UNIVERSIDAD DE GRANADA

Escuela Internacional de Posgrado

Desarrollo de un paquete en SCALA y R para preprocesamiento y clasificación de datos ordinales

Mª CRISTINA HEREDIA GÓMEZ

Trabajo Fin de Máster
Máster en Ciencia de Datos e Ingeniería de Computadores

Tutor
Salvador García López

Co-tutor
Pedro A. Gutiérrez

Escuela internacional de posgrado
E.T.S. ingenierías informática y de telecomunicación

Granada, a 12 de Septiembre de 2018

Director: Salvador García López Mª CRISTINA HEREDIA GÓMEZ

Development of a package in SCALA and R for ordinal data preprocessing and classification

Abstract

The present master's final project develop a software package with multiple algorithms for efficiently preprocessing and classifying data with specific intrinsic restrictions due to its nature, as ordinal and monotonic data.

Specifically, four ordinal classification techniques are implemented in Scala: Support Vector Machine for Ordinal Regression, Proportional Odd models, Kernel Discriminant Learning for Ordinal Regression, and Weighted-nearest-neighborgs, along with two preprocessing techniques: one feature selector based in information theory which uses the Minimum Redundancy Maximum Relevance along with Rank Mutual Information, and an instance selector based in three phases which first selects promising features, then removes collisions and finally detects the relevant instances by using delimitation and interest metrics.

KEYWORDS: Machine learning, Ordinal classification, Monotonic classification, Ordinal preprocessing Monotonic preprocessing.

Director: Salvador García López Mª CRISTINA HEREDIA GÓMEZ

Desarrollo de un paquete en SCALA y R para preprocesamiento y clasificación de datos ordinales

Resumen

El presente trabajo de fin de máster desarrolla un paquete software para R de múltiples algoritmos para la eficiente clasificación y preprocesamiento de datos con restricciones intrínsecas específicas debido a su propia naturaleza, como son los datos ordinales y monotónicos.
En concreto se implementan en Scala cuatro técnicas de clasificación ordinal: Support Vector Machine for Ordinal Regression, Proportional Odd models, Kernel Discriminant Learning for Ordinal Regression, y Weighted-nearest-neighborgs, y dos algoritmos de preprocesamiento: un selector de características basado en teoría de la información que usa el criterio de Minimum Redundancy Maximum Relevance junto con Rank Mutual Information, y un selector de instancias trifásico que primero selecciona características, porteriormente elimina colisiones y por último detecta las instancias más relevantes según medidas de delimitación e interés.

KEYWORDS: Aprendizaje automático, Clasificación ordinal, Clasificación monotónica, Preprocesamiento ordinal, Preprocesamiento Monotónico.

Contenidos

Parte 1

Desarrollo teórico

Introdución al problema y descripción teórica de los algoritmos implementados.

1

Introducción

La predicción y clasificación supervisada de datos numéricos siempre ha sido un tema central de interés científico en el área de ciencia de datos y aprendizaje automático. También hay numerosas investigaciones publicadas en torno a el preprocesamiento de datos. Sin embargo, hasta hace unos años apenas se había prestado interés a la clasificación de datos ordinales, también conocida como regresión ordinal.

1.1 El problema de la clasificación ordinal

Los datos de naturaleza ordinal no son más que datos cuyas clases muestran un claro orden. Es decir, para cada instancia $x \in X \subseteq \Re^K$, con clase $y \in Y = C_1, C_2, ..., C_n$, se tiene una relación de orden sobre las clases tal que $C_1 \prec C_2 \prec ... \prec C_n$. Por ejemplo, datos recogidos de encuestas con diferentes grados de valoración, datos de resultados médicos considerando diferentes enfermedades, datos de psicología o, en general, datos obtenidos en cualquier área

científica donde los humanos intervengan para la generación de datos.

Por tanto, el problema de la clasificación de datos ordinales, o regresión ordinal, trata de clasificar datos ordinales considerando en los algoritmos el orden que presentan entre sí las clases de los datos, para obtener una clasificación más fiel a la naturaleza intrínseca de estos datos, y por tanto, construir un modelo más preciso. Pongamos como ejemplo datos obtenidos de una encuesta de valoración de un servicio que realizan un número de clientes, donde las posibles etiquetas son: [*malo*, *normal*, *bueno*, *muy bueno*, *excelente*]. En este caso, las clases contienen información de orden, pues pertenecer a la clase *malo* implica una peor valoración que pertenecer a la clase *excelente*, y por tanto un algoritmo de clasificación diseñado para tratar con datos ordinales debería considerar diferentes costes según el tipo de errores de clasificación que se produjesen. Por ejemplo, debería de penalizar más el clasificar como *malo* un servicio *excelente* que clasificarlo como *muy bueno*. Otro ejemplo similar de regresión ordinal sería la clasificación del riesgo de varias enfermedades en [*bajo*, *moderado*, *severo*] en base a los síntomas presentados por los pacientes. En este caso, la relación de orden presente sería los niveles de gravedad de una enfermedad.

Los problemas de clasificación de datos ordinales han estado presentes desde hace mucho, sin embargo se han abordado mayoritariamente como problemas nominales estándar, sin tener en cuenta el orden presente entre clases que las hace comparables, y por tanto llegando a soluciones no óptimas, aunque en trabajos como [15] se demostró que el uso de técnicas específicamente diseñadas para este tipo de problemas da mejores resultados que simplemente aplicar las técnicas estándar para datos nominales.

1.1.1 Ranking, ordenación y ranking multipartito

A pesar de su parecido, existen diferencias entre las técnicas de regresión ordinal y las de ránking, ordenación o ránking multipartito. En concreto, dado un conjunto de datos con etiquetas ordenadas, el objetivo del ránking será aprender un orden parcial subyacente en los datos, mientras que el de un problema de ordenación será de nuevo aprender un orden, en este caso

total, usando los datos de train (sin etiqueta) para ordenar los de test. El objetivo final de un problema de ranking multipartito es aprender un orden total de los datos usando para ello las etiquetas de los datos de entrenamiento, mientras que en regresión ordinal el enfoque no es encontrar un orden que describa los patrones, sino usar el orden presente en las etiquetas para tratar de asignar a las muestras las etiquetas más cercanas a las etiquetas reales de los datos.

1.1.2 Técnicas de Regresión Ordinal

Las técnicas de regresión ordinal propuestas en la literatura se agrupan principalmente en tres: *enfoques naïve, enfoques de descomposición binaria* y *enfoques basados en límites.*

Enfoques naïve

En ellos el modelo se obtiene usando otros algoritmos estándares de predicción, simplicando previamente el problema ordinal a un problema estándar. Se traducen sobre todo a problemas de regresión como NN,SVR,..., para lo que se transforman las etiquetas a valores reales [28] o se reconstruye la variable clase a partir de las distancias entre clases [26], de clasificación nominal, por ejemplo SVM [30], donde simplemente se ignora el orden de las etiquetas o de clasificación sensitiva al coste, para diferentes errores de clasificación [29].

Enfoques de descomposición ordinal

Estas técnicas se basan en descomponer las etiquetas ordinales en varias variables binarias, estimadas a partir de uno o varios modelos. Algunos de los algoritmos entrenan un modelo por cada subproblema (*multiple model*) como el enfoque seguido en [13], que considera varios problemas de clasificación binaria independientemente, que luego combina en una única etiqueta, mientras que otros aprenden un único modelo para todos (*multiple-output single model*). Por ejemplo el perceptrón ordinal en redes neuronales [8], que codifica las clases de la misma forma que Frank & Hall [13] inicialmente.

Enfoques basados en límites

Asumen que en una variable continua no observada, llamada variable latente, reside la clase ordinal. Por tanto su objetivo es estimar una función f que prediga los valores de esa variable latente y un conjunto de límites o umbrales $b = (b_1, b_2, ...b_n) \in \Re^n$ que representan intervalos en el rango de f y que cumplan que $b_1 \leq b_2 \leq ... \leq b_n$. Dentro de estos enfoques se incluyen los *Cumulative Link Models* como POM[23] que trata de extender la regresión logística binaria a regresión ordinal. También presentan este enfoque las variantes de SVM: SVOREX (Support Vector Ordinal Regression with Explicit Constraints) y SVORIM (con restricciones implícitas)[10], métodos de aprendizaje discriminativo como KDLOR (Kernel Dis- criminant Learning for Ordinal Regression)[25], clasificación binaria aumentada, ensembles (ORBoost)[21] y procesos gaussianos (GPOR)[9].

1.2 El problema de la clasificación monotónica

La clasificación monotónica es un caso especial de clasificación ordinal, donde las clases son ordinales y discretas, y existen ciertas restricciones de monotonicidad entre las características y sus clases, tal que para cualesquiera instancias x, x' donde $x \leq x' \Rightarrow f(x) \leq f(x')$. Es decir, una instancia (x') que sea mayor que otra (x) no puede tener una clase menor. Se dice que las instancias x, x' son comparables si y sólo si $x \leq x'$ o $x' \leq x$, y se dicen idénticas si $x = x'$. Por tanto, un par de instancias cualesquiera x, x' serán monótonas si se cumple que:

$$x \preceq x' \wedge x \neq x' \wedge Y(x) \leq Y(x')$$

o si

$$x = x' \wedge Y(x) = Y(x')$$

donde $Y = y_1, y_2, ..., y_c$ es la variable de clase.

Siguiendo con la definición, un conjunto de datos se dirá que es monotónico si para todos los

posibles pares de instancias se cumple que son monótonas o incomparables.
Las restricciones monotónicas están presentes en muchos datos reales, como por ejemplo en precios de viviendas, dado que los precios se incrementan a medida que lo hace el tamaño de la casa y la distancia al centro de cuidad. También están presentes en medicina, leyes y fianzas, por ejemplo para predicción de bancarota en las empresas, donde se comparan compañías donde unas dominan a otras en todos los indicadores financieros.

