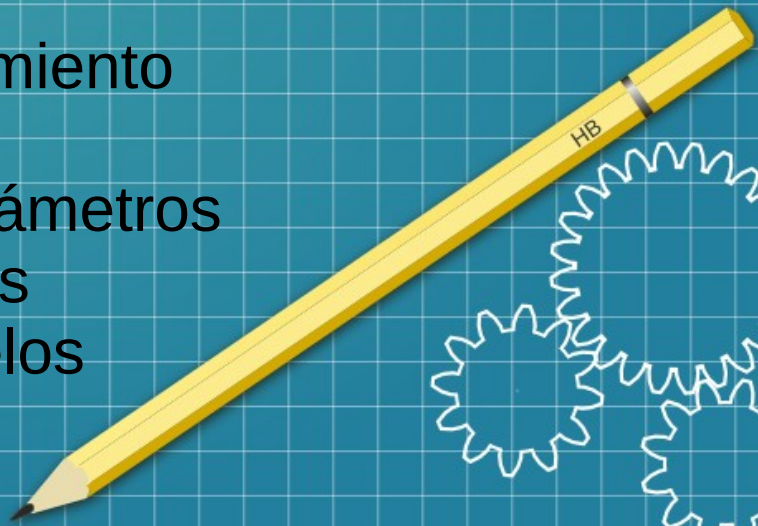
De EDA, CV y otras siglas...

MENÚ

* Análisis Exploratorio
* Tipos de datos
* Preprocesamiento
* Pipelines
* Métricas para evaluar rendimiento
* Cross Validation
* Búsqueda de los mejores parámetros
* Clases desbalanceadas
* Combinaciones de modelos



Análisis Exploratorio de Datos (EDA)

- Lo primero que necesitamos es conocer nuestro dataset...
- Usamos librerías como Pandas (pd), Numpy (np) y Seaborn (sns)
- Siempre que sea posible, conviene visualizar en gráficos
- Instrucciones más usadas:

  - pd.read_csv() ó read_excel() ó read_html()…

  - df.shape, df.columns

  - df.info(), df.describe()

  - df.isnull.sum()

- Para visualizar: sns.pairplot(df) o más paquete: sns.pairplot(df, kind='kde') -» https://towardsdatascience.com/seaborn-pairplot-enhance-your-data-understanding-with-a-single-plot-bf2f44524b22

Tipos de datos - 1

- NUMÉRICOS: enteros o flotantes. La mayoría de los algoritmos sólo manejan este tipo
- CATEGÓRICOS: no numéricos, identifican categorías o etiquetas. Tenemos de 2 tipos:

  * ORDINALES: pueden ser ordenados o clasificados (x ej talles como S, M, L)

  * NOMINALES: no implican ningún orden (x ej, colores)

TIPOS DE DATOS - 2

- Cuando tenemos datos ordinales, hay que convertirlos a valores enteros. No hay una función específica. Puede ser algo así:

  » talles ={‘xl’:3, ‘l’:2, ‘m’:1, ‘s’:0}

  » df[‘size’]= df[‘size’].map(talles)

- Si tenemos etiquetas de clase en forma categórica (‘moroso’, ‘no moroso’), aunque el algoritmo pueda manejarlas, se considera buena práctica ingresar las etiquetas de clase como matrices de números enteros
- Para ello, Scikit Learn tiene varios ‘encoders’:

  * LabelEncoder

  * OneHotEncoder (en Pandas, está pd.get_dummies())

  Ej »from sklearn.preprocessing import LabelEncoder

  »clase= LabelEncoder()

  »y = clase.fit_transform(df[‘etiquetas’].values)

  Existe el método inverso: clase.inverse_transform(y)

PREPROCESAMIENTO - 1

- Casi siempre tendremos que tomar decisiones sobre nuestro dataset antes de aplicar los modelos de aprendizaje automático:

