# PREPROCESSING

```
import os, argparse, platform
import pandas as pd, numpy as np
from typing import Literal
from sklearn.preprocessing import StandardScaler

# ===== Limpia la terminal dependiendo el OS
if platform.system() == 'Windows': os.system('cls')
elif platform.system() == 'Linux': os.system('clear')

# ===== Procesamiento de Argumentos
parser = argparse.ArgumentParser()
parser.add_argument('--tipoProcesamiento', default='reg_logistica', type=str, help="Indica
qué tipo de procesamiento seguir: ['reg_logistica','arboles','ensamble','red_neuronal']")
args = parser.parse_args()

# ===== Clase para procesamiento de data
class PreprocesarData:
    # =============== CONSTRUCTOR ===============
    def
__init__(self,tipo_procesamiento:Literal['reg_logistica','arboles','ensamble','red_neuronal']):

        self.location_path = os.path.dirname(os.path.dirname(__file__))
        self.tipo_procesamiento = tipo_procesamiento



    # =============== METODOS PRIVADOS ===============
    # ===== General
    def __cargar_dataframe(self):
        print('Cargando Dataframe...')
        # ===== Lectura de Data
        file_path = os.path.join(self.location_path,'Data','raw','german.data')
        with open(file_path,'r') as file:
            lines = [line.replace('\n','').split(' ') for line in file]

        # ===== Creacion de DataFrame
        self.dataframe = pd.DataFrame(
            data = lines,
            columns = [
                'status_checking_account', # 1.  cualitative
                'month_credit_duration',   # 2.  numeric
                'credit_history',          # 3.  cualitative
                'purpose',                 # 4.  cualitative
                'credit_amount',           # 5.  numeric
```

```
            'savings_type',            # 6.  cualitative
            'years_of_employment',     # 7.  cualitative
            'pct_fee_income',          # 8.  numeric
            'status_sex',              # 9.  cualitative
            'debtors',                 # 10. cualitative
            'years_of_residence',      # 11. numeric
            'property',                # 12. cualitative
            'age',                     # 13. numeric
            'other_installments',      # 14. cualitative
            'housing',                 # 15. cualitative
            'num_existing_credits',    # 16. numeric
            'job',                     # 17. cualitative
            'num_dependents',          # 18. numeric
            'telephone',               # 19. cualitative
            'foreign_worker',          # 20. cualitative
            'target'                   # 21. (1 = Good, 2 = Bad)
        ]
    )

    self.dataframe['target'] = self.dataframe['target'].str.replace('2','0')

  def __validar_tipos_de_dato(self):
    print('\tValidando tipos de dato numerico...')
    self.dataframe['month_credit_duration'] =
self.dataframe['month_credit_duration'].astype(int)
    self.dataframe['credit_amount']        = self.dataframe['credit_amount'].astype(int)
    self.dataframe['age']                  = self.dataframe['age'].astype(int)
    self.dataframe['pct_fee_income']       = self.dataframe['pct_fee_income'].astype(int)
    self.dataframe['years_of_residence']   =
self.dataframe['years_of_residence'].astype(int)
    self.dataframe['num_existing_credits'] =
self.dataframe['num_existing_credits'].astype(int)
    self.dataframe['num_dependents']       = self.dataframe['num_dependents'].astype(int)
    self.dataframe['target']               = self.dataframe['target'].astype(int)

  def __guardar_dataframe(self):
    print('Guardando dataframe...')
    self.dataframe.to_csv(

os.path.join(self.location_path,'Data','processed',f'data_preprocesada_{self.tipo_procesamie
nto}.csv'),
        index = False
    )

  def __explicar_dataframe(self):
    print('\tEl dataframe está así:')
    index = 1
    for column in self.dataframe.columns:
```

```
            text = f"{index}.- {column}:"
            if len(text) < 40: text = f"{text}{' '*(40-len(text))}"
            print(f"\t\t{text}{sorted(list(set(self.dataframe[column]))[:5])}")
            index += 1

    def __cambiar_nomenclaturas(self):
        print('\tCambiando dataframe para que sea más interpretativo...')