1.2.1 Clasificadores monotónicos

La mayoría de los clasificadores monotónicos requieren que los datos que reciben cumplan las restricciones de monotonía. En la literatura se han propuesto varios clasificadores que tienen en cuenta estas restricciones en los datos, como OLM (Ordinal Learning Model)[3] que fue el primero en hacerlo y se basa en el uso de la función:

$$f_{OLM}(x) = max\left\{y_i : x_i \in D', x_i \preceq x\right\}$$

donde $D' \subseteq D$ y D denota el conjunto de instancias. También OSDL (Ordinal Stochastic Dominance Learner) que emplea un enfoque estocástico consistentes en dos funciones[20], MID que se basa en el uso de entropía e ID3[2] y KNN Monotónico que fija unos rangos que definen los valores de clase que puede tomar cada instancia en base a su condición[12], entre otros.

1.3 Aplicaciones

Dado que existen multitud de datos de naturaleza ordinal y monotónica, técnicas de esta índole se han aplicado en problemas médicos[4] como la extracción de reglas para la detección temprana de demencia[24], estimación de edad[7], interfaz cerebro-máquina[34], credit rating[19], econometía[22], reconocimiento facial[18], clasificación de imágenes[27] y de texto[1] y asesoramiento de belleza facial[33] son algunos ejemplos de las aplicaciones actuales de este campo.

1.4 Objetivos del trabajo

- Estudio y selección de algoritmos para clasificación y preprocesamiento de datos ordinales y diseño para datos ordinales y monotónicos.
- Estudio de documentación sobre cómo hacer un paquete en R, cómo hacer código funcional e inmutable en Scala y cómo comunicarlo todo componiendo un paquete. Diseño de algoritmos y del paquete.
- Implementación de algoritmos en Scala, generación de FAT Jar, integración con R.
- Realización de experimentos y análisis de los resultados obtenidos.

1.5 Organización

En esta sección se ha introducido el problema de trabajar con datos ordinales y monotónicos, su naturaleza y aplicaciones. En el segundo capítulo ofrece una explicación teórica de los cuatro algoritmos supervisados de clasificación desarrollados (SVMOP, KDLOR, POM and WKNN) y los dos de preprocesamiento (un selector de instancias y un selector de características). El tercer capítulo instroduce el lenguaje de programación Scala empleado en el desarrollo de este paquete. El capítulo cuarto, incluye una descripción del desarrollo e instalación del paquete desarrollado para R, de nombre OCAPIS. El capítulo quinto contiene el manual de usuario que será distribuido junto con el paquete en CRAN. Finalmente, en el capítulo sexto se recogen resultados, conclusiones y futuras ideas sobre OCAPIS.

2

Algoritmos implementados

En este capítulo se revisan los seis algoritmos de preprocesamiento y clasificación de datos ordinales y monotónicos implementados en el paquete software desarrollado, ofreciendo una descripción teórica de los mismos. Para ello se divide el capítutlo en dos secciones: algoritmos de preprocesamiento y algoritmos de clasificación.

2.1 Preprocesamiento

2.1.1 Selector de características para clasificación monotónica

Se implementa en el paquete el algoritmo para selección de características[17] basado en Rank Mutual Information. El funcionamiento del algoritmo propuesto por Qinghua Hu et Al. es el siguiente:

Consideremos un conjunto de datos $\langle U, A, D \rangle$ donde U denota las instancias, A denota los

atributos y D las clases, tal que $D = d_1 < d_2 < ... < d_k$. Entonces, para cada atributo $a \in A$ se calculan las relaciones entre las instancias usando la siguiente función *logsig*:

$$r_{ij}^{<} = \frac{1}{1 + e^{\,k\ (v(x_i,a) - v(x_j,a))}}$$

Donde $v(x_i, a)$ es el valor de la instancia x_i para el atributo a, y k es una constante positiva. El resultado de aplicar esta función es:

$$\left\{ \begin{array}{ll} 0.5, & \text{si } r_{ii}^{>} = r_{ii}^{<} \\ \approx 1, & \text{si } v(x_j, a) >> v(x_i, a) \\ \approx 0, & \text{si } v(x_j, a) << v(x_i, a) \end{array} \right\}$$

Es decir, en el primer caso no habría diferencia entre x_i y x_j, en el segundo caso x_i sería significativamente inferior a x_j, y en el tercer caso x_i sería significativamente mayor.
Con esto se calculan los conjuntos ordinales difusos mayores que x_i en términos de a de la siguiente forma:

$$[x_i]_a^{\leq} = r_{i1}/x_1 + r_{i2}/x_2 + ... + r_{in}/x_n$$

que serán posteriormente usados para calcular la *mutual rank information* entre dos atributos cualesquiera sobre un conjunto de instancias de la siguiente forma:

$$RMI_{a_1,a_2}(U) = -\sum_{i=1}^{n} \frac{1}{n} \log \frac{|[x_i]_{a_1}^{\leq}| \times |[x_i]_{a_2}^{\leq}|}{n \times |[x_i]_{a_1}^{\leq} \cap [x_i]_{a_2}^{\leq}|}$$

Esta ganancia de información se combina junto con la ganancia de información de cada característica con respecto a la clase, obteniendo el criterio mRMR (minimum redundancy maximum relevance), que trata de encontrar las características más significativas y menos relevantes:

$$\Phi = \frac{1}{|B|} \sum_{a_i \in B} RMI_{a_i,D} - \frac{\beta}{|B|^2} \sum_{a_i,a_j \in B} RMI_{a_i,a_j}$$

donde β es un parámetro regulativo. Finalmente se seleccionan las k características con mayor mRMR.

2.1.2 Selector de instancias para clasificación monotónica

EL selector de instancias para datos monotónicos implementado en el paquete es el algoritmo propuesto por J.-R. Cano, S. García, de nombre MonTSS[5].
MonTSS tiene como objetivo eliminar las instancias que no son interesantes para el problema monotónico, lo que reduce la dimensión del problema y, como efecto colateral, los costes de computación. El algoritmo se compone de tres fases, como ilustra la siguiente figura:

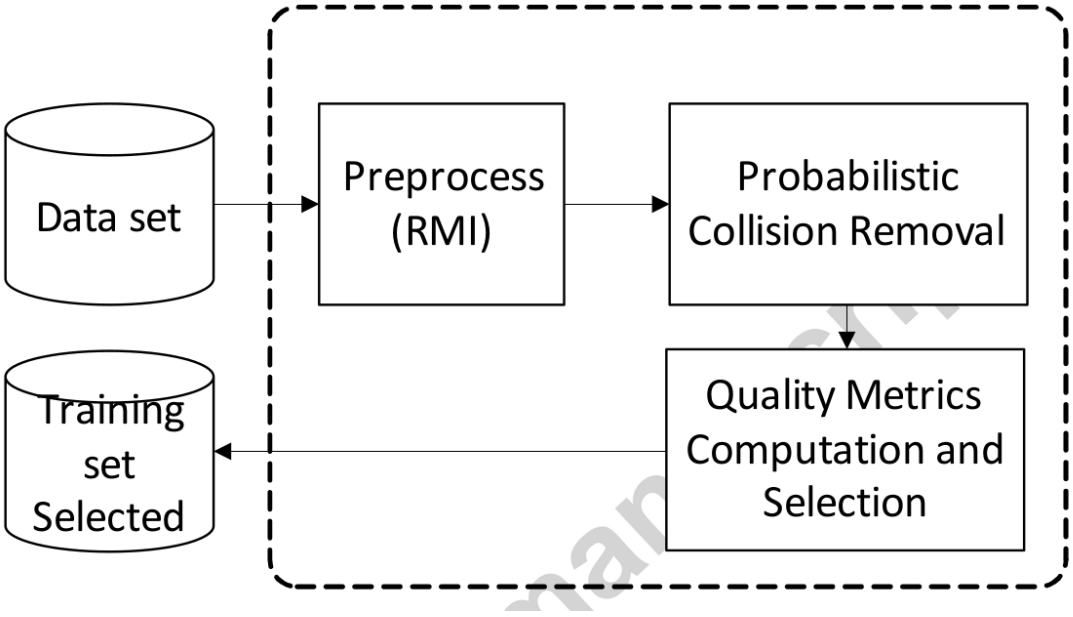


Figure 2.1: Fases del proceso de MonTSS.

- **Preprocesamiento**: se analiza inicialmente la relación de cada característica con la clase, usando RMI de forma similar a la descrita en el algoritmo anterior. Las características seleccionadas en este paso serán las empleadas en pasos posteriores del algoritmo.
- **Eliminación de colisiones**: se trata de un proceso iterativo en el que se eliminan la mayoría de las instancias que producen colisiones, es decir, las que no son monótonas.

Para ello se comienza calculando el número de colisiones que produce cada instancia, tras lo que se ordenan de forma decreciente. Luego se emplean la tasa de candidatos a seleccionar y el número de colisiones permitidas especificadas previamente al algoritmo como parámetro, para eliminar las instancias. Este último paso se hace de forma probabilística, pues para seleccionar la instancia a eliminar se genera un valor entre $[1, maxCandidates]$, Donde *maxCandidates* se calcula a partir del tamaño de los datos y de la tasa de candidatos. El proceso se repite hasta que se alcanza el número permitido de colisiones.

- **Cálculo de métricas**: de entre las instancias seleccionadas en el paso anterior, se seleccionan finalmente las más importantes, que son las que se corresponden con fronteras de decisión. Para detectarlas, se emplean dos métricas, llamadas *Delimitación* e *Influencia.* Se calculan como sigue:

$$Del(x_i) = \frac{|Dom(x_i) - NoDom(x_i)|}{Dom(x_i) + NoDom(x_i)}$$

Donde

$$Dom(x_i) = x' \in X' \iff x_i \prec x' \wedge Y(x_i) = Y(x')$$

$$NoDom(x_i) = x' \in Z' \iff x_i \succeq x' \wedge Y(x_i) = Y(x')$$

Es decir, $Dom(x_i)$ son las instancias con igual clase que x_i que dominan a ésta, mientras que $NoDom(x_i)$ son las instancias con igual clase a x_i que son dominadas por ella.
Por otro lado, la influencia de una instancia x_i se calcula empleando los vecinos con diferente clase y su distancia a x_i, que se traduce en un peso asociado:

$$Infl(x_i) = \sum_{j=1}^{k} influenceWeight(x_j)$$

donde *influenceWeight* se calcula como el peso normalizado de la instancia x_j y se cumple que $Y(x_i) \neq Y(x_j) \wedge x_j \in kNN_{xi}$.
Ambas métricas toman valores en el rango $[0, 1]$, considerando 1 como muy relevante.