  @ Escalar las características

  @ Eliminar o imputar valores ausentes

  @ Dar forma a datos categóricos para que el modelo los comprenda

  @ Seleccionar características importantes para construir el modelo

Preprocesamiento - 2

- **Escalamiento**: La mayoría de los algoritmos de AA funcionan mucho mejor si las características están en la misma escala. Sin embargo, algunos como los basados en árboles de decisión que no lo necesitan. En general se considera una BUENA PRÁCTICA.
- Hay dos enfoques:

  * NORMALIZACIÓN

  * ESTANDARIZACIÓN
- La NORMALIZACIÓN consiste en el re-escalado de las características dentro de un rango: [0...1], [min...max]. Se aplica la sgte expresión:

  Xnorm = (X – Xmin) / (Xmax – Xmin)

  »from sklearn.preprocessing import MinMaxScaler

  »mms=MinMaxScaler()

  »X_norm= mms.fit_transform(X)

Preprocesamiento - 3

- La ESTANDARIZACIÓN puede ser más conveniente para modelos que usan algoritmos del descenso del gradiente, porque facilita la convergencia del mismo.
- Además, asegura que la distribución de las características seguirá una normal (gaussiana) con media nula y desviación estandar igual a 1
- $X\_std = (X - \mu)/\sigma$
- »from sklearn.preprocessing import StandardScaler

Preprocesamiento – 4

- **Tratar con valores ausentes**: depende del problema que estemos analizando. NaN: Not a Number, Null: indicador de datos desconocidos o espacio en blanco
- Si se eliminan, se puede perder información importante.

  DataFrame.dropna(*, axis=0, how=_NoDefault.no_default, thresh=_NoDefault.no_default, subset=None, inplace=False)
- Pero hay que ser cuidadoso con la imputación que se haga. Algunos métodos:

  * *Interpolación entre vecinos*:

  DataFrame.interpolate(method='linear', *, axis=0, limit=None, inplace=False, limit_direction=None, limit_area=None, downcast=None, **kwargs)

  * *Promedio de característica*

  »from sklearn.preprocessing import Imputer

  »imput=Imputer(missing_values= ‘NaN’, strategy= ‘mean’, axis =0)

  »imput=imput.fit(df.values)

  »imp_data= imput.transform(df.values)

  También puede usar el most_frequent o la median

  * *Otros valores ‘a mano’*

Preprocesamiento - 5

- En Tipos de Datos vimos cómo se **codifica a los datos categóricos**
- **Selección de las características más relevantes**: hay distintos enfoques:

  * ‘A mano’ según el problema de negocio

  * Distintos algoritmos tienen un método .feature_importance_

  * sklearn y mlxtend tienen feature_selection

  * existen algoritmos como LDA y PCA

- La selección de características es un tema muy amplio y no voy a extenderme más por ahora

PIPELINES

- Sirven para simplificar los flujos de trabajo
- Se puede ajustar un modelo incluyendo cualquier número de pasos de transformación y aplicarlo para hacer predicciones sobre nuevos datos -» fit y predict
- Se pueden poner transformadores y estimadores pero el último elemento debe ser un estimador
- Ejemplo: estandarizamos el dataset y aplicamos regresión logística (el dataset ya fue trabajado y dividido en Train y Test)

```
»from sklearn.preprocessing import StandardScaler
»from sklearn.pipeline import make_pipeline
»from sklearn.linear_model import LogisticRegression
»pipe=make_pipeline(StandardScaler(),
                    LogisticRegression(random_state=1) )
»pipe.fit(X_train, y_train)
»y_pred=pipe.predict(X_test)
```

MÉTRICAS PARA EVALUAR RENDIMIENTO

- Para disponer de alguna manera de evaluar el rendimiento del modelo de aprendizaje automático (y poder compararlo para cada modelo), se definen MÉTRICAS
- Cabe destacar que para cada lógica de negocio podemos definir una métrica, lo cual puede tener mucho sentido. Pero no siempre reflejará lo bien o mal que funciona el algoritmo de AA con nuestros datos. Por ello se definen ciertas métricas ‘estandarizadas’. Sin embargo, nunca hay que olvidar que estamos tratando de ajustar un modelo a la ‘realidad’ y ésta es la que manda
- Entonces, siempre que se usen las métricas hay que mirarlas dentro de la lógica del problema y el conocimiento que pretendemos extraer de nuestros datos