        self.dataframe['status_checking_account'] = self.dataframe['status_checking_account'
].str.replace('A11','low'    ).str.replace('A12','mid'          ).str.replace('A13','high'
).str.replace('A14','low')
        self.dataframe['credit_history']          = self.dataframe['credit_history'
].str.replace('A30','warning').str.replace('A31','good'         ).str.replace('A32','good'
).str.replace('A33','warning'      ).str.replace('A34','critical')
        self.dataframe['purpose']                 = self.dataframe['purpose'
].str.replace('A410','others').str.replace('A40','durable-goods').str.replace('A41','durable-goods
').str.replace('A42','durable-goods').str.replace('A43','durable-goods').str.replace('A44','durabl
e-goods').str.replace('A45','business').str.replace('A46','education').str.replace('A47','others').
str.replace('A48','education').str.replace('A49','business')
        self.dataframe['savings_type']            = self.dataframe['savings_type'
].str.replace('A61','low'    ).str.replace('A62','mid'          ).str.replace('A63','high'
).str.replace('A64','high'         ).str.replace('A65','low')
        self.dataframe['years_of_employment']     = self.dataframe['years_of_employment'
].str.replace('A71','jr'     ).str.replace('A72','jr'           ).str.replace('A73','md'
).str.replace('A74','md'           ).str.replace('A75','sr')
        self.dataframe['status_sex']              = self.dataframe['status_sex'
].str.replace('A91','male'   ).str.replace('A93','male'         ).str.replace('A94','male'
).str.replace('A92','female'       ).str.replace('A95','female')
        self.dataframe['debtors']                 = self.dataframe['debtors'
].str.replace('A101','none'  ).str.replace('A102','co-applicant').str.replace('A103','guarantor')
        self.dataframe['property']                = self.dataframe['property'
].str.replace('A121','high'  ).str.replace('A122','mid'         ).str.replace('A123','mid'
).str.replace('A124','low')
        self.dataframe['housing']                 = self.dataframe['housing'
].str.replace('A151','rent'  ).str.replace('A152','own'         ).str.replace('A153','for free')
        self.dataframe['other_installments']      = self.dataframe['other_installments'
].str.replace('A141','bank'  ).str.replace('A142','stores'      ).str.replace('A143','none')
        self.dataframe['job']                     = self.dataframe['job'
].str.replace('A171','no'    ).str.replace('A172','no'          ).str.replace('A173','yes'
).str.replace('A174','yes')
        self.dataframe['telephone']               = self.dataframe['telephone'
].str.replace('A191','no'    ).str.replace('A192','yes')
        self.dataframe['foreign_worker']          = self.dataframe['foreign_worker'
].str.replace('A201','yes'   ).str.replace('A202','no')

        if not self.tipo_procesamiento == 'reg_logistica': self.dataframe['pct_fee_income'] =
self.dataframe['pct_fee_income'].astype(str).str.replace('1','low').str.replace('2','low-mid').str.r
eplace('3','mid-high').str.replace('4','high')
```

```
    def __asegurar_encoding_numerico(self):
        """Asegura que TODAS las columnas (excepto 'target') sean numéricas.
        Si quedan strings, las transforma vía One-Hot Encoding (sin drop_first para evitar
pérdida)."""
        import pandas as pd
        import numpy as np

        print("\t🔍 Verificando y saneando tipos de datos finales...")

        # Identificar columnas no numéricas (excluyendo 'target')
        non_numeric_cols =
self.dataframe.select_dtypes(exclude=[np.number]).columns.tolist()
        if 'target' in non_numeric_cols:
            non_numeric_cols.remove('target')

        if non_numeric_cols:
            print(f"\t⚠️  Columnas no numéricas detectadas: {non_numeric_cols}")
            # Aplicar One-Hot Encoding *agresivo* (sin drop_first) a lo que quede
            print("\t🔧 Aplicando One-Hot Encoding final a columnas restantes...")