2.2 Clasificación

2.2.1 SVMOP

Se implementa el ensemble de WSVMs para clasificación ordinal[31] usando descomposición binaria. Esta técnica se basa en descomponer el problema en varios subproblemas binarios, entrenando $r-1$ SVM's, donde r denota el número de clases del problema. Sigue un enfoque muy similar al de emsembles probabilísticos que propuso Frank & Hall[14]:

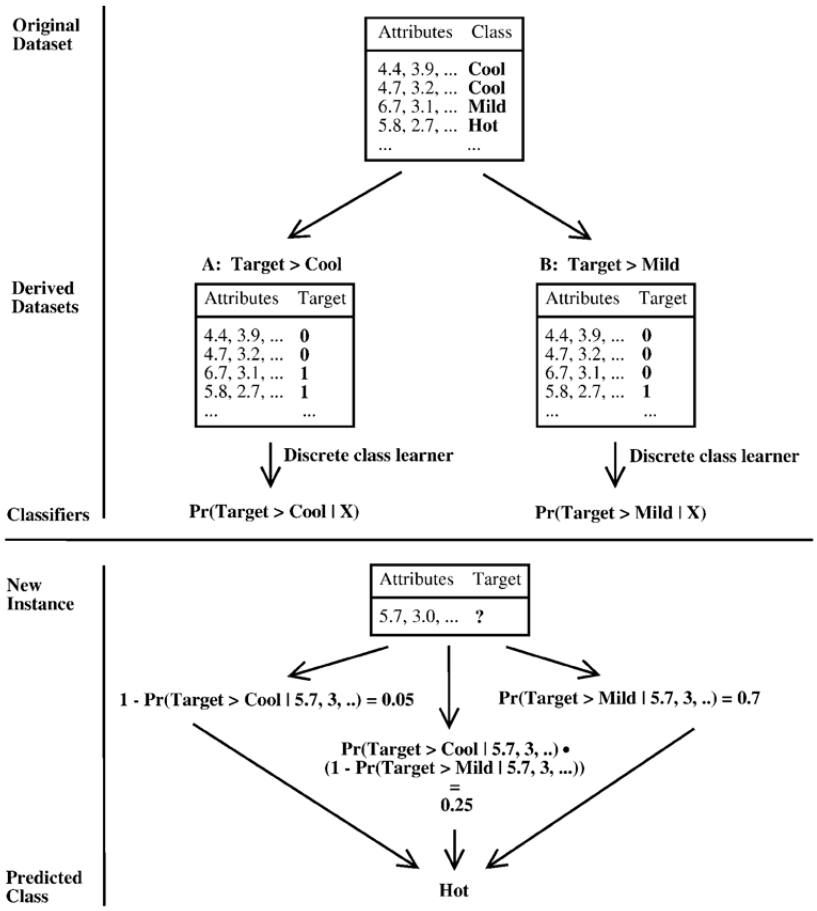


Figure 2.2: Aplicando algoritmos de clasificación estándar a predicción ordinal.

En concreto, para cada clasificador binario se transforman las instancias de entrenamiento

a la clase positiva o negativa, binarizándolas en función de si su etiqueta es mayor o no que p, donde $p = 1, \ldots, r-1$.

$$y_{pi} = \left\{ \begin{array}{ll} \text{-1}, & \text{si } y_i \leq p \\ \text{1}, & \text{si } y_i > p \end{array} \right\}$$

Siguiendo con el enfoque Frank & Hall, tras esto hay que asignarle un peso a cada una de las muestras, que será diferente para cada clasificador, lo que implica resolver los $r-1$ problemas de optimización subyacentes.
Los pesos se asignan de forma que los errores de clasificación sean penalizados proporcionalmente a la diferencia absoluta de la clase de la instancia y p:

$$v_{pi} = \left\{ \begin{array}{ll} (p+1-y_i)\frac{|\{x_i|y_i \leq p\}|}{\sum_{i=1;y_i \leq p}^{n}(p+1-y_i)}, & \text{si } y_i \leq p \\ (y_i-p)\frac{|\{x_i|y_i > p\}|}{\sum_{i=1;y_i > p}^{n}(y_i-p)}, & \text{si } y_i > p \end{array} \right\}$$

y se escalan para que mantenga igual la suma de errores de las clases positiva y negativa. Las clases de los datos de test se estiman combinando las $\hat{y}_{pi}$ predicciones obtenidas por los $r-1$ clasificadores, de tal forma que asignaremos la clase k a x_i si $\hat{y}_{pi} = 1$ para todo $p < (k-1)$ y $\hat{y}_{pi} = -1$ para todo $p \geq k$. Sin embargo, para problemas multiclase a menudo ocurren ambigüedades con este proceso de asignación de clase, por lo que se asigna la clase con mayor probabilidad. En la implementación se usa *libsvm-weights*[6] para el entrenamiento de las SVMs.

2.2.2 POM

Se implementa el modelo linear de regresión *Proportional Odd Model for Ordinal Regression*[23] como un modelo de regresión logística, en base a lo siguiente:
Sean k las categorías ordenadas de la variable de clase, con probabilidades $\pi_1(x), \pi_2(x), \ldots, \pi_k(x)$ donde las variables observadas toman valor x. En el caso de que hubiera dos grupos, x sería una variable factor que indicaría la pertenencia a cada grupo. Sea Y la variable de clase que toma valores en el intervalo $[1, \ldots, k]$ con las probabilidades mencionadas arriba, y sea $k_j(x)$ la

cuota de que $Y \leq j$ dado x. Entonces el modelo POM especifica que:

$$k_j(x) = k_j \exp(-\beta^T x), (1 \leq j < k)$$

Donde β es un vector de parámetros desconocidos. El ratio de las cuotas correspondientes se calcula como:

$$k_j(x_1)/k_j(x_2) = \exp\left\{\beta^T(x_2 - x_1)\right\}, (1 \leq j < k)$$

y es independiente de j, dependiendo únicamente de la diferencia de las covariables x_2, x_1. Entonces, como la cuota de $Y \leq j$ es el ratio $\gamma_j(x)/\left\{1 - \gamma_j(x)\right\}$, donde $\gamma_j(x) = \pi_1(x) + ... + \pi_j(x)$, el modelo POM resulta idéntico a el modelo de regresión logística lineal:

$$\log\left[\gamma_j(x)/\left\{1 - \gamma_j(x)\right\}\right] = \theta_j - \beta^T x, (1 \leq j < k)$$

donde $\theta_j = \log(k_j)$ por lo que la diferencia entre logits cumulativos es independiente de la categoría.

2.2.3 KDLOR

Kernel Discriminant Learning for Ordinal Regression[25] es una versión Kernel de LDA aplicable a datos no lineales de naturaleza ordinal.
Consideremos un problema de regresión ordinal con K clases ordenadas, tal que $Y = \{1, 2, ..., K\}$. El objetivo será encontrar una proyección de los datos donde:

- se minimize la distancia dentro de las clases y se maximize la distancia entre clases, como en un enfoque LDA.
- se respete la información de orden de las diferentes clases, por ejemplo, la proyección de las muestras con mayor rango deben ser mayores que las proyecciones de las clases con bajo rango.

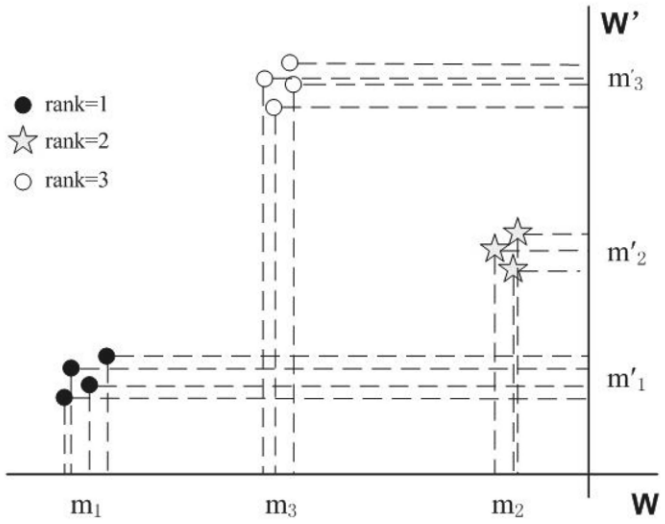


Figure 2.3: W es mejor para clasificación según el LMP, mientras que W' preserva el orden de las clases.

Aunque la proyección sobre **w** resulta mejor para la clasificación según el principio de máxima-marginalidad, la proyección sobre **w'** preserva el orden de las clases, a diferencia de **w**. Por tanto el problema es:

$$min(\mathcal{J}, \rho) = w^T \cdot S_w \cdot w - C \cdot \rho$$

sujeto a:

$$w^T \cdot (m_{k+1} - m_k) \geq \rho, k = 1, 2, ..., K-1$$

donde

- S_w es la matriz dispersa de separación interclase:

$$S_w = \frac{1}{N} \sum_{k=1}^{K} \sum_{x \in X_k} (x - m_k)(x - m_k)^T$$

- m_k es la media de las muestras de la clase k-ésima:

$$m_k = \frac{1}{N_k} \sum_{x \in X_k} x$$

- C es un coneficiente de penalización

Es decir, el modelo trata de minimizar las varianzas de los datos de las mismas clases mientras que al mismo tiempo trata de extender la diferencia entre las medias proyectadas entre dos clases vecinas, lo que hace que por un lado maximize la distancia entre dos vectores medios del par de clases más cercano, y por otro lado utilize la información de distribución de los datos en el algoritmo.
Para resolver el problema anterior, se resuelve una ecuación de Lagrange con multiplicadores $\alpha_k \geq 0$, quedando a resolver el siguente problema de optimización de programación cuadrática convexa (QP) con restricciones lineales:

$$\min f(\alpha) = \sum_{k=1}^{K-1} \alpha_k (m_{k+1} - m_k)^T S_w^{-1} \sum_{k=1}^{K-1} \alpha_k (m_{k+1} - m_k)$$

sujeto a $\alpha_k \leq 0, k = 1, \ldots, K-1$ y $\sum_{k=1}^{K-1} \alpha_k = C$. Tras obtener la dirección óptima de $\mathbf{w}$ usando α, la clase de una nueva muestra se clasifica usando la siguiente regla de decisión:

$$f(x) = \min_{k \in \{1,\ldots,K\}} \{k : w \cdot x - b_k < 0\}$$

donde

$$b_k = w(m_{i+1} + m_i)/2$$

Para el caso no lineal, será necesario mapear los datos originales a un espacio de mayor dimensionalidad usando una función de mapeo $\varphi : x \to \varphi(x)$, por lo que $\mathbf{w}$ quedaría como:

$$w = \sum_{i=1}^{N} \beta_i \varphi(x_i), \beta_i \in \Re$$

Quedando el problema de optimización como:

$$\min \mathcal{J}(\beta, \rho) = \beta^T \cdot H \cdot \beta - C \cdot \rho$$

sujeto a

$$\beta^T \cdot (M_{k+1} - M_k) \geq \rho, k = 1, ..., K-1$$

donde

- $(M_k)_j = \frac{1}{N_k} \sum_{x \in X_k} \varphi(x_j) \cdot \varphi(x)$
- $H = \sum_{k=1}^{K} P_k (I - 1_{N_k}) P_k^T$
- P_k es una matriz NxN_k con $(P_k)_{i,j} = \varphi(x_i) \cdot \varphi(x_j), x_j \in X_k$
- I es la matriz identidad
- 1_{N_k} es la matriz con entradas $1/N_k$

