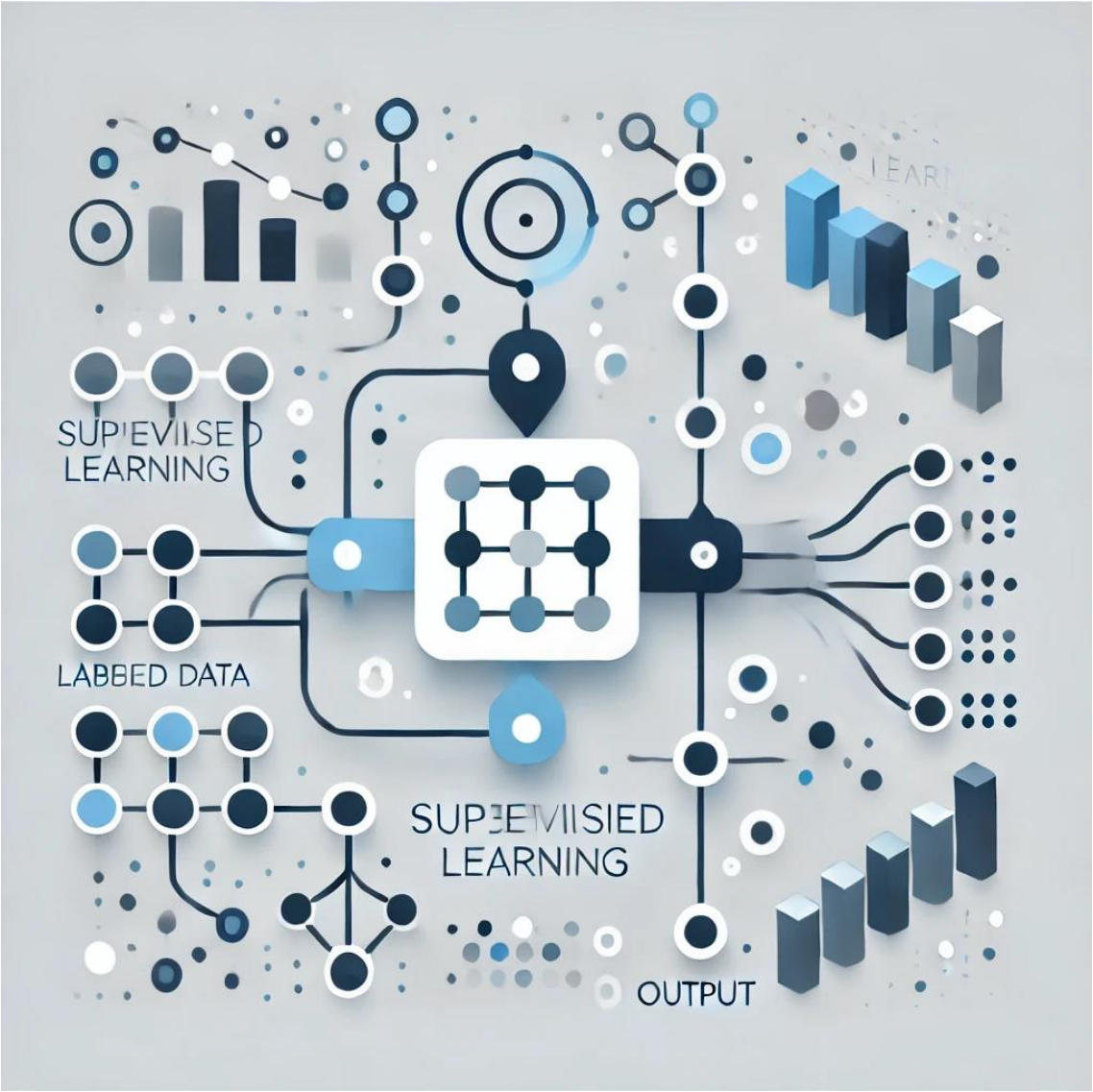
Aprendizaje supervisado

Aprendizaje Automático I



(AI generated)

UNIVERSIDAD POLITÉCNICA DE MADRID
ETSI SISTEMAS INFORMÁTICOS

Juan Moreno Segura

Alonso Ruiz Palomo

Índice

Preprocesado (Pág. 3)

- **Tratamiento de datos faltantes**
- **One-Hot y normalización**
- **Análisis de datos desbalanceados**
- **Evaluación exhaustiva de métodos de selección de características.**
- **Separación en entrenamiento y test**

Entrenamiento (Pág. 5)

- **Modelos Clásicos**
- **Árboles de Decisión**
- **Aprendizaje basado en Instancias**
- **Máquinas de Soporte Vectorial**
- **Redes Neuronales (usando capas fully connected y dropout)**

Best-model (Pág. 9)

Parte 1. Preprocesado

Durante esta tarea encontramos varios problemas, siendo la primera de ellas la dificultad para cargar el conjunto de datos, al ser tan grande era casi imposible de alojar en memoria y los tiempos de carga eran ridículos, era demasiado difícil trabajar; la solución que propusimos fue coger una muestra aleatoria de este (sin cargarle previamente en memoria) para poder ejecutar nuestros futuros algoritmos sin temer una pérdida de información. Unimos el dataframe a sus etiquetas para tener la información estructurada desde un principio y no esperar a realizar posibles cambios que dificulten este proceso en un futuro.