            try:
                # Guardar target
                target = self.dataframe['target'].copy()
                features = self.dataframe.drop(columns=['target'], errors='ignore')

                # One-Hot seguro (solo a object/string)
                features = pd.get_dummies(
                    features,
                    columns=non_numeric_cols,
                    dtype=int,
                    prefix=non_numeric_cols
                )

                # Volver a armar
                self.dataframe = pd.concat([features, target], axis=1)
                print(f"\t✅ Conversión completada. Nuevas dimensiones: {self.dataframe.shape}")
            except Exception as e:
                print(f"\t❌ Error en encoding final: {e}")
                raise
        else:
            print("\t✅ Todas las columnas (excepto target) ya son numéricas.")

    # ===== Logista
    def __winsorize_column(self, col):
        if col == 'month_credit_duration':
            return np.clip(self.dataframe[col], 4, 42)  # Límites del EDA
        elif col == 'credit_amount':
            return np.clip(self.dataframe[col], 250, 7882)  # Límite superior EDA
        elif col == 'age':
```

```
            return np.clip(self.dataframe[col], 19, 65)  # Considerar outliers >65

    def __crear_features_agrupadas(self,eliminar_granularidad:bool = True):
        print('\tCreando nuevas categorias agrupadas...')
        self.dataframe['fx_credit_duration'] = [('less 1 year'   if year < 12     else ('more 3 years' if
year > 36     else '1 to 3 years')) for year    in self.dataframe['month_credit_duration']]
        self.dataframe['fx_credit_amount']   = [('less 3.5k'     if amount < 3500 else ('more 7k'
if amount > 7000 else '3.5k to 7k'))   for amount  in self.dataframe['credit_amount']]
        self.dataframe['fx_age']             = [('early adult'   if age < 38      else ('late adult'   if age >
57      else 'adult'))       for age     in self.dataframe['age']]

        if eliminar_granularidad:
self.dataframe.drop(['month_credit_duration','credit_amount','age'],axis=1,inplace=True)

    def __aplicar_hot_encoding(self):
        print('\tAplicando One-Hot Encoding...')

        # 1. Guardamos la columna 'target' para que no se vea afectada por la codificación
        target = self.dataframe['target']
        features = self.dataframe.drop('target', axis=1)

        # 2. Aplicamos get_dummies.
        #    - Esto convertirá automáticamente TODAS las columnas 'object' (strings) en
columnas dummy.
        #    - Dejará las columnas que ya son numéricas (como 'years_of_residence') intactas.
        #    - drop_first=True es crucial para la regresión logística, ya que evita la
multicolinealidad.
        #    - dtype=int asegura que las nuevas columnas sean 0 y 1, no booleanos.
        features_encoded = pd.get_dummies(features, drop_first=True, dtype=int)

        # 3. Volvemos a unir las features (ya 100% numéricas) con la columna target
        self.dataframe = pd.concat([features_encoded, target], axis=1)

    def __flujo_regresion_logistica(self):
        print('\tPreprocesamiento optimizado para regresión logística...')