MÉTRICAS - 1

- Se definen como TP (true positive, verdadero positivo) a la etiqueta de clase predicha ‘1’ que coincide con la etiqueta real, en el conjunto de testeo
- Análogamente, se definen TN (true negative, verdadero negativo) a la etiqueta de clase ‘0’ que coincide con la real…
- FP (false positive, falso positivo) a la etiqueta de clase predicha ‘1’ cuando la real es ‘0’…
- FN (false negative, falso negativo) a la etiqueta de clase predicha ‘0’ cuando la real es ‘1’
- Entonces, TP y TN son **aciertos** del modelo y FP y FN son **errores** del modelo
- Si bien acá estamos pensando en una clasificación binaria (por simplicidad), es muy fácil generalizar la idea a una multiclasificación

MÉTRICAS - 2

- Se define **Exactitud** (accuracy) como la fracción de predicciones correctas hechas por el modelo sobre el total

  accuracy=(TP+TN)/(TP+TN+FP+FN)

- Análogamente: la **Precisión** (precision) es la fracción de identificaciones positivas correctas

  precision=TP/(TP+FP)

- La **Recuperación** o Sensibilidad (recall) es la proporción de positivos reales que se identificó correctamente

  recall=TP/(TP+FN)

- La **Especificidad** (especificity) es la proporción de verdaderos negativos que se identificaron correctamente

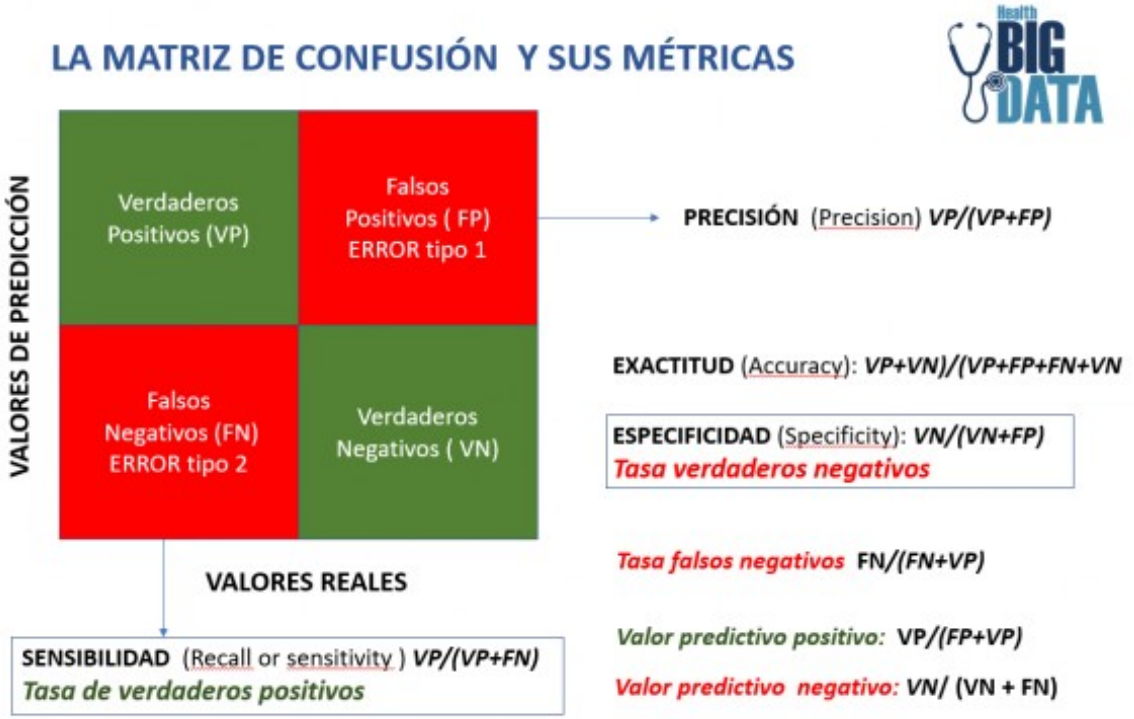
  especificity=TN/(TN+FP)