1. Tratamiento de datos faltantes:

El primer vistazo al mapa de calor de valores faltantes indica que hay muchas columnas con información casi nula, por no faltar filas vacías e inútiles. Para paliarlo, buscamos las filas y columnas con un ínfimo porcentaje de información (o en este caso, un alto porcentaje de información faltante) y las eliminamos del dataframe. Tras esto el mapa sigue mostrando muchas entradas faltantes, pero por miedo a perder información, sintetizamos lo que faltaba con el método fillna y la moda de cada columna. Finalmente, en nuestros datos no había un solo valor faltante.

2. One-Hot y normalización:

Comenzamos visualizando los tipos de datos alojados en el dataframe, en el que vemos que predominan los valores en coma flotante y objetos. Comenzando por los objetos, vemos que tenemos una fecha, datos rechazados por muchos modelos, luego los convertimos a numérico añadiendo tres columnas de datos; año, mes y día. Los otros datos tipo objeto presentaban muy pocos valores únicos, lo que dio pie al siguiente paso, la conversión a one-hot y eliminación de los que tenían poca varianza con un método VarianceThreshold casero.

Con los objetos fuera del camino, quedan las muchas columnas numéricas, las cuales procesamos rápidamente con una normalización. Ante la decisión de si usar el método minmax o el estándar, nos decantamos por el último por su superior rendimiento en los sistemas creados después.

3. Análisis de datos desbalanceados:

De nuevo, tras prueba, error y razonamiento, decidimos quedarnos con un método de undersampling, y para añadir simplicidad al asunto, uno aleatorio (RandomUnderSampling), para no solo reducir el tamaño de los datos, sino igualar la cantidad de ejemplo, lo cual subió la precisión de nuestros modelos.

Ya tratado y preparado, pasamos el dataframe por el proceso de selección de características.

4. Evaluación exhaustiva de métodos de selección de características.

Aunque usamos los 3 métodos estudiados en la asignatura (y probamos todos en los modelos), nos decantamos por el filtro por las características de la tarea (disponíamos de una cantidad absurda de datos, lo que hacía del wrapper y método embebidos procesos muy lentos). El filtro nos devuelve puntuaciones entre 2500 y 0.5 (aproximadamente) y nos decantamos por tomar solamente las características que tuvieran más de 750 puntos (de nuevo tras comprobar que la precisión y f1-score de los modelos incrementaba al tomar esta decisión)

Una vez terminado, separamos el código antes descrito en funciones de preproceso y las ejecutamos sobre el dataframe entero con un enfoque parecido al inicial, lo único que en vez de tomar una muestra, tomaremos porciones de este para preprocesarlo. En primer lugar está la función preproceso (hecha de snippets de todo el código anterior), a la cual si le pasas columnas usará las indicadas en la carga del dataset (usado en el tercer paso); mientras que si la llamada carece de estas cargará el dataframe normal. Después tenemos una función con un filtro para obtener las columnas importantes. También probamos con el embebido y wrappers con árbol de decisión y RandomForest, quedando el filtro por encima de todos estos en rendimiento. Finalmente, hay un proceso que une todo; obteniendo las columnas de los fragmentos limpios, componiendo el dataset ya limpio y haciendo una agrupación por ID del usuario, pues el objetivo es estudiar si un usuario será moroso o no en cuestión de sus interacciones; de esto sacamos la media y variación de cada una de las 17 columnas empleadas. Al tomar la desviación, es posible que aparezca un NaN si los usuarios solo han hecho una transacción, por lo que rellenamos los valores con 0

5. Separación en entrenamiento y test

Por motivos de efectividad, se encuentran en el cuaderno de supervised. De forma homóloga a como cargamos el dataset al principio, tomamos los datos preprocesados y los unimos a su label por medio de su identificador, para tener los labels ordenados.

Lo que sigue son distintas separaciones por tamaño, pues existen métodos que tardan demasiado en ejecutar y para poder ver un resultado de forma más o menos efectiva, reducimos las entradas que están disponibles para estos.

En primer lugar, los datos recién cargados se separan en los datos de entrenamiento con sufijo _big, para indicar que no hay ninguna selección.