        # 1. Tratar outliers mediante winsorización
        for col in ['credit_amount', 'month_credit_duration', 'age']:
            self.dataframe[col] = self.__winsorize_column(col)

        # 2. Crear variables transformadas
        self.dataframe['log_credit_amount'] = np.log1p(self.dataframe['credit_amount'])
        self.dataframe['monthly_payment'] = self.dataframe['credit_amount'] /
self.dataframe['month_credit_duration']

        # 3. Recodificar variables categóricas
        self.__cambiar_nomenclaturas()
```

```
        # 4. Crear features agrupadas PERO mantener variables originales
        self.__crear_features_agrupadas(eliminar_granularidad=False)

        # 5. One-Hot Encoding
        self.__aplicar_hot_encoding()

        # 6. Estandarizar variables numéricas (aplicar después de One-Hot Encoding)
        numeric_cols = ['month_credit_duration', 'credit_amount', 'age', 'years_of_residence',
                        'num_existing_credits', 'num_dependents', 'pct_fee_income',
                        'log_credit_amount', 'monthly_payment']

        scaler = StandardScaler()
        self.dataframe[numeric_cols] = scaler.fit_transform(self.dataframe[numeric_cols])

    # ===== arboles
    def __winsorize_extreme_outliers(self,col):
        Q1 = self.dataframe[col].quantile(0.25)
        Q3 = self.dataframe[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 3 * IQR
        upper_bound = Q3 + 3 * IQR
        return np.clip(self.dataframe[col], lower_bound, upper_bound)

    def __target_encode_purpose(self):
        global_mean = self.dataframe['target'].mean()
        purpose_counts = self.dataframe['purpose'].value_counts()
        purpose_target_means = self.dataframe.groupby('purpose')['target'].mean()

        # Regularización basada en el tamaño de cada categoría
        min_samples_leaf = 50
        smoothing = 1 / (1 + np.exp(-(purpose_counts - min_samples_leaf) / 10))
        self.dataframe['purpose_encoded'] = self.dataframe['purpose'].map(
            lambda x: smoothing[x] * purpose_target_means.get(x, global_mean) +
                  (1 - smoothing[x]) * global_mean if x in smoothing.index else global_mean
        )

    def __flujo_arboles(self):
        print('\tPreprocesamiento optimizado para árboles de decisión...')

        # 1. Tratar outliers extremos (no eliminar, sino suavizar)
        for col in ['credit_amount', 'month_credit_duration']:
            self.dataframe[col] = self.__winsorize_extreme_outliers(col)

        # 2. Transformaciones clave basadas en EDA
        self.dataframe['credit_to_duration_ratio'] = self.dataframe['credit_amount'] /
self.dataframe['month_credit_duration']
        self.dataframe['age_squared'] = self.dataframe['age'] ** 2
```

```
        # 3. Codificación ordinal mejorada
        ordinal_mappings = {
            'savings_type': {'A65': 0, 'A61': 1, 'A62': 2, 'A63': 3, 'A64': 4},
            'years_of_employment': {'A71': 0, 'A72': 1, 'A73': 2, 'A74': 3, 'A75': 4},
            'status_checking_account': {'A14': 0, 'A11': 1, 'A12': 2, 'A13': 3},
            'credit_history': {'A30': 0, 'A31': 1, 'A32': 2, 'A33': 3, 'A34': 4},
            'pct_fee_income': {'1': 0, '2': 1, '3': 2, '4': 3},  # Asegurar strings
            'job': {'A171': 0, 'A172': 1, 'A173': 2, 'A174': 3}
        }

        # 4. Creación de características basadas en hallazgos del EDA
        # Variables numéricas con alta diferencia entre clases según EDA
        self.dataframe['high_duration'] = (self.dataframe['month_credit_duration'] >
24).astype(int)
        self.dataframe['high_amount'] = (self.dataframe['credit_amount'] > 3972).astype(int)

        # Interacciones críticas identificadas en el EDA
        self.dataframe['high_risk_profile'] = ((self.dataframe['credit_history'].isin([0, 4])) &  #
A30/A34
                                               (self.dataframe['status_checking_account'] == 1) &  # A11
                                               (self.dataframe['credit_amount'] > 3972)).astype(int)

        # 5. One-Hot Encoding para variables nominales
        low_cardinality_vars = ['status_sex', 'debtors', 'property', 'housing',
                                'other_installments', 'telephone', 'foreign_worker']
        self.dataframe = pd.get_dummies(self.dataframe, columns=low_cardinality_vars,
                                        prefix=low_cardinality_vars, dtype=int)

        # 6. Target Encoding regularizado para purpose
        self.__target_encode_purpose()

        # 7. Eliminar variables de baja relevancia según EDA
        low_relevance_vars = ['years_of_residence', 'num_dependents']
        for var in low_relevance_vars:
            if var in self.dataframe.columns:
                self.dataframe.drop(var, axis=1, inplace=True)

        # 8. Transformaciones adicionales para variables sesgadas
        self.dataframe['log_credit_amount'] = np.log1p(self.dataframe['credit_amount'])

    # ===== ensamble
    def __target_encoding_regularizado(self):