- Como la precisión y la recuperación son difíciles de conciliar, se define la **Puntuación F1** (F1-score)

  F1=2*precision*recall/(precision+recall) = 2*TP/(2*TP+FP+FN)

MÉTRICAS - 3

- Para tener un panorama más completo, podemos usar la Matriz de Confusión

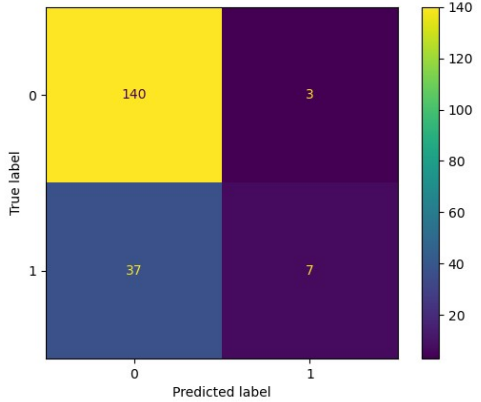


MÉTRICAS - 4

- Entonces, podemos tener cuatro casos posibles para cada clase:
- **Alta precisión y alto recall**: el modelo de AA escogido maneja perfectamente esa clase.
- **Alta precisión y bajo recall**: el modelo de AA escogido no detecta la clase muy bien, pero cuando lo hace es altamente confiable.
- **Baja precisión y alto recall**: El modelo de AA escogido detecta bien la clase, pero también incluye muestras de la otra clase.
- **Baja precisión y bajo recall**: El modelo de AA escogido no logra clasificar la clase correctamente.
- Cuando tenemos un dataset con desequilibrio de clases, suele ocurrir que obtenemos un alto valor de precisión en la clase mayoritaria y un bajo recall en la clase minoritaria, por ello tenemos que recurrir al balanceo de clases.

MÉTRICAS - 5

- En sklearn tenemos todas estas métricas y algunas más...
  »from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
  »metrica=accuracy_score(y_true, y_pred)
- Matriz de confusión y una manera de verla ‘linda’
  »from sklearn.metrics import confusion_matrix
  »from sklearn.metrics import ConfusionMatrixDisplay
  »y_pred=modelo.predict(X_test)
  »cm=confusion.matrix(y_test, y_pred)
  »ConfusionMatrixDisplay(cm).plot()
- Podemos obtener un informe donde se muestren las métricas en texto
  »from sklearn.metrics import classification_report
  »print(classification_report(y_true, y_pred)



	precision	recall	f1-score	support
Iris-setosa	1.00	1.00	1.00	50
Iris-versicolor	0.77	0.96	0.86	50
Iris-virginica	0.95	0.72	0.82	50
avg / total	0.91	0.89	0.89	150