Por lo que usando núcleos de mercer con un conjunto de funciones de $s(x_i, x_j) = \varphi(x_i) \cdot \varphi(x_j)$ se resuelve el problema de forma similar al caso linear.

2.2.4 WKNN

Se implementa la extensión de KNN para datos ordinales *Weighted k-Nearest-Neighbors*[16] que se basa en la idea de que las observaciones del conjunto de entrenamiento más próximas a la nueva observación a clasificar deben tener un mayor peso en su decisión. Además, para que el algoritmo sea también aplicable a datos monotónicos, se le incorporan restricciones de monotonicidad[12].

El procedimiento del algoritmo se puede describir como:

1. **Estandarización de variables**: previamente al cálculo de las distancias, para lo que se divide cada variable por su desviación estándar.
2. **Obtención de vecinos**: Sea $L = \{(y_i, x_i), i = 1, ..., n_L\}$ un conjunto de entrenamiento donde x_i denota la instancia e y_i denota la clase. Dada una nueva instancia de test x a clasificar, se obtienen sus $k+1$ vecinos más cercanos de acuerdo a la distancia de Minkowski, dado el parámetro q de la misma.

3. **Estandarización por el (k+1)ésimo vecino**: se estandarizan las k distancias obtenidas en el paso anterior usando el vecino $k+1$ que no será considerado para la clasificación de x, tal que:

$$D_{(i)} = D(x, x_{(i)}) = \frac{d(x, x_{(i)})}{d(x, x_{k+1})}$$

4. **Asignación de pesos**: se transforman las distancias $D_{(i)}$ normalizadas a pesos, usando para ello alguna de las funciones Kernel implementadas, tal que $w_{(i)} = K(D_{(i)})$. Los Kernels que se implementan son:

   - Rectangular: $\frac{1}{2} \cdot I(|d| \leq 1)$
   - Triangular: $(1 - |d|) \cdot I(|d| \leq 1)$
   - Epanechnikov: $\frac{3}{4}(1 - d^2) \cdot I(|d| \leq 1)$
   - Biweight: $\frac{15}{16}(1 - d^2)^2 \cdot I(|d| \leq 1)$
   - Triweight: $\frac{35}{32}(1 - d^2)^3 \cdot I(|d| \leq 1)$
   - Coseno: $\frac{\pi}{4} \cos(\frac{\pi}{2} d) \cdot I(|d| \leq 1)$
   - Gaussiano: $\frac{1}{\sqrt{2\pi}} \exp(-\frac{d^2}{2})$
   - Inversión: $\frac{1}{|d|}$

5. **Asignación de clase**: por último, se etiqueta x con clase y donde y es la clase con mayor peso de entre sus vecinos.

Si se especifica el parámetro de monotonía, entonces se aplican restricciones monotónicas en la asignación de clases, de tal forma que se restinge la clase que puede tomar una instancia al intervalo $[y_{min}, y_{max}]$, donde:

$$y_{min} = \max \left\{ y | (x', y) \in D \wedge x' \leq x \right\}$$

y

$$y_{max} = \min \left\{ y | (x', y) \in D \wedge x \leq x' \right\}$$

Por tanto el procedimiento es el mismo que el anterior pero, en este caso, al obtener los k vecinos de la nueva instancia a clasificar, x, le asignamos la etiqueta de sus vecinos más frecuente que se encuentre en el intervalo $[y_{min}, y_{max}]$. Si ninguna etiqueta se encuentra en ese intervalo, se le asigna una clase aleatoria del intervalo.

"If I were to pick a language to use today other than Java, it would be Scala."

James Gosling

3

Introducción a Scala

SCALA, (Scalable language) es un lenguaje de programación de propósito general cuya primera versión fue lanzada en 2004, y desde entonces ha ido creciendo enormemente en usuarios. Estas son algunas razones por las que se decide usar Scala en este proyecto (Wampler & Payne [32]):

Es escalable

Tal y como su nombre indica, este lenguaje ha sido diseñado para crecer con las demandas de sus usuarios, por lo que es una buena opción para escribir desde scripts pequeños a grandes sistemas, abordar desafíos actuales como el Big data o proporcionar servicios con gran disponibilidad y robustez. Esta es la principal razón por la que se escoge este lenguaje para este proyecto, dado que la intención es ir ampliándolo.

Soporta un paradigma mixto

Por una parte Scala soporta programación orientada a objetos (POO), mejorando los objects de Java incluyendo los traits, una forma clara de implementar los tipos usando composiciones mixtas. En Scala todo son objetos realmente, incluso los tipos numéricos. Por otro lado, también soporta totalmente programación funcional (FP), herramienta que se ha convertido en la mejor forma de pensar en concurrencia, Big data y corrección del código en general (empleo de inmutabilidad, funciones de primera clase, funciones de alto orden...).

Tiene un sofisticado sistema de tipos

Extiende el sistema de tipos de Java con otros tipos genéricos más flexibles y otras mejoras para mejorar la corrección del código. Además Scala incorpora un mecanismo de inferencia de tipos.

Es estaticamente tipado

Scala incorpora el tipado estático como herramienta para crear aplicaciones más robustas, pero añade algunas modificaciones para hacerlo más llevadero, como incorporar la inferencia de tipos y hacerlo más flexible, permitiendo identificación de patrones y nuevas formas de escribir y componer tipos.

Un lenguaje JVM y Javascript

Explota las funcionalidades y optimizaciones de JVM, así como la gran cantidad de liberías y herramientas disponibles para Java. Además tiene un puerto para JavaScript (Scala.js).

Sintaxis concisa, elegante y flexible

Si de algo hablan los programadores de Scala es de su sintaxis y de las reducciones de código que experimentan con respecto a Java. Por ejemplo:

```
// código en Java
class Persona {
  private int edad;
  private String nombre;

  public Persona(int edad, String nombre) {
    this.edad = edad;
    this.nombre = nombre;
  }
}

// código en Scala
class Persona(edad: Int, nombre: String)
```

Sí, dado este código en Scala el compilador creará dos atributos privados edad(entero) y nombre(cadena de caracteres) y un constructor que tomará los valores que se le pasen inicialmente para inicializar esas variables. Más rápido de escribir, de leer, y menos errores.

3.1 Más ejemplos de Scala

A continuación se muestran algunos ejemplos simples de Scala obtenidos de ?

3.1.1 Encontrar caracter en mayúscula

Éste es un ejemplo en Java y Scala del problema de encontrar un caracter en mayúcula en una cadena de texto:

```
// código en Java
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
   if(Character.isUpperCase(name.charAt(i))) {
     hasUpperCase = true;
     break;
   }
}
// código en Scala
val hasUpperCase = name.exists(_.isUpper)
```

Mientras que el código en Java itera sobre cada caracter del string, comprobando uno a uno hasta que encuentra una mayúscula, modifica el flag boolenano y se sale del bucle, en Scala basta con llamar a la función exist sobre el string name, devolviendo un booleano. La _ representa cada caracter de la cadena. De nuevo Scala seduce con su sintaxis.

3.1.2 Contar las líneas de un fichero

```
# código en Ruby
count = 0
File.open "someFile.txt" do |file|
file.each { |line| count += 1 }
end
```

```
// código en Scala
val count = scala.io.Source.fromFile("someFile.txt").getLines().map(x => 1).sum
```

En Ruby también se realiza de forma breve, pero no tan elegante, ya que necesita una variable contador que va incrementando en cada vez que cuenta una línea. En Scala, una posible forma

de hacerlo es usando map, una función que a cada línea del fichero le hace corresponder un 1. Finalmente los suma todos llamando a sum.

Además, Scala soporta composición mixta a través de los traits. Los traits son parecidos a las clases abstractas con implementación parcial. Por ejemplo, podríamos crear un nuevo tipo de colección que permitiera acceder al contenido del fichero como un iterable, mezclando el trait Iterable de Scala:

```
class FileAsIterable {
def iterator = scala.io.Source.fromFile("someFile.txt").getLines()
}
```

Ahora si lo mezclamos con el trait Iterable de Scala, al crear un objeto de esa clase, éste será un Iterable, teniendo acceso a los métodos de Iterable:

```
val newIterator = new FileAsIterable with Iterable[String]
newIterator.foreach { line => println(line) }
```

donde el método foreach al que llama, es de Iterable.