Tenemos los datos “tamaño mediano”, siendo un 40% de los datos normales y separados de igual forma en un 80-20%.

Para los que tardan más, tomamos un 50% de los anteriores y hacemos la misma separación que en los otros dos.

Parte 2. Aplicación de modelos de aprendizaje supervisado

A continuación, vamos a entrenar distintos modelos para ver cual predice mejor los labels.

1. Modelos Clásicos:

- **Modelo lineal:**

El modelo lineal es el más simple que hemos visto, intenta ajustar una línea recta para separar las clases. Aun así, nos ha sorprendido su gran valor f1 = 0.8833, muy alto para el modelo sencillo que es. Los datos que tenemos son complejos y difíciles de clasificar, por eso este modelo se nos queda corto. Hemos utilizado todo el dataset para entrenar, pues el modelo entrena muy rápido.

- **Modelo Polinómico:**

El modelo polinómico es un poco más complejo que el lineal, por lo que esperamos un mejor ajuste de los datos. En efecto, conseguimos un f1 = 0.9102, mayor que el anterior. Este modelo captura relaciones entre los datos más complejas, pero aun así sigue siendo demasiado básico para los datos que tenemos. Hemos entrenado con todo el dataset debido al rápido entrenamiento del modelo.

- **Modelo logístico:**

El modelo logístico aplica una función sigmoidal. En este caso lo podemos utilizar de manera correcta ya que la salida es una clasificación binaria (0 o 1). Por la complejidad de los datos vemos que no clasifica muy bien, f1 = 0.8977. También entrenamos con todos los datos por el rápido entrenamiento del modelo.

2. Árboles de Decisión:

No nos hemos molestado en probar con DecisionTreeClassifier o ExtraTreeClassifier debido a que tenemos un dataset grande y con el RandomForestClassifier capturaremos relaciones más complejas entre los datos, que es lo que nos interesa. Entrenando con el parámetro n_estimators = 90 (lo habitual es 100 pero con 90 nos sale mejor), conseguimos el mayor f1_score de todos los modelos: 0.9438. Además, el RandomForestClassifier tiene resistencia al sobreajuste por la naturaleza de los árboles individuales, de los cuales se saca el voto mayoritario para obtener el resultado de la clasificación final. Hemos entrenado con todos los datos, ya que este es el modelo que mejor clasifica, aunque es verdad que nos ha tardado más.

3. Aprendizaje basado en Instancias:

Utilizaremos el modelo KNearestNeighbors (KNN) que es un modelo que aprende basado en instancias (no aprende parámetros), fijándose en los datos que tiene más cercanos para hacer la

clasificación. Este modelo tiene 2 hiperparámetros interesantes que debemos de elegir, estos son n_neighbors y weights. Para ello, vamos a hacer un Grid Search con los siguientes valores:

n_neighbors tomará valores entre 3 y 15, dando saltos de 3: [3,6,9,12,15]

weights toma 2 valores: distance, que da más importancia en la decisión de clasificación a los vecinos más cercanos, y uniform que da igual importancia a n_neighbors elegidos.

Debido a que para hacer el Grid Search tenemos que entrenar y evaluar el conjunto de test para cada combinación posible de los hiperparámetros, esto tarda demasiado. Por ello no utilizamos todo el dataset, sino que nos quedamos con el dataset de tamaño moderado para no eternizarnos.

Como resultado, obtenemos que los mejores hiperparámetros son: n_neighbors = 15 puesto que, como es lógico, cuantos más vecinos nos fijemos para la clasificación, más precisa será esta, aunque es verdad que llega un punto que aumentando este hiperparámetro, no conseguimos mayor f1 score. Además, cuanto mayor sea este hiperparámetro, más tardará el modelo en entrenar porque se tiene que fijar en un número de vecinos mayor. Como hiperparámetro weights, vemos que obtenemos valores f1 parecidos con ambos valores, aunque es un poco mejor cuando este es distance, puesto que tiene sentido que los vecinos más cercanos influyan más en la decisión de clasificación.

Es importante recalcar que no utilizamos conjunto de validación debido a que el Grid Search utiliza validación cruzada dividiendo el conjunto de entrenamiento en 5 pliegues (cv=5). Así, cada combinación de hiperparámetros se entrena y valida en los diferentes subconjuntos de datos, obteniendo el promedio de las métricas de cada pliegue para evaluar el rendimiento.

Una vez tenemos los hiperparámetros elegidos, entrenamos otro KNN con estos, y todo el dataset. Obtenemos un f1 = 0.894796, un valor alto pero que, en comparación con otros modelos, tiene un valor parecido al modelo logístico. Por lo que este modelo basado en instancias no va a ser muy útil para nuestros datos.