MÉTRICAS - 6

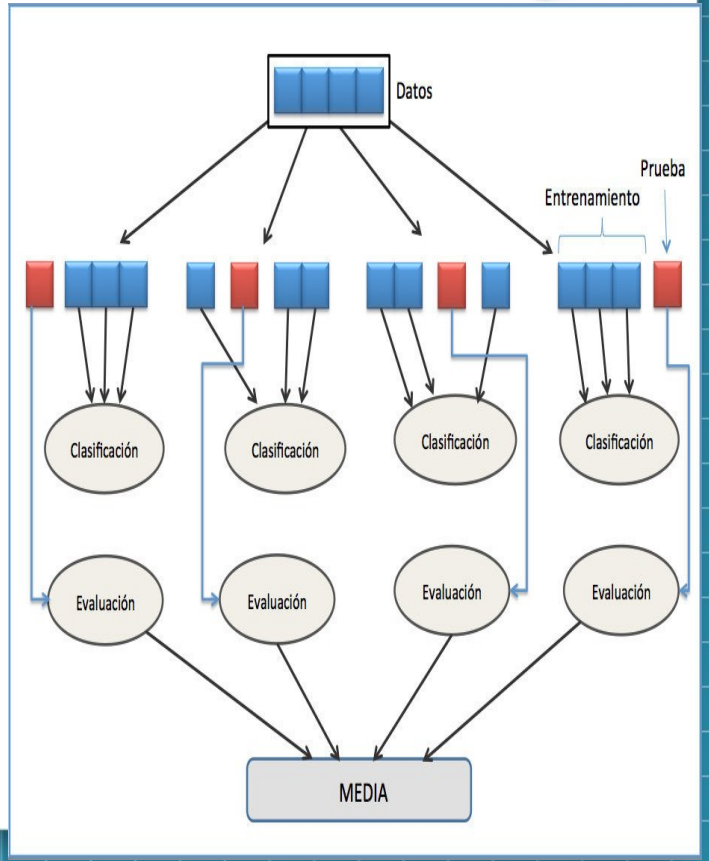
- Finalmente, sklearn también tiene una clase para ‘defina-su-propia-métrica’
- sklearn.metrics.**make_scorer**(score_func, *, greater_is_better=True, needs_proba=False, needs_threshold=False, **kwargs)

  donde score_func es la métrica que deseamos definir
- Más info en la documentación oficial (como siempre)

  https://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html

CROSS VALIDATION - 1

- Es una técnica utilizada para evaluar los resultados de un análisis estadístico y garantizar que son independientes de la partición entre datos de entrenamiento y prueba
- Los resultados sólo son significativos si los cjtos se extraen de la misma población
- Consiste en ajustar y predecir usando el mismo modelo y promediar el rendimiento obtenido de las medidas de evaluación sobre diferentes particiones provenientes del mismo conjunto de entrenamiento
- 2 métodos: con retención y k iteraciones



CROSS VALIDATION -2

- **Con retención**: es el más usado. Se separa el dataset en 2 subconjuntos: *entrenamiento y prueba.* Se ajusta el modelo con el cjto de entrenamiento y se observa el rendimiento sobre el cjto de testeo, repitiendo varias veces el mismo proceso hasta seleccionar los mejores parámetros del modelo. Sin embargo, de esta manera el cjto de testeo acaba por ser parte del entrenamiento y el modelo puede sobreajustarse.

  Para evitar este efecto, conviene dividir el dataset en 3 subconjuntos: *entrenamiento, prueba* y *validación.* El modelo se entrena con el cjto de entrenamiento, se utiliza el de validación para obtener los mejores parámetros y sólo se usa el cjto de testeo para determinar el rendimiento de modelo.

  El inconveniente es que la estimación del rendimiento puede ser muy sensible a cómo se divide el dataset

CROSS VALIDATION - 3

- **K iteraciones (k-fold CV)**: es una técnica más robusta y precisa por la manera de dividir el dataset.

Se separa el dataset en cjto de entrenamiento y cjto de testeo.

Se divide aleatoriamente el cjto de entrenamiento en k subconjuntos **sin reemplazo**. De éstos, k-1 se usan para entrenar el modelo y el restante para evaluar el rendimiento.

El proceso se repite k veces, por lo que se obtienen k modelos con sus parámetros y estimaciones de rendimiento.

Luego se calcula el rendimiento medio de los modelos a partir de las estimaciones independientes.

Finalmente, elegido el modelo con los parámetros que dan el mejor rendimiento, se lo vuelve a entrenar con el cjto de entrenamiento entero y se obtiene una estimación final del modelo con el cjto de testeo independiente.