3.2 Scala y la concurrencia

Uno de los grandes problemas de la concurrencia es que si no se coordina bien el acceso a los recursos puede haber cambios inesperados e indeseados en los mismos, por acción de alguna de las hebras. Scala ayuda en esto con la inmutabilidad, de hecho, por defecto lo hace todo inmutable. Aunque en Scala se puede usar cualquier mecanismo de Java, Scala incorpora también sus propias herramientas específicas para concurrencia. Algunas de ellas son: Wampler & Payne [32]

3.2.1 Futures

La API scala.concurrent.Future simplifica la concurrencia en código. Los Futures empiezan a correr concurrentemente cuando son creados, aunque lo hacen de forma asíncrona. Se puede hacer que las tareas sean independientes y sin bloqueo o por el contrario, bloquear, y además la API ofrece muchas funcionalidades para manejar los resultados (que pueden ser un Future), como callbacks que pueden ser invocados cuando el resultado esté listo. Veamos un ejemplo simple en el que se lanzan concurrentemente 5 tareas:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
def sleep(millis: Long) = {
  Thread.sleep(millis)
}
// Busy work ;)
def doWork(index: Int) = {
  sleep((math.random * 1000).toLong)
  index
}
(1 to 5) foreach { index =>
   val future = Future {
    doWork(index)
   }
   future onSuccess {
       case answer: Int => println(s"Success! returned: $answer")
   }
   future onFailure {
       case th: Throwable => println(s"FAILURE! returned: $th")
   }
}
```

```
    sleep(1000) // Wait long enough for the "work" to finish.
    println("Finito!")
```

Se usa un método SLEEP para simular la ocupación por un tiempo. El método DOWORK llama a sleep con un número aleatorio de milisegundos. Se itera con foreach en un rango de enteros (1 a 5 inclusive) y se llama a scala.concurrent.Future.apply. En este caso, Future.apply recibe una función de trabajo a realizar (doWork), y devuelve un nuevo objetoFuture, el cual ejecuta el cuerpo de doWork (index) en otra hebra, devolviendo el control inmediatamente al bucle. Finalmente se usan onSucess (si la tarea se completa correctamente) y onFailure (si falla) para registrar los callbacks.

3.2.2 MODELO DE ACTORES

Una regla de la concurrencia es: si puedes, no compartas. Los actores son entidades software independientes que no comparten unos con otros información mutable que se pueda alterar. En su lugar se comunican mediante mensajes, eliminando la necesidad de sincronizar el acceso a datos, estados e información mutable. De esta forma es más fácil crear aplicaciones concurrentes robustas. Cada actor cambia su estado según lo necesite, pero sólo si tiene acceso exclusivo a ese estado y sus invocaciones se garantizan seguras.

3.3 SCALA Y BIG DATA

Las funciones map, flatMap, filter, fold, reduce...etc siempre han sido funciones para trabajar con datos, independientemente de que estos fueran grandes o pequeños. Es por esto que una vez que se entiende Scala y sus colecciones, resulta fácil enganchar una API de Big data basada en Scala, sin embargo, sabiendo Java, enganchar con MapReduce Java API resulta más complicado por ser más difícil de usar y estar a más bajo nivel.
Pero la principal ventaja de Scala en Big data frente a otros lenguajes no es la curva de aprendizaje, sino la programación funcional. Scala permite escribir lo mismo (o más), con menos

código que otros lenguajes. El ejemplo simple de contar el número de palabras, sacado de Wampler & Payne[32] bastará para evidenciar a simple vista la ventaja de las APIs Big data basadas en Scala frente a las basadas en Java.

Esto es sólo parte de la versión Hadoop. El original se compone de unas 60 líneas de código, sin imports ni comentarios.

```
// src/main/java/progscala2/bigdata/HadoopWordCount.javaX
...
class WordCountMapper extends MapReduceBase
implements Mapper<IntWritable, Text, Text, IntWritable> {
   static final IntWritable one = new IntWritable(1);
   static final Text word = new Text;

   @Override public void map(IntWritable key, Text valueDocContents,
   OutputCollector<Text, IntWritable> output, Reporter reporter) {
       String[] tokens = valueDocContents.toString.split("\\s+");
       for (String wordString: tokens) {
         if (wordString.length > 0) {
           word.set(wordString.toLowerCase);
           output.collect(word, one);
         }
       }
   }
}
class WordCountReduce extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text keyWord, java.util.Iterator<IntWritable> counts,
    OutputCollector<Text, IntWritable> output, Reporter reporter) {
       int totalCount = 0;
       while (counts.hasNext) {
```

```
            totalCount += counts.next.get;
        }
```

Ahora una versión Scalding, basado en Scala. El código orignial tiene 12 líneas de código, contando el import:

```
// src/main/scala/progscala2/bigdata/WordCountScalding.scalaX
import com.twitter.scalding._

class WordCount(args : Args) extends Job(args) {
    TextLine(args("input"))
    .read.flatMap('line -> 'word) {
       line: String => line.trim.toLowerCase.split("""\s+""")
    }.groupBy('word){ group => group.size('count) }
    .write(Tsv(args("output")))
}
```

Parte 2

Desarrollo práctico

Descripción y documentación del software que implementa las técnicas mencionadas anteriormente.

4

Descripción del Software desarrollado

Ésta es la primera versión de OCAPIS, una librería software para preprocesamiento y clasificación de datos ordinales y monotónicos desarrollada para R pero implementada en Scala mayoritariamente, para potenciar su rendimiento. Para comunicar R con Scala se hace uso del paquete **rScala**[Dahl].

4.1 Instalación

Como dependencias a nivel de sistema, el paquete requiere que el usuario tenga instaladas alguna versión de Python y alguna versión de Scala en su sistema. Si este requisito no se cumple, el paquete no funcionará. El requerimiento de *Scala* es debido a que el serializer necesitará que esté instalado en el sistema para poder hacer las comunicaciones entre Scala y R.

4.1.1 Instalación de Scala

Scala puede instalarse desde los repositorios, en caso de usar linux:

```
$ sudo apt-get install scala
```

En caso de Ubuntu, sustituyendo el comando por *emerge* si la distribución usada es gentoo o el correspondiente comando de instalación. Scala también puede instalarse desde su página web: https://www.scala-lang.org/download/install.html.

4.1.2 Instalación de Python

Python se instala de forma similar, vía el repositorio o la página oficial : https://www.python.org/downloads/. Python es requerido porque la SVM ordinal implementada usa una librería C con wrapper para python para aplicar pesos por instancias, de nombre **libsvm-weights**.

4.1.3 Instalación de Libsvm-weights

Esta librería es muy potente, pero no la podemos encontrar en ningún repositorio. Es por tanto que debemos descargarla desde github: https://github.com/claesenm/EnsembleSVM/tree/master/libsvm-weights-3.17 y compilarla para windows o linux, según nuestro sistema, usando el makefile correspondiente (seguir las instrucciones del README de la libería). Por último debemos colocar la librería compilada en el directorio de librerías de usuario: /usr/local/lib o /usr/lib en Linux, o su homólogo en windows.

4.1.4 Instalación de OCAPIS

Actualmente OCAPIS puede descargarse desde Github, usando el paquete *devtools* desde R:

```
$ devtools::install_github("CristinaHG/OCAPIS")
```

Aunque pronto podrá descargarse desde CRAN. El resto de dependencias R del paquete se instalarán automáticamente.

4.2 Uso

Todos los algoritmos de clasificación de OCAPIS tienen un método asociado para entrenar con los datos de entrenamiento y otro para clasificar las instancias de test con el modelo aprendido, siguiendo un enfoque típico en los paquetes de R. Por ejemplo, KDLOR incluye un método *kdlortrain* y otro *kdlorpredict*, SVMOP incluye otros dos métodos *svmofit* y *svmopredict*, etc, cuyo uso se detalla en el manual de usuario.
Por ejemplo, así sería el uso de SVMOP con el dataset ordinal balance:

```
library("OCAPIS")
 # lectura de datos de train
 dattrain<-read.table("train_balance-scale.0", sep=" ")
 # entrenamiento del modelo
 modelstrain<-svmofit(dattrain[,-ncol(dattrain)],dattrain[,ncol(dattrain)],TRUE,1,1)
 # lectura de datos de test
 dattest<-read.table("test_balance-scale.0", sep=" ")
 # cálculo de predicciones
 predictions<-svmopredict(modelstrain,dattest[,-ncol(dattest)])
```

La documentación completa para estos métodos puede consultarse con

```
?OCAPIS::svmofit
?OCAPIS::svmopredict
```

5

Manual de usuario de OCAPIS 1.0

5.1 Fselector

5.1.1 Tittle

Feature Selection for Monotonic Classification.

5.1.2 Use

```
fselector(traindata, trainlabels, k, beta, nselected)
```

5.1.3 Arguments

- **traindata**: Training data of numeric type without labels.
- **trainlabels**: A vector of numeric tags for each instance of training data.

- **k**: positive constant for logit function. If large, fuzzy ordinal set is understood as slightly larger. If small, is understood as significantly larger.
- **beta**: Regulation param for relative importance of MI between features and decision.
- **nselected**: Number of features to select.

5.1.4 Return

nselected most important features.

5.1.5 Description

Selects the N most relevant features from ordinal and monotonic data, based on mRMR criterion.

5.1.6 Examples

```
dattrain<-read.table("train_balance-scale.0", sep=" ")
trainlabels<-dattrain[,ncol(dattrain)]
traindata=dattrain[,-ncol(dattrain)]
selected<-fselector(traindata,trainlabels,2,2,2)
```

5.2 Iselector

5.2.1 Tittle

Instance Selection for Monotonic Classification

5.2.2 Use

```
iselector(traindata, trainlabels, candidates, collisions, kEdition)
```

5.2.3 Arguments

- **traindata**: Training data of numeric type without labels.
- **trainlabels**: A vector of numeric tags for each instance of training data.
- **candidates**:Rate of the best candidates to be selected.
- **collisions**: Minimal rate of collisions permitted to stop the removal process.
- **kEdition**: Maximum number of nearest neighborgs to consider.

5.2.4 Return

A reduced dataset with the selected instances and its labels.

5.2.5 Description

Selects the N most relevant instances from ordinal and monotonic dataset, using a three-steps algorithm.

5.2.6 Examples

```
dattrain<-read.table("train_balance-scale.0", sep=" ")
trainlabels<-dattrain[,ncol(dattrain)]
traindata=dattrain[,-ncol(dattrain)]
selected<-iselector(traindata,trainlabels,0.01,0.01,5)
```

5.3 kdlorpredict

5.3.1 Tittle

Predicts KDLOR model outputs for test data.

5.3.2 Use