4. Máquinas de Soporte Vectorial:

Utilizaremos la SVC de sklearn, no tiene sentido probar con el LinearSVC por el tipo de datos que tenemos, queremos que nos capture relaciones complejas de los datos. SVC buscará el mejor hiperplano que separe las clases.

Nos encontramos 2 hiperparámetros importantes: kernel que hace referencia al tipo de núcleo que utilizaremos para transformar los datos no lineales en lineales, tenemos 3 tipos; polinomial, rbf y sigmoidal. El otro hiperparámetro es C, que hace referencia a la penalización que aplicamos a los errores, tomará valores del 1 al 3.

Para escoger los mejores hiperparámetros, hacemos un Grid Search como el que hemos hecho anteriormente, esta vez con el dataset más reducido de todos ya que SVC tarda demasiado en entrenar.

El hiperparámetro gamma, lo pondremos igual a scale, que suele ser una buena opción por defecto ya que ajusta el valor de gamma teniendo en cuenta el número de características y la varianza de los datos.

Obtenemos que los mejores hiperparámetros son kernel = rbf, que es el núcleo gaussiano debido a que es muy efectivo cuando los datos no son linealmente separables y se adapta mejor a datos con patrones complejos, y C = 3, forzando al modelo a ajustarse con más precisión a los datos de entrenamiento.

Al igual que en el anterior Grid Search, utilizamos validación cruzada.

Una vez tenemos los hiperparámetros elegidos, entrenamos otro SVC con estos, y el dataset de tamaño medio. Obtenemos un f1 = 0.915947, un valor bastante alto pero que, en comparación con otros modelos, el árbol de decisión sigue superándolo. Además, este modelo además tarda mucho en entrenar, por lo que no hemos podido entrenar con todos los datos.

5. Redes Neuronales (usando capas fully connected y dropout)

Para las redes de neuronas, primero vamos a utilizar un perceptrón multicapa de sklearn, el MLPClassifier con una capa oculta de 100 neuronas, y max_iter = 4000. Obtenemos un f1 muy alto: 0.159, el segundo más alto que obtenemos tras el árbol de decisión (entrenando también con todo el dataset). Este modelo tiene una regularización y utiliza optimizadores por defecto, por eso funciona tan bien. Hemos probado con más capas ocultas pero los resultados son mejores con una única capa oculta.

Después hemos creado una red de neuronas artificial con capas densas y de dropout.

Hemos separado para tener conjunto de validación y así ir comprobando que no hay sobreajuste para poder generalizar mejor en datos nuevos.

Vamos a probar con varios modelos de redes neuronales para ver cuál nos sale mejor f1 score.

Tenemos elementos comunes entre ellos:

- La dimensión de input es el número de características (columnas) que tenemos.
- La función de activación en capas ocultas será ReLu para introducir no linealidad, y la de salida utilizará una sigmoide ya que son valores finales 0 o 1.
- Utilizamos el optimizador Adam, ya que es eficiente y ajusta bien para problemas de clasificación con muchos ejemplos.
- Como pérdida, tenemos la binary cross entropy ya que usamos clasificación binaria.
- Ejecutamos 20 épocas con batch size = 32.

Primera red neuronal:

Con una capa densa de 100 neuronas y función de activación relu; otra capa de dropout con un valor 0.5 y una capa de salida densa de una única neurona y la función sigmoide (para que la clasificación sea binaria). Obtenemos un F1 score de 0.85529

Segunda red neuronal: tiene alternadas 3 capas densas (con 150, 75 y 25 neuronas cada una) y 3 de dropout (todas con el valor 0.5); todo esto junto con la misma capa de salida que la primera red. Obtenemos un F1_score de 0.85363

Tercera red neuronal: alterna dos capas densas (con número de neuronas decrecientes) con una capa de dropout (de valor 0.5) y la misma capa de salida que las anteriores. Obtenemos un F1_score de 0.84619

Podemos ver que, con redes más complejas, el f1 score baja (aunque sea muy poco). Por lo que, entre las redes neuronales de Keras, nos vamos a quedar con la primera, la más simple, que tiene un f1 score mayor. Hemos entrenado todas con el dataset completo.

Pero está claro que la mejor red neuronal artificial es la de MLP Clasiffier de sklearn, con el segundo mejor f1_score de todos los modelos.

Parte 3. Best-model

Finalmente, unimos todo en un solo cuaderno. Comenzamos usando las funciones que ya teníamos en la parte uno, quitando el undersampling en el primero y dejando la última parte en el segundo (pues ya tenemos las columnas que necesitamos).

Cargamos el modelo que entrenamos en la parte 2 y, tras preprocesar los datos para que encajen con el aprendizaje de nuestro RandomForestClassifier, predecimos los labels de las características alojadas en test_data. Estos datos los guardamos en un csv, test_labels.csv, resultado final de la práctica.