CROSS VALIDATION - 4

- Como se usa muestreo sin reemplazo, cada registro de muestra se usa UNA SOLA VEZ para entrenar y validar.
- Los rendimientos (la métrica que se decida usar) estimados sobre cada iteración (o fold) se promedian -» $E=\Sigma Ei /k$ (i va de 1 a k)
- Según distintos estudios, un buen valor para k = 10, pues ofrece una mejor compensación entre sesgo y varianza (lo que mejora la calidad de la estimación)
- Para cjtos de datos pequeños, se puede usar un enfoque alternativo: **LOOCV** (validación cruzada dejando uno afuera). Se toma k=n (nro de muestras) y en cada iteración, se deja una sola sobre la cuál se testea. Al final, todas las muestras fueron usadas para testear dentro del cjto de entrenamiento.
- Para clases desbalanceadas, es mejor usar **Validación Cruzada Estratificada de k iteraciones** (k-fold stratified CV), ya que mejora la estimación de sesgo y varianza.

  Se respetan las proporciones de clase en cada iteración, lo que garantiza que siempre se respeten las proporciones del cjto de entrenamiento

CROSS VALIDATION - 5

- En SKLEARN:

```
»from sklearn.model_selection import StratifiedKFold
»kfold=StratifiedKFold(n_splits=10,random_state=1).split(X_train, y_train)
»scores=[]
»for k, (train, test) in enumerate(kfold):
»       model.fit(X_train[train], y_train[train])
»       score=model.score(X_train[test], y_train[test])
»       scores.append(score)
```

- Existe un score para CV

```
»from sklearn.model_selection import cross_val_score
»scores=cross_val_score(estimator=model, X=X_train, y=y_train, cv=10, n_jobs=-1)
```

- Se pueden usar dentro de un pipeline también

BÚSQUEDA DE LOS MEJORES PARÁMETROS - 1

- En Sklearn existen dos clases para ajustar modelos de AA, buscando los mejores hiperparámetros
- El algoritmo hace una búsqueda sobre diferentes valores para los hiperparámetros y luego el método evalúa el rendimiento del modelo hasta encontrar la combinación óptima de valores para ese dataset
- Una vez obtenidos los mejores parámetros, se usa el cjto de testeo para estimar el rendimiento del mejor modelo seleccionado
- Una de las clases es GridSearch (búsqueda de cuadrículas), que hace una búsqueda exhaustiva de fuerza bruta dentro del rango indicado. Es muy costosa computacionalmente
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
- class sklearn.model_selection.GridSearchCV(estimator, param_grid, *, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', error_score=nan, return_train_score=False)
- Se indica el modelo, los rangos de valores para los parámetros, scoring, si se usa cross validation entre otros. Tiene métodos fit y predict

BÚSQUEDA MEJORES PARÁMETROS - 2

Ej: »pipe=make_pipeline(StandardScaler(), SVC(random_state=1))

»param_range=[0.0001, 0.001, 0.01, 0.1, 1.0, 10., 100., 1000.]

»param_grid=[{‘svc_c’: param_range, ‘svc_kernel’: [‘linear’]},

{‘svc_c’: param_range, ‘svc_gamma’: param_range, ‘svc_kernel’:[‘rbf’]}]

»gs=GridSearchCV(estimator=pipe, param_grid=param_grid,scoring=’accuracy’,cv=10, n_jobs=-1)

»gs=gs.fit(X_train, y_train)

»print(gs.best_score)

»print(gs.best_params)

»clf=gs.best_estimator_

»clf.fit(X_train, y_train)

»print(‘Test accuracy’, clf.score(X_test, y_test))