```
kdlorpredict(fittedmodel, trainData, testData)
```

5.3.3 Arguments

- **fittedmodel**: fitted model of class kdlorModel obtained with kdlortrain(...).
- **trainData**: Data used to previously train the model. Tags should not be provided in traindata.
- **testData**: Test data without labels.

5.3.4 Return

A list of two elements containing the predicted labels for each instances and the projected values.

5.3.5 Description

Predicts the test data labels using a Kernel Discriminant Learning for Ordinal Regression fitted model

5.3.6 Examples

```
testdata<-read.table("test_balance-scale.0", sep=" ")
testdata<-testdata[,-ncol(testdata)]
pred<-kdlorpredict(myfit,traindata,testdata)
```

5.4 kdlortrain

5.4.1 Tittle

Trains a KDLOR model.

5.4.2 Use

```
kdlortrain(traindata, trainlabels, kernel, d, u, k)
```

5.4.3 Arguments

- **traindata**: Data to train kdlor model. Tags should not be provided in traindata.
- **trainlabels**: Class labels for training data. Must be numeric of type integer.
- **kernel**: Type of kernel to compute the Gram matrix. One of *"rbf","gauss","gaussian","sigmoid", "linear","poly","polynomial"*.
- **u**: Numeric parameter for H matrix computation. Default is 0.01.
- **k**: Array of kernel Params. If kernel type is sigmoid, Array of two values should be provided.
- **c**: Numeric parameter for optimization method. Default is 10..

5.4.4 Return

An instance of kdlorModel class containing the fields: projectedTrain, predictedTrain, kerneltype, kernelParam, projection and thresholds, where:

- projectedTrain is the projected matrix for training data.
- predictedTrain are the predicted numeric labels for training data.
- kerneltype is the used kernel type for the fit. Should be used again for prediction.
- kernelParam is the numeric kernel param used for computing the kernel matrix.
- projection is the general projected matrix that should be used in predict.
- thresholds is an array of doubles representing the model thresholds to be used in prediction.

Each of these fields can be accesed with @ (see section examples) below.

5.4.5 Description

Trains the Kernel Discriminant Learning for Ordinal Regression model with training data

5.4.6 Examples

```
# read train data
dattrain<-read.table("train_balance-scale.0", sep=" ")
traindata=dattrain[,-ncol(dattrain)]
trainlabels=dattrain[,ncol(dattrain)]
# fit the kdlor model
myfit<-kdlortrain(traindata,trainlabels,"rbf",10,0.001,1)
# acess kdlor model fields
myfit@predictedTrain
```

5.5 pomfit

5.5.1 Tittle

Train a proportional odd model for ordinal regression.

5.5.2 Use

```
pomfit(train, trainLabels, linkfunction = "logistic")
```

5.5.3 Arguments

- **linkfunction**: link function to be used in the ordinal logistic regression fit. Possible functions are: logistic',probit',loglog',cloglog' or 'cauchit'.
- **train**: Training data of numeric type without labels.
- **trainLabels**: Tags for each instance of training data. Must be factors.

5.5.4 Return

the fitted model.

5.5.5 Description

train data must be the training data without labels. Labels should be provided in trainLabels.

5.5.6 Examples

```
dattrain<-read.table("train_balance-scale.0", sep=" ")
fit<-pomfit(dattrain[,-ncol(dattrain)],as.factor(dattrain[,ncol(dattrain)]),"logistic")
```

5.6 pompredict

5.6.1 Tittle

Predict over the new data instances using the trained model.

5.6.2 Use

```
pompredict(model, test)
```

5.6.3 Arguments

- **model**: A trained POM model.
- **test**: Numeric test data without labels.

5.6.4 Return

A list containing at the first position the projected values per instance per class and at the second position the predicted label for the values.

5.6.5 Description

Predict over the new data instances using the trained model.

5.6.6 Examples

```
dattrain<-read.table("train_balance-scale.0", sep=" ")
fit<-pomfit(dattrain[,-ncol(dattrain)],as.factor(dattrain[,ncol(dattrain)]),"logistic")
dattest<-read.table("test_balance-scale.0", sep=" ")
```

```
predictions<-pompredict(fit,dattest[,-ncol(dattest)])
projections<-predictions[[1]]
predictedLabels<-predictions[[2]]
```

5.7 svmofit

5.7.1 Tittle

Train n-1 SVM models for ordinal data with given parameters.

5.7.2 Use

```
svmofit(train, trainLabels, weights = TRUE, cost, gamma)
```

5.7.3 Arguments

- **weights**: A boolean indicating whether weights per instance are used.
- **cost**: numeric value indicating the cost parameter to train the SVM.
- **gamma**:numeric value indicating the gamma parameter to train the SVM.
- **train**: Training data of numeric type without labels.
- **trainLabels**: A vector of numeric tags for each instance of training data.

5.7.4 Return

A matrix of 1xn svm trained with weights models.

5.7.5 Description

Train data must be the data without labels. Labels should be provided in trainLabels

5.7.6 Examples

```
dattrain<-read.table("train_balance-scale.0", sep=" ")
modelstrain<-svmofit(dattrain[,-ncol(dattrain)],dattrain[,ncol(dattrain)],TRUE,1,1)
```

5.8 svmopredict

5.8.1 Tittle

Predict over the new data instances using the trained models.

5.8.2 Use

```
svmopredict(models, test)
```

5.8.3 Arguments

- **models**: A matrix of 1xN trained SVM models. Where N denotes the number of classes of the problem minus one.
- **test**: Numeric test data without labels.

5.8.4 Return

A list containing the projected values per instance per class and the predicted values (the maximum probability for each data instance).

5.8.5 Description

Predict over the new data instances using the trained models.

5.8.6 Examples

```
dattrain<-read.table("train_balance-scale.0", sep=" ")
modelstrain<-svmofit(dattrain[,-ncol(dattrain)],dattrain[,ncol(dattrain)],TRUE,1,1)
dattest<-read.table("test_balance-scale.0", sep=" ")
predictions<-svmopredict(modelstrain,dattest[,-ncol(dattest)])
```

5.9 wknnor

5.9.1 Tittle

K-nearest neighbors for ordinal and monotonic data

5.9.2 Use

```
wknnor(traindata, trainlabels, testdata, k, q, kerneltype, monotonicity)
```

5.9.3 Arguments

- **traindata**: Training data of numeric type without labels.
- **trainlabels**: A vector of numeric tags for each instance of training data.
- **testdata**: Test data of numeric type.
- **q**: Minkowski distance param. Use q=1 for Manhattan distance and q=2 for Euclidean distance.
- **kerneltype**: Kernel used to compute neighbors weights. Avaiable kernels are: rectangular, triangular,epanechnikov, biweight, triweight, cosine, gauss and inversion.
- **monotonicity**: Boolean param specifying whether data is monotone or not.

5.9.4 Return

Predicted labels for test data.

5.9.5 Description

Predicts labels for test data with ordinal nature or monotonic constraints using an adaptation of wknn.

5.9.6 Examples