        global_mean = self.dataframe['target'].mean()
        min_samples_leaf = 50  # Mínimo de muestras para confiar en la media local

        # Para variables categóricas clave identificadas en el EDA
        for cat_col in ['purpose', 'status_checking_account', 'credit_history']:
            if cat_col in self.dataframe.columns:
```

```
            cat_counts = self.dataframe[cat_col].value_counts()
            cat_means = self.dataframe.groupby(cat_col)['target'].mean()

            # Factor de suavizado basado en el tamaño de la categoría
            smoothing = 1 / (1 + np.exp(-(cat_counts - min_samples_leaf) / 10))

            # Crear columna encoded
            encoded_col = f'{cat_col}_encoded'
            self.dataframe[encoded_col] = self.dataframe[cat_col].map(
                lambda x: smoothing.get(x, 0) * cat_means.get(x, global_mean) +
                      (1 - smoothing.get(x, 0)) * global_mean if x in cat_counts.index else
global_mean
            )

            # Eliminar variable original después de encoding
            self.dataframe.drop(cat_col, axis=1, inplace=True)

    def __crear_interacciones_eda(self):
        """Crea interacciones usando los valores recodificados (strings)"""
        print('\tCreando interacciones específicas basadas en hallazgos del EDA...')

        # Combinación identificada como de máximo riesgo en el EDA
        self.dataframe['high_risk_combo'] = (
            (self.dataframe['credit_history'] == 'critical') &      # A30 - 62.5% bad loans
            (self.dataframe['status_checking_account'] == 'low') &  # A11 - 49.27% bad loans
            (self.dataframe['credit_amount'] > 3972) &              # Monto alto (percentil 75)
            (self.dataframe['month_credit_duration'] > 24)          # Duración larga (percentil 75)
        ).astype(int)

        # Para el hallazgo sobre trabajadores extranjeros
        self.dataframe['domestic_worker_high_risk'] = (
            (self.dataframe['foreign_worker'] == 'no') &      # Trabajador local
            (self.dataframe['credit_history'] == 'critical')  # Historial crítico
        ).astype(int)

        # Interacciones de las variables más discriminativas según EDA
        self.dataframe['long_credit_high_amount'] = (
            (self.dataframe['month_credit_duration'] > 24) &  # Percentil 75
            (self.dataframe['credit_amount'] > 3972)          # Percentil 75
        ).astype(int)

        self.dataframe['critical_history_low_balance'] = (
            (self.dataframe['credit_history'].isin(['critical', 'warning'])) &  # A30/A34
            (self.dataframe['status_checking_account'] == 'low')                # A11
        ).astype(int)

    def __crear_variables_riesgo_acumulativo(self):
        """Crea variables de riesgo usando los valores recodificados"""
```

```
        print('\tCreando variables de riesgo acumulativo...')

        # Variable acumulativa de factores de riesgo identificados en el EDA
        risk_factors = [
            # Factores con >25% diferencia entre clases según EDA
            (self.dataframe['credit_amount'] > 3972).astype(int),
            (self.dataframe['month_credit_duration'] > 24).astype(int),
            (self.dataframe['credit_history'].isin(['critical', 'warning'])).astype(int),
            (self.dataframe['status_checking_account'] == 'low').astype(int),
            # Factores con 15-25% diferencia
            (self.dataframe['savings_type'] == 'low').astype(int),
            (self.dataframe['years_of_employment'] == 'jr').astype(int),
            (self.dataframe['purpose'].isin(['durable-goods'])).astype(int),
            (self.dataframe['debtors'] == 'co-applicant').astype(int),
            (self.dataframe['housing'] == 'rent').astype(int),
            (self.dataframe['other_installments'].isin(['bank', 'stores'])).astype(int)
        ]

        self.dataframe['risk_factor_count'] = sum(risk_factors)
        self.dataframe['risk_factor_ratio'] = self.dataframe['risk_factor_count'] / len(risk_factors)

        # Score de estabilidad financiera (inverso al riesgo)
        stability_factors = [
            (self.dataframe['credit_history'] == 'good').astype(int),
            (self.dataframe['status_checking_account'] == 'high').astype(int),
            (self.dataframe['savings_type'] == 'high').astype(int),
            (self.dataframe['years_of_employment'] == 'sr').astype(int),
            (self.dataframe['property'].isin(['high', 'mid'])).astype(int),
            (self.dataframe['housing'] == 'own').astype(int),
            (self.dataframe['foreign_worker'] == 'yes').astype(int),
            (self.dataframe['age'] >= 38).astype(int)
        ]

        self.dataframe['stability_factor_count'] = sum(stability_factors)
        self.dataframe['stability_factor_ratio'] = self.dataframe['stability_factor_count'] /
len(stability_factors)

    def __eliminar_variables_baja_relevancia(self):
        print('\tEliminando variables de baja relevancia según EDA bivariado...')