BÚSQUEDA MEJORES PARÁMETROS - 3

- El otro enfoque es una búsqueda randomizada, es decir, a diferencia de GridSearchCV, no se prueban todos los valores de los parámetros, sino que se muestrea un número fijo de configuraciones de parámetros de las distribuciones especificadas. El número de configuraciones de parámetros que se prueban viene dado por n_iter
- Si todos los parámetros se presentan como una lista, se realiza un muestreo sin reemplazo. Si se da al menos un parámetro como distribución, se utiliza el muestreo con reemplazo. Se recomienda encarecidamente utilizar distribuciones continuas para parámetros continuos
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html
- class sklearn.model_selection.RandomizedSearchCV(estimator, param_distributions, *, n_iter=10, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', random_state=None, error_score=nan, return_train_score=False)

CLASES DESBALANCEADAS

AJuste de Parámetros del modelo: Consiste en ajustar parametros ó metricas del propio algoritmo para intentar equilibrar a la clase minoritaria penalizando a la clase mayoritaria durante el entrenamiento. Ejemplos: ajuste de peso en árboles, también en logistic regression tenemos el parámetro class_weight= “balanced”. No todos los algoritmos tienen estas posibilidades. En redes neuronales por ejemplo podríamos ajustar la métrica de Loss para que penalice a las clases mayoritarias.
Modificar el Dataset: OVERSAMPLING Ó UNDERSAMPLING: podemos eliminar muestras de la clase mayoritaria para reducirlo e intentar equilibrar la situación. Tiene como “peligroso” que podemos prescindir de muestras importantes, que brindan información y por lo tanto empeorar el modelo. Entonces para seleccionar qué muestras eliminar, deberíamos seguir algún criterio. También podríamos agregar nuevas filas con los mismos valores de las clases minoritarias, por ejemplo repetir filas de la clase minoritaria hasta empatar con las de la clase mayoritaria. Pero esto no sirve demasiado y podemos llevar al modelo a caer en overfitting.

Muestras artificiales: podemos intentar crear muestras sintéticas (no idénticas) utilizando diversos algoritmos que intentan seguir la tendencia del grupo minoritario (algoritmo SMOTE). Según el método, podemos mejorar los resultados. Lo peligroso de crear muestras sintéticas es que **podemos alterar la distribución** “natural” de esa clase y confundir al modelo en su clasificación

Balanced Ensemble Methods: Utiliza las ventajas de hacer ensamble de métodos, es decir, entrenar diversos modelos y entre todos obtener el resultado final (por ejemplo “votando”) pero se asegura de tomar muestras de entrenamiento equilibradas

COMBINACIONES DE MODELOS - 1

- Se pueden combinar un conjunto de distintos clasificadores en un ‘metaclasificador’ para que tenga un rendimiento de clasificación mejor que cada clasificador individualmente. Se ha demostrado que un cjto de clasificadores débiles da un rendimiento mejor que cada uno por separado
- El **voto mayoritario** es un método para obtener una clasificación a partir de las predicciones de varios modelos sobre la misma muestra. Se elige como clase predicha a la que reúne el 50% o más de los votos de los clasificadores usados. Se denomina **voto pluralista** cuando la clasificación es multiclase
- Varios enfoques de conjuntos de clasificadores que usan voto mayoritario:

  * **Bagging**

  * **Boosting**

  * **Stacking**
- El aprendizaje conjunto aumenta la **complejidad computacional** comparando con la de los clasificadores individuales. En la práctica hay que evaluar si esta complejidad adicional compensa la mejora en rendimiento del modelo.

COMBINACIONES DE MODELOS - 2

- BAGGING: A partir del cjto de entrenamiento se extraen muestras de BOOTSTRAP (aleatorias con reemplazo) que se usan para entrenar cada una a un clasificador de la combinación que se use. La clase predicha se determina por voto mayoritario sobre todas las predicciones para el mismo ejemplo del dataset.