```
dattrain<-read.table("train_balance-scale.0", sep=" ")
traindata=dattrain[,-ncol(dattrain)]
trainlabels=dattrain[,ncol(dattrain)]
testdata<-read.table("test_balance-scale.0", sep=" ")
testdata<-testdata[,-ncol(testdata)]
testlabels<-wknnor(traindata,trainlabels,testdata,5,2,"rectangular",FALSE)
```

6

Resultados, conclusiones y vías futuras

6.1 Resultados experimentales

En esta sección se muestran los experimentos realizados con 10 datasets ordinales distintos, emplemando como medidas de evaluación el error absoluto medio y la tasa de error. Para realizar las experimentaciones de forma general se aplica la siguiente configuración no óptima de parámetros para los algoritmos:

- SVM: coste =10,gamma=0.7
- POM: núcleo logístico
- KDLOR: núcleo RBF con parámetros 10, 0.001 y 1.
- WKNNOR: núcleo rectangular, cinco vecinos, distancia euclídea
- FSelector: k y β =2, selecciona la mitad de características del dataset.

- ISelector: candidatos=0.01,colisiones=0.02, kedition=5

Las especificaciones de los conjuntos de datos empleados se resumen en la siguiente tabla:

Dataset	Instancias	Atributos	Clases
balance-scale	625	4	3
winequality-red	1599	11	6
SWD	1000	10	4
contact-lenses	24	6	3
toy	300	2	5
ESL	488	4	9
LEV	1000	4	5
Automobile	205	71	6
Pasture	36	25	3
Squash-stored	52	51	3

6.1.1 Dataset balance-scale

Resultados base de los algoritmos aplicados al dataset:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.840764	0.1019108	0.1656051	0.4076433
MZE	0.955414	0.089172	0.1656051	0.2484076

Resultado de aplicar selección de características como paso previo, seleccionando 2/4 características:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.038217	0.5159236	0.477707	0.8089172
MZE	0.5605096	0.2993631	0.3375796	0.4458599

Resultados de aplicar se selección de instancias:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.700637	0.1910828	0.4522293	0.4458599
MZE	0.8917197	0.1401274	0.433121	0.2675159

Observamos en la tabla de resultados base que los algoritmos POM y KDLOR obtienen resultados en la línea de sus implementaciones matlab originales realizadas por P.A.Gutiérrez [15], resultando en un peor rendimiento de los algoritmos cuando se les aplica alguna de las dos técnicas de preprocesamiento implementadas. WKNNOR con núcleo rectangular empeora su rendimiento ligeramente tras aplicarle selección de instancias, mientras que lo empeora significativamente más si su lugar, preprocesamos con selección de características. En cualquier caso, obtiene una tasa de error inicial no muy diferente a la obtenida en el algoritmo original que lo implementa [12], empleando un dataset de dimensiones muy similares. En cuanto a los resultados obtenidos por SVMOP, se alejan mucho de los obtenidos en [15] para este dataset, lo que puede deberse a un error en la implementación o a que los parámetros coste y gamma de la SVM empleados no son los óptimos para este problema.

6.1.2 Dataset Wine-quality red

Resultados base de aplicar las técnicas de clasificación desarrolladas al dataset:

	SVMOP	POM	KDLOR	WKNNOR
MAE	2.6375	0.4425	0.51	2.635
MZE	0.995	0.4025	0.46	0.995

Aplicando selección de características (5/11):

	SVMOP	POM	KDLOR	WKNNOR
MAE	2.5475	0.5525	0.95	2.635
MZE	0.995	0.48	0.6275	0.995

Aplicando selección de Instancias (1195 de 1199):

	SVMOP	POM	KDLOR	WKNNOR
MAE	2.6125	0.44	0.5075	2.635
MZE	0.99	0.4025	0.4575	0.995

De nuevo observamos el mal rendiemiento de la SVM ordinal, a diferencia del paper original[31] donde obtiene un error absoluto medio de entre 0.38-0.49 para datasets reales. El error absoluto medio y la tasa de error del resto de técnicas se mantiene similar a sus implementaciones originales para POM y KDLOR[15 25], mejorando en el caso de POM ligeramente el MZE (0.4025 frente a 0.40789) y KDLOR frente al caso base cuando se preprocesan los datos con el selector de instancias. El resto de algoritmos no se ven afectados por la selección de instancias, mientras que la selección de la mitad de características empeora el rendimiento en POM y KDLOR, aunque quizás no sería así eligiendo un número de características a seleccionar más óptimo. Por último el KNN ordinal obtiene una tasa de error muy superior a la que obtiene la implementación original con un dataset con igual número de atributos[12], si bien el dataset no era el mismo y en este caso el número de instancias era el doble (1199 frente a 546). Esto quizás podría paliarse con la utilización de otros parámetros en WKNNOR y otro tipo de kernel.

6.1.3 Dataset SWD

Resultados base de la ejecución de los algoritmos de clasificación:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.82	0.48	0.508	1.324
MZE	0.976	0.464	0.456	0.84

Aplicando selección de características como preprocesamiento:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.852	0.54	0.564	1.204
MZE	0.98	0.488	0.504	0.82

Aplicando selección de instancias como preprocesamiento:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.532	0.476	0.492	1.28
MZE	0.88	0.464	0.448	0.812

En este caso, los resultados base para KDLOR de MAE y MZE son mejores que los obtenidos en el experimento de [15], donde para el mismo dataset obtienen MAE y MZE igual a 0.5785 y 0.5137, respectivamente. Para POM son ligeramente inferiores (0.45 y 0.43 en el original), lo que puede deberse a una distinta función kernel empleada. WKNNOR no da buenos resultados, si bien no es comparable con los resultados del trabajo original [12] dado que los autores no tienen resultados para datasets de más de 600 instancias. No obstante, este rendimiento puede mejorarse usando otros parámetros como cambiando el número de vecinos o el tipo de núcleo empleado.

En cuanto a las técnicas de preprocesamiento empleadas, podemos decir que la selección de características solo mejora muy levemente a WKNN mientras que las otras no parecen beneficiarse, mientras que la selección de instancias sí parece beneficiar levemente a los cuatro algoritmos en este caso.

6.1.4 Dataset Contact-lenses

Resultados base de los algoritmos de clasificación implementados:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.5	-	0.5	0.5
MZE	0.8333333	-	0.5	0.3333333

Resultados de aplicar selección de características previamente:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.5	0.5	0.8333333	0.5
MZE	0.8333333	0.333333	0.8333333	0.3333333

Resultados de aplicar selección de instancias previamente:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.5	1	0.5	0.5
MZE	0.8333333	0.5	0.8333333	0.3333333

El empleo de todos los atributos inicialmente hace que POM no converga, lo que se soluciona posteriormente con una selección de características. EL MAE base de KDLOR es ligeramente inferior al obtenido en la implementación en la que se basa ésta (0.5167)[15], sin embargo el valor de MZE es superior (0.5 frente a 0.33). Seguro que puede arreglarse usando otro núcleo y otros parámetros que no sean los de por defecto. Para este dataset parece que el rendimiento de WKNNOR es mejor, de hecho en la implementación original[12] con un dataset de seis atributos como este y más características, obtienen una tasa de error de 51.8 % frente a 33% en este caso, si bien los problemas no son exactamente los mismos. En cuanto a los algoritmos de preprocesamiento, sólo se beneficia de la selección de características POM, mientras que ninguno de beneficia de la selección de instancias.

6.1.5 Dataset Toy

Resultados base de los clasificadores:

	SVMOP	POM	KDLOR	WKNNOR + rectangular
MAE	2.86667	0.88	0.1466667	1.933333
MZE	1	0.6666667	0.1466667	0.8933333

Aplicando selector de instancias:

	SVMOP	POM	KDLOR	WKNNOR
MAE	2.626667	1.12	0.5466667	1.88
MZE	0.9866667	0.64	0.4266667	0.88

En este caso, al tratarse de un dataset con dos atributos no merecía la pena considerar una selección de características. Se observa en los resultados base que el único algoritmo que funciona bien inicialmente con este dataset es KDLOR, obteniendo resultados ligeramente superiores de MZE y MAE que en la implementación original de ORCA[15] (0.107 y 0.114 respectivamente). Sin embargo, todos los algoritmos excepto este mejoran su rendimiento algo cuando se aplica selección de instancias previamente a la clasificación.

6.1.6 Dataset ESL

Resultados base de los algoritmos:

	SVMOP	POM	KDLOR	WKNNOR
MAE	4.204918	0.3606557	0.3934426	1.803279
MZE	1	0.3278689	0.352459	0.8852459

Resultados de aplicar selección de características:

	SVMOP	POM	KDLOR	WKNNOR
MAE	4.270492	0.5	0.5163934	1.803279
MZE	1	0.442623	0.4590164	0.7622951

Resultados de aplicar selección de instancias:

	SVMOP	POM	KDLOR	WKNNOR
MAE	3.45082	0.4262295	0.647541	2.52459
MZE	0.9918033	0.3934426	0.5819672	0.9508197

Los resultados base de KDLOR son similares a los del trabajo de referencia[15], al igual que los de POM. WKNNOR obtiene unas medidas de error muy altas para este problema de 9 clases, sin embargo en el tratajo de referencia[12] no se ejemplifica el funcionamiento del algoritmo para un problema de 9 clases. La selección de características parece que beneficia a WKNNOR, mientras que la selección de instancias sólo beneficia sutilmente a SVM.

6.1.7 Dataset LEV

Resultados base de los algoritmos de clasificación:

	SVMOP	POM	KDLOR	WKNNOR
MAE	2.276	0.412	0.484	1.44
MZE	0.98	0.376	0.42	0.784

Con selección de características:

	SVMOP	POM	KDLOR	WKNNOR
MAE	2.212	0.584	0.7	1.412
MZE	0.972	0.512	0.572	0.804

Aplicando selector de instancias:

	SVMOP	POM	KDLOR	WKNNOR
MAE	2.236	0.44	0.48	1.396
MZE	0.976	0.388	0.412	0.744

Para POM y KDLOR de nuevo obtenemos resultados muy similares a los de el tabajo de referencia[15]. A pesar de que el resultado de WKNNOR no es bueno inicialmente, no lo podemos comparar con el trabajo original[12] dado que no muestran resultados para datasets con 1000 instancias. Aplicar el algoritmo de extracción de características a estos datos repercute en una mejora en WKNNOR, mientras que la selección de instancias sólo mejora ligeramente la SVM.

6.1.8 Dataset Automobile

Resultados base para los algoritmos de clasificiación:

	SVMOP	POM	KDLOR	WKNNOR
MAE	2.826923	-	1.019231	2.826923
MZE	0.9807692	-	0.7307692	0.9807692

Selector de características (35 de 71):

	SVMOP	POM	KDLOR	WKNNOR
MAE	2.826923	1.134615	0.9807692	2.826923
MZE	0.9807692	0.7692308	0.7115385	0.9807692

En este caso, con los parámetros por defecto el selector de instancias sólo eliminaba una muestra, por lo que los resultados no fueron relevantes. Automobile es un dataset complicado de 71 atributos y 6 clases, por lo que los resultados no son prometedores para ninguno de los algoritmos. Sin embargo, sí que es útil experimentar con este dataset para comprobar que la selección de características puede ser crucial en problemas donde tenemos muchas, como este, pues a pesar de que hemos eliminado sólo la mitad por una prueba, claramente POM y KDLOR mejoran sus resultados gracias a ella.

