la ejecución de las pruebas del estudio en el archivo ***./Estudio Experimental/Scripts.zip***

D REGISTROS DE ACEPTACIÓN DE LOS ALGORITMOS PROGRAMADOS

Como se ha descrito en el apartado cuatro del documento, la fase de aceptación está compuesta por una serie de fichas de aceptación de los algoritmos propuestos. Dichas fichas de presentación y aprobación constan del nombre del algoritmo, de sus referencias bibliográficas, y de los conjuntos de datos y parámetros seleccionados en la fase de validación.

SOLICITUDES DE APROBACIÓN

<table>
<tr><th colspan="4">SOLICITUD DE APROBACIÓN</th></tr>
<tr><td>Cliente</td><td colspan="2">Alejandro Cervantes Rovira</td><td>Solicitud Nº</td></tr>
<tr><td>Proyecto</td><td colspan="2">Integración a MOA de nuevas técnicas de aprendizaje</td><td rowspan="3">1</td></tr>
<tr><td colspan="2">Código Proyecto</td><td>Fecha Solicitud</td></tr>
<tr><td colspan="2">TFG80038</td><td>20/06/12</td></tr>
<tr><td>Nombre del algoritmo</td><td colspan="3">Online Incremental Support Vector Machine</td></tr>
<tr><td colspan="2">Bibliografía consultada para su implementación:

1. (Jun Zheng, 2010)
2. (Ye Xu S. F., 2009)
3. (Ye Xu F. S., 2010)</td><td colspan="2">Conjuntos de datos y parámetros utilizados en la validación

• Optdigits (Coste = 8, Gamma = 0.0008)
• Usps (Coste = 4, Gamma = 0.0135)
• Shuttle (Coste = 10, Gamma = 0.05)

Las pruebas se han realizado con un número de bloques igual a: 1,2,3,5,7,9,10</td></tr>
<tr><td>Autor de la solicitud</td><td>Andrés León Suárez Cetrulo</td><td colspan="2">Firma</td></tr>
</table>

SOLICITUD DE APROBACIÓN

Cliente	Alejandro Cervantes Rovira	***Solicitud Nº***
Proyecto	Integración a MOA de nuevas técnicas de aprendizaje	2
Código Proyecto	***Fecha Solicitud***	
TFG80038	19/09/12	
Nombre del algoritmo	Incremental GNGSVM	

Bibliografía consultada para su implementación:	***Conjuntos de datos y parámetros utilizados en la validación***
1. (Fritzke, 1995) 2. (Ondrej Linda, 2009) 3. (Jun Zheng, 2010)	• Conjunto Shuttle modificado con dos clases, y evaluado con kernel polinómico (de forma similar a (Ondrej Linda, 2009)): o Se utiliza LibSVM en lugar de SMO (Coste = 8, Gamma = 0.8845) o Pues la validación es de GNGSVM, la prueba se realiza sobre N_TS = 1

		Firma
Autor de la solicitud	Andrés León Suárez Cetrulo	

INFORMES DE APROBACIÓN

INFORME DE APROBACIÓN

Cliente	Alejandro Cervantes Rovira	***Solicitud Nº***
Proyecto	Integración a MOA de nuevas técnicas de aprendizaje	1
Código Proyecto	***Fecha de Certificación***	
TFG80038	14/09/12	

Pruebas Realizada

	LIBSVM		OISVM (Batch)		OISVM (10 blocks)	
	Paper	Nuestros	Paper	Nuestros	Paper	Nuestros
Optdigits	98.37	98.44	97.33	96.22	97.33	93.63
Shuttle	99.83	98.38	99.68	92.57	99.77	91.40
Usps	99.53	99.83	98.96	95.76	99.04	93.36

Solicitud de aprobación por	***Aprobación del algoritmo por***
Andrés L. Suárez Cetrulo	Alejandro Cervantes Rovira
Firma	*Firma* *Aceptación de validación*

INFORME DE APROBACIÓN

Cliente	Alejandro Cervantes Rovira	***Solicitud Nº***
Proyecto	Integración a MOA de nuevas técnicas de aprendizaje	2

Código Proyecto	***Fecha de Certificación***
TFG80038	21/09/12

Pruebas Realizada

	LIBSVM		**GNGSVM (Batch)**	
	Paper	Nuestros	Paper	Nuestros
Shuttle	96.57	97.55	91.34	93.66

Solicitud de aprobación por	***Aprobación del algoritmo por***
Andrés L. Suárez Cetrulo	Alejandro Cervantes Rovira
Firma	*Firma* *Aceptación de validación*