  Cabe destacar que como los subconjuntos del dataset de entrenamiento se eligen con reemplazo, algunas muestras pueden aparecer varias veces y otras, ninguna en estos conjuntos re-muestreados.
- La técnica de bagging la propuso Leo Breigman en 1994, quien demostró que puede mejorar la precisión de modelos inestables y reducir el sobrejuste.
- El bagging puede reducir la varianza del conjunto de modelo (dispersión de clasificaciones frente a distintos conjuntos de entrenamiento) pero no el sesgo (diferencia entre el valor predicho y el verdadero). Por eso se hace bagging con modelos que tienen un sesgo bajo, como un árbol de decisión sin podar.
- Ejemplo: **bosque aleatorio (random forest)** es un caso particular donde todos los clasificadores son árboles de decisión que no se podan

  * En SKLEARN tenemos:

  class sklearn.ensemble.BaggingClassifier(estimator=None, n_estimators=10, *, max_samples=1.0, max_features=1.0, bootstrap=True, bootstrap_features=False, oob_score=False, warm_start=False, n_jobs=None, random_state=None, verbose=0, base_estimator='deprecated')

  Info en https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html

COMBINACIONES DE MODELOS - 3

- BOOSTING: Se conforma el ensamble con clasificadores débiles (simples, apenas mejores que la adivinación aleatoria, x ej el tocón de un árbol de decisión). Los subconjuntos de entrenamiento son muestreados sin reemplazo.

  La idea clave es centrarse en las muestras del cjto de entrenamiento que son difíciles de clasificar, es decir, hacer que los clasificadores simples aprendan posteriormente de muestras antes mal clasificadas.
- Pasos (esta es una versión simple, se usan en gral más de 3 clasificadores):

  * Obtener subcjto aleatorio de muestras SIN REEMPLAZO ‘d1’ (a partir del cjto de entrenamiento ‘D’) para entrenar al clasificador débil C1

  * Obtener un 2do cjto de entrenamiento aleatorio sin reemplazo ‘d2’ a partir de ‘D’ y añadir el 50% de las muestras previamente mal clasificadas por C1, para entrenar a otro clasificador débil C2

  * Encontrar las muestras de entrenamiento ‘d3’ en el cjto de entrenamiento ‘D’, para las cuales C1 y C2 NO están de acuerdo en la predicción, para entrenar a otro clasificador débil C3

  * Combinar las clasificaciones obtenidas de C1, C2 y C3 mediante el voto mayoritario
- Según Breiman, el boosting podría disminuir sesgo y varianza frente el bagging (en la práctica los algoritmos de boosting tienen alta varianza).
- Ejemplo: AdaBoost, XGBoost (éste además usa Descenso del Gradiente para minimizar los errores de clasificación)

  En SKLEARN hay varias implementaciones de algoritmos que usan boosting y también existe la librería XGBOOST

  Más info en https://machinelearningmastery.com/gradient-boosting-with-scikit-learn-xgboost-lightgbm-and-catboost/

COMBINACIONES DE MODELOS - 4

- STACKING: se puede pensar como un clasificador con 2 capas. La primera capa consta de varios clasificadores individuales que proporcionan sus predicciones al 2do nivel, en el cual otro clasificador (usualmente una regresión logística) se entrena con las predicciones de los clasificadores del nivel 1 para hacer las predicciones finales.

  * En SKLEARN existe la clase:

  class sklearn.ensemble.StackingClassifier(estimators, final_estimator=None, *, cv=None, stack_method='auto', n_jobs=None, passthrough=False, verbose=0)

  Info en https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html

  * En MLXTEND existe un StackingClassifier y StackingCVClassifier y son compatibles con Sklearn, más info en:

  @ https://rasbt.github.io/mlxtend/user_guide/classifier/StackingClassifier

  @ https://rasbt.github.io/mlxtend/user_guide/classifier/StackingCVClassifier/

Fuentes:

* *'Python Machine Learning'*, Sebastian Raschka - Vahid Mirjalili, Editorial Marcombo, 2da edición en castellano, 2019 (mayoritariamente)
* Documentación oficial de Scikit Learn
* Documentación oficial de MLXTEND
* Wikipedia