        # Variables con mínima diferencia entre clases según EDA bivariado (<1%)
        low_relevance_vars = [
            'years_of_residence',    # 0.25% diferencia
            'num_dependents'         # 0.21% diferencia
        ]

        # Variables con diferencia moderada pero que pueden ser redundantes
        moderate_relevance_vars = [
```

```
            'num_existing_credits',  # 4.05% diferencia
            'age'                    # 6.24% diferencia
        ]

        # Eliminar variables de muy baja relevancia
        for var in low_relevance_vars:
            if var in self.dataframe.columns:
                print(f"\t\tEliminando variable de muy baja relevancia: {var}")
                self.dataframe.drop(var, axis=1, inplace=True)

        # Considerar eliminar variables de relevancia moderada si ya tenemos
transformaciones
        for var in moderate_relevance_vars:
            if var in self.dataframe.columns:
                print(f"\t\tConsiderando eliminación de variable de relevancia moderada: {var}")
                # Mantener si no tenemos transformaciones alternativas
                if f'log_{var}' not in self.dataframe.columns and f'{var}_binned' not in
self.dataframe.columns:
                    print(f"\t\tManteniendo {var} por no tener transformaciones alternativas")
                else:
                    print(f"\t\tEliminando {var} por tener transformaciones alternativas")
                    self.dataframe.drop(var, axis=1, inplace=True)

    def __aplicar_one_hot_encoding_selectivo(self):
        print('\tAplicando One-Hot Encoding selectivo a variables nominales restantes...')

        # Identificar variables categóricas restantes (object o category dtype)
        categorical_cols = self.dataframe.select_dtypes(include=['object',
'category']).columns.tolist()

        # Excluir variables que ya fueron codificadas o que serán eliminadas
        cols_to_exclude = ['purpose']  # Ya fue target encoded

        for col in cols_to_exclude:
            if col in categorical_cols:
                categorical_cols.remove(col)

        # Aplicar One-Hot Encoding selectivo
        if categorical_cols:
            print(f"\t\tAplicando One-Hot Encoding a: {categorical_cols}")
            self.dataframe = pd.get_dummies(
                self.dataframe,
                columns=categorical_cols,
                prefix=categorical_cols,
                dtype=int,
                drop_first=True  # Evitar multicolinealidad
            )
        else:
```

```
            print("\t\tNo hay variables categóricas restantes para codificar")

    def __feature_selection(self):
        print('\tRealizando feature selection final basado en varianza y correlación...')