6.1.9 Dataset Pasture

Resultados base de los clasificadores:

	SVMOP	POM	KDLOR	WKNNOR + rectangular
MAE	1	-	0.6666667	1
MZE	0.6666667	-	0.6666667	0.6666667

Con Selección de características (7 de 25):

	SVMOP	POM	KDLOR	WKNNOR
MAE	1	0.2222222	0.6666667	1
MZE	0.6666667	0.2222222	0.6666667	0.6666667

Puesto que el problema tiene sólo 27 instancias no se aplica selección de instancias. Inicialmente es necesario aplicar selección de características para aplicar POM, tras lo que obtenemos un MAE y MZE de 0.22, mucho mejor que la media de 0.58 y 0.50 respectivamente que obtiene el estudio de referencia [15]. Sin embargo los valores obtenidos para KDLOR sí son muy inferiores a los del estudio de referencia (0.34 y 0.32 respectivamente), quizás por los parámetros y el tipo de núcleo usados. WKNN también proporciona peores resultados que los del trabajo original para dataset de dimensiones similares, pero pordría arreglarse probando con otros parámetros dado que WKNN es muy sensible a la configuración de parámetros.

6.1.10 Dataset Squash-stored

Resultados base del clasificador:

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.230769	-	0.5384615	0.7692308
MZE	0.8461538	-	0.5384615	0.6153846

Con selección de características (8 de 51):

	SVMOP	POM	KDLOR	WKNNOR
MAE	1.230769	0.3846154	0.5384615	0.7692308
MZE	0.8461538	0.3076923	0.5384615	0.6153846

Tras ejecutar selección de características, KDLOR obtiene una MZE=0.53, inferior al obtenido en el estudio de referencia (0.39), al igual que el MAE (0.53 frente a 0.374). POM sin embargo da unos resultados mucho mejores, con un MZE=0.3076 frente a 0.6179 y MAE= 0.3846 frente a 0.81 en el trabajo de referencia [15]. En este caso no se elimina ninguna instancia al realizar la selección de instancias, dada la configuración de parámetros por defecto.

Análisis de parámetros

En esta sección se muestran los resultados de lanzar los algoritmos usando el dataset de referencia **Balance Dataset** probando con distintos tipos de núcleo para los algoritmos de clasificación:

KDLOR con distintos kernels:

	rbf	gauss	lineal	poly
MAE	0.1656051	0.1656051	0.2484076	0.2484076
MZE	0.8461538	0.8461538	0.2484076	0.2484076

POM con distintos kernels:

	logistic	probit	loglog	cloglog	cauchit
MAE	0.1019108	0.1210191	0.1592357	0.2101911	0.1082803
MZE	0.089172	0.1082803	0.1210191	0.1464968	0.0955414

WKNNOR con distintos kernels:

	rectangular	triangular	epanechnikov	biweight	triweight	cosine	inversion
MAE	0.4076433	0.388535	0.388535	0.3821656	0.3821656	0.388535	0.1019108
MZE	0.2484076	0.2292994	0.2292994	0.2229299	0.2229299	0.2292994	0.089172

6.2 Conclusiones y vías futuras

6.2.1 Conclusiones

POM y KDLOR dan unos resultados que, en general no se alejan del estudio de referencia [15], si bien como se puede ver en la sección anterior, puede variar mucho el resultado de estos cambiando simplemente el tipo de kernel que emplean. Aunque no se han empleado los mismos

datasets que en la propuesta de KDLOR original[12] se ha comprobado que, para datasets con dimensiones similares a veces da resultados cercanos a los originales y a veces no. Esto puede deberse simplemente a una configuración de parámetros, pues WKNN es un método muy sensible al número de vecinos empleado (en este caso, el de por defecto, 5), a la distancia empleada y al núcleo usado para calcular los pesos. De hecho, en la sección anterior podemos ver con el dataset **balance** como para WKNNOR simplemente el cambio del tipo de kernel puede resultar en una mejora del 16% de la tasa de error.

En cuanto a SVMOP, los resultados exigen una revisión del algoritmo, en términos de implementación y de parámetros. Sobre los algoritmos de preprocesamiento, no se puede concluir que sean o no útiles, ni tampoco que uno lo sea más que otro, pues a lo largo de la experimentación, ambos han servido para mejorar el rendimiento del clasificador en algunos casos y empeorarlo en otro. Podría decirse pues, que depende de los datos y de la naturaleza del problema su aplicación.

6.2.2 Vías futuras

Esta es la primera versión de OCAPIS y como tal, hay mucho que hacer aún sobre ella. Queda pendiente por tanto, mejorar la SVM para datos ordinales, implementar la selección de instancias de forma inmutable y funcional en lugar de iterativa, lo que tiene un impacto directo en la eficiencia y velocidad del paquete. También queda pendiente añadir más algoritmos a este paquete para clasificación ordinal, como SVOR en sus variantes SVOREX, SVORIM, ORBoost y una red neuronal.

References

[1] Baccianella, S., Esuli, A., & Sebastiani, F. (2014). Feature selection for ordinal text classification. *Neural computation*, 26(3), 557–591.

[2] Ben-David, A. (1995). Monotonicity maintenance in information-theoretic machine learning algorithms. *Machine Learning*, 19(1), 29–43.

[3] Ben-David, A., Sterling, L., & Pao, Y.-H. (1989). Learning and classification of monotonic ordinal concepts. *Computational Intelligence*, 5(1), 45–49.

[4] Bender, R. & Grouven, U. (1997). Ordinal logistic regression in medical research. *Journal of the Royal College of physicians of London*, 31(5), 546–551.

[5] Cano, J.-R. & García, S. (2017). Training set selection for monotonic ordinal classification. *Data & Knowledge Engineering*, 112, 94–105.

[6] Chang, C.-C. & Lin, C.-J. (2011). Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3), 27.

[7] Chang, K.-Y., Chen, C.-S., & Hung, Y.-P. (2011). Ordinal hyperplanes ranker with cost sensitivities for age estimation. In *Computer vision and pattern recognition (cvpr), 2011 ieee conference on* (pp. 585–592).: IEEE.

[8] Cheng, J., Wang, Z., & Pollastri, G. (2008). A neural network approach to ordinal regression. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on* (pp. 1279–1284).: IEEE.

[9] Chu, W. & Ghahramani, Z. (2005). Gaussian processes for ordinal regression. *Journal of machine learning research*, 6(Jul), 1019–1041.

[10] Chu, W. & Keerthi, S. S. (2007). Support vector ordinal regression. *Neural computation*, 19(3), 792–815.

[Dahl] Dahl, D. B. Integration of r and scala using rscala.

[12] Duivesteijn, W. & Feelders, A. (2008). Nearest neighbour classification with monotonicity constraints. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (pp. 301–316).: Springer.

[13] Frank, E. & Hall, M. (2001a). A simple approach to ordinal classification.

[14] Frank, E. & Hall, M. (2001b). A simple approach to ordinal classification. In *European Conference on Machine Learning* (pp. 145–156).: Springer.

[15] Gutiérrez, P. A., Pérez-Ortiz, M., Sánchez-Monedero, J., Fernández-Navarro, F., & Hervás-Martínez, C. (2016). Ordinal regression methods: Survey and experimental study. *IEEE Transactions on Knowledge and Data Engineering*, 28(1), 127–146.

[16] Hechenbichler, K. & Schliep, K. (2004). Weighted k-nearest-neighbor techniques and ordinal classification.

[17] Hu, Q., Pan, W., Zhang, L., Zhang, D., Song, Y., Guo, M., & Yu, D. (2012). Feature selection for monotonic classification. *IEEE Transactions on Fuzzy Systems*, 20(1), 69–81.

[18] Kim, M. & Pavlovic, V. (2010). Structured output ordinal regression for dynamic facial emotion intensity prediction. In *European conference on computer vision* (pp. 649–662).: Springer.

[19] Kwon, Y. S., Han, I., & Lee, K. C. (1997). Ordinal pairwise partitioning (opp) approach to neural networks training in bond rating. *Intelligent Systems in Accounting, Finance & Management*, 6(1), 23–40.

[20] Lievens, S., De Baets, B., & Cao-Van, K. (2008). A probabilistic framework for the design of instance-based supervised ranking algorithms in an ordinal setting. *Annals of Operations Research*, 163(1), 115–142.

[21] Lin, H.-T. & Li, L. (2006). Large-margin thresholded ensembles for ordinal regression: Theory and practice. In *International Conference on Algorithmic Learning Theory* (pp. 319–333).: Springer.

[22] Mathieson, M. J. (1996). Ordinal models for neural networks. In *Proc. 3rd Int. Conf. Neural Netw. Capital Markets* (pp. 523–536).

[23] McCullagh, P. (1980). Regression models for ordinal data. *Journal of the royal statistical society. Series B (Methodological)*, (pp. 109–142).

[24] Pazzani, M. J., Mani, S., & Shankle, W. R. (2001). Acceptance of rules generated by machine learning among medical experts. *Methods of information in medicine*, 40(05), 380–385.

[25] Sun, B.-Y., Li, J., Wu, D. D., Zhang, X.-M., & Li, W.-B. (2010). Kernel discriminant learning for ordinal regression. *IEEE Transactions on Knowledge and Data Engineering*, 22(6), 906–910.

[26] Sánchez-Monedero, J., Gutiérrez, P. A., Tino, P., & Martínez, C. (2013). Exploitation of pairwise class distances for ordinal classification. 25.

[27] Tian, Q., Chen, S., & Tan, X. (2014). Comparative study among three strategies of incorporating spatial structures to ordinal image regression. *Neurocomputing*, 136, 152–161.

[28] Torra, V., Domingo-Ferrer, J., Maria Mateo-Sanz, J., & Ng, M. (2006). Regression for ordinal variables without underlying continuous variables. 176, 465–474.

[29] tu, h.-h. & Lin, H.-T. (2010). One-sided support vector regression for multiclass cost-sensitive classification.

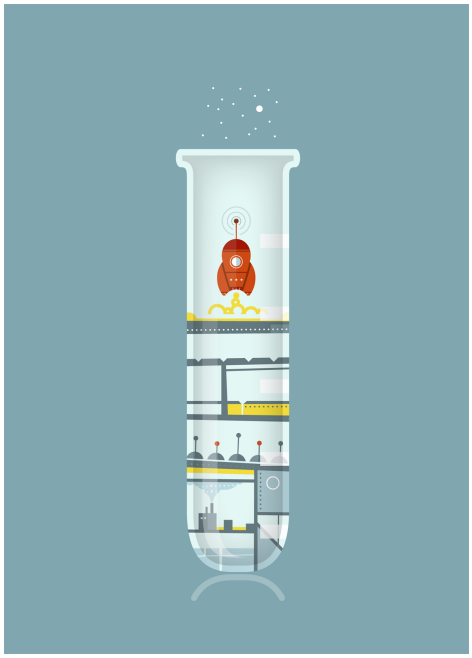
[30] Vapnik, V. N. & Chervonenkis, A. Y. (2015). *On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities*, (pp. 11–30). Springer International Publishing: Cham.

[31] Waegeman, W. & Boullart, L. (2009). An ensemble of weighted support vector machines for ordinal regression. *International Journal of Computer Systems Science and Engineering*, 3(1), 47–51.

[32] Wampler, D. & Payne, A. (2014). *Programming Scala: Scalability = Functional Programming + Objects*. O'Reilly Media.

[33] Yan, H. (2014). Cost-sensitive ordinal regression for fully automatic facial beauty assessment. *Neurocomputing*, 129, 334–342.

[34] Yoon, J. W., Roberts, S. J., Dyson, M., & Gan, J. Q. (2011). Bayesian inference for an adaptive ordered probit model: An application to brain computer interfacing. *Neural Networks*, 24(7), 726–734.



This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. The above illustration, *Science Experiment 02*, was created by Ben Schlitter and released under CC BY-NC-ND 3.0. A template that can be used to format a PhD dissertation with this look & feel has been released under the permissive AGPL license, and can be found online at github.com/suchow/Dissertate or from its lead author, Jordan Suchow, at suchow@post.harvard.edu.