        # 1. Eliminar variables con varianza muy baja (<1%)
        numeric_cols = self.dataframe.select_dtypes(include=[np.number]).columns.tolist()
        if 'target' in numeric_cols:
            numeric_cols.remove('target')

        low_variance_cols = []
        for col in numeric_cols:
            variance = self.dataframe[col].var()
            if variance < 0.01:  # Umbral de varianza
                low_variance_cols.append(col)
                print(f"\t\tVariable de baja varianza detectada: {col} (var={variance:.4f})")

        # Eliminar las columnas de baja varianza
        for col in low_variance_cols:
            if col in self.dataframe.columns:
                self.dataframe.drop(col, axis=1, inplace=True)

        # ACTUALIZAR: Recalcular numeric_cols después de eliminar las de baja varianza
        numeric_cols = self.dataframe.select_dtypes(include=[np.number]).columns.tolist()
        if 'target' in numeric_cols:
            numeric_cols.remove('target')

        # 2. Eliminar variables altamente correlacionadas (>0.9)
        if len(numeric_cols) > 0:  # Solo proceder si hay columnas numéricas restantes
            corr_matrix = self.dataframe[numeric_cols].corr().abs()
            upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))

            high_corr_cols = []
            for col in upper.columns:
                high_corr = upper.index[upper[col] > 0.9].tolist()
                for high_corr_col in high_corr:
                    if f"{col}_{high_corr_col}" not in [f"{a}_{b}" for a, b in high_corr_cols]:
                        high_corr_cols.append((col, high_corr_col))
                        print(f"\t\tVariables altamente correlacionadas: {col} y {high_corr_col}
(corr={upper[col][high_corr_col]:.4f})")

            # Mantener la variable con mayor correlación con el target
            cols_to_drop = []
            for col1, col2 in high_corr_cols:
                if col1 in self.dataframe.columns and col2 in self.dataframe.columns:
                    corr1 = abs(self.dataframe[col1].corr(self.dataframe['target']))
                    corr2 = abs(self.dataframe[col2].corr(self.dataframe['target']))
```

```
                if corr1 < corr2:
                    cols_to_drop.append(col1)
                else:
                    cols_to_drop.append(col2)

        for col in set(cols_to_drop):
            if col in self.dataframe.columns:
                print(f"\t\tEliminando variable redundante: {col}")
                self.dataframe.drop(col, axis=1, inplace=True)
    else:
        print("\t\tNo hay variables numéricas restantes para analizar correlación")

def __flujo_ensamble(self):
    print('\tPreprocesamiento optimizado para modelos ensemble...')

    # 1. Recodificar variables categóricas primero
    self.__cambiar_nomenclaturas()

    # 2. Crear interacciones específicas basadas en hallazgos del EDA
    self.__crear_interacciones_eda()

    # 3. Crear variables de riesgo acumulativo
    self.__crear_variables_riesgo_acumulativo()

    # 4. Tratar outliers extremos (no eliminar, sino suavizar)
    for col in ['credit_amount', 'month_credit_duration', 'num_dependents']:
        if col in self.dataframe.columns:
            self.dataframe[col] = self.__winsorize_extreme_outliers(col)

    # 5. Mantener transformaciones logarítmicas valiosas
    self.dataframe['log_credit_amount'] = np.log1p(self.dataframe['credit_amount'])
    self.dataframe['log_credit_duration'] = np.log1p(self.dataframe['month_credit_duration'])

    # 6. Target encoding regularizado para variables categóricas clave
    self.__target_encoding_regularizado()

    # 7. Eliminar variables de baja relevancia según EDA bivariado
    self.__eliminar_variables_baja_relevancia()

    # 8. One-Hot Encoding para variables nominales restantes
    self.__aplicar_one_hot_encoding_selectivo()

    # 9. Feature selection final basado en varianza y correlación
    self.__feature_selection()

    # 10. Crear variables adicionales para ensemble
    print('\tCreando variables adicionales para ensemble...')
```

```
        # Interacciones específicas para ensembles
        if 'status_checking_account_encoded' in self.dataframe.columns and
'credit_history_encoded' in self.dataframe.columns:
            self.dataframe['checking_credit_interaction'] = (
                self.dataframe['status_checking_account_encoded'] *
                self.dataframe['credit_history_encoded']
            )

        if 'years_of_employment_encoded' in self.dataframe.columns and
'savings_type_encoded' in self.dataframe.columns:
            self.dataframe['employment_savings_interaction'] = (
                self.dataframe['years_of_employment_encoded'] *
                self.dataframe['savings_type_encoded']
            )

        if 'age' in self.dataframe.columns and 'credit_to_duration_ratio' in
self.dataframe.columns:
            self.dataframe['age_credit_ratio'] = self.dataframe['age'] /
(self.dataframe['credit_to_duration_ratio'] + 1)

        # Binning estratégico para variables continuas
        if 'age' in self.dataframe.columns:
            self.dataframe['age_binned'] = pd.cut(
                self.dataframe['age'],
                bins=[0, 25, 35, 45, 55, 100],
                labels=[0, 1, 2, 3, 4],
                include_lowest=True
            ).astype(int)

        if 'credit_amount' in self.dataframe.columns:
            self.dataframe['credit_amount_binned'] = pd.cut(
                self.dataframe['credit_amount'],
                bins=[0, 2500, 5000, 7500, 10000, 20000],
                labels=[0, 1, 2, 3, 4],
                include_lowest=True
            ).astype(int)

        # CORRECCIÓN: Usar las columnas dummy de pct_fee_income en lugar de la
columna original
        # Ratios y proporciones - debt_burden
        pct_fee_dummy_cols = [col for col in self.dataframe.columns if
col.startswith('pct_fee_income_')]
        if pct_fee_dummy_cols and 'credit_amount' in self.dataframe.columns:
            # Calcular debt_burden usando la categoría con mayor riesgo
(pct_fee_income_high)
            if 'pct_fee_income_high' in self.dataframe.columns:
                self.dataframe['debt_burden'] = self.dataframe['credit_amount'] / (
                    self.dataframe['pct_fee_income_high'] + 1
```

```
                )
            else:
                # Si no existe 'high', usar la primera disponible
                self.dataframe['debt_burden'] = self.dataframe['credit_amount'] / (
                    self.dataframe[pct_fee_dummy_cols[0]] + 1
                )

        # CORRECCIÓN: stability_score - verificar que las columnas existan
        if ('years_of_residence' in self.dataframe.columns and
            'years_of_employment_encoded' in self.dataframe.columns and
            'age' in self.dataframe.columns):
            self.dataframe['stability_score'] = (
                self.dataframe['years_of_residence'] +
                self.dataframe['years_of_employment_encoded']
            ) / (self.dataframe['age'] + 1)
        else:
            print("\t\tNo se pudo crear 'stability_score' - faltan columnas requeridas")

    # ===== Red Neuronal
    def __winsorize_outliers_neural(self, col):
        """Suaviza outliers manteniendo información extrema pero reduciendo su impacto"""
        Q1 = self.dataframe[col].quantile(0.25)
        Q3 = self.dataframe[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 2.5 * IQR  # Límite más conservador que para árboles
        upper_bound = Q3 + 2.5 * IQR
        return np.clip(self.dataframe[col], lower_bound, upper_bound)

    def __create_polynomial_features(self):
        """Crea características polinómicas para capturar relaciones no lineales"""
        print('\tCreando características polinómicas para relaciones no lineales...')

        # Transformación logarítmica para variables sesgadas
        self.dataframe['log_credit_amount'] = np.log1p(self.dataframe['credit_amount'])
        self.dataframe['log_month_credit_duration'] =
np.log1p(self.dataframe['month_credit_duration'])

        # Características polinómicas para variables clave según EDA
        self.dataframe['credit_amount_sq'] = self.dataframe['credit_amount'] ** 2
        self.dataframe['credit_duration_sq'] = self.dataframe['month_credit_duration'] ** 2
        self.dataframe['age_sq'] = self.dataframe['age'] ** 2

        # Interacciones de segundo orden para variables con alta correlación con target
        self.dataframe['credit_duration_amount_ratio'] = (
            self.dataframe['credit_amount'] / (self.dataframe['month_credit_duration'] + 1)
        )
        self.dataframe['age_credit_ratio'] = self.dataframe['age'] /
(self.dataframe['credit_amount'] + 1)
```

```
    # Características cíclicas para edad (capturar efectos no lineales)
    self.dataframe['age_sin'] = np.sin(2 * np.pi * self.dataframe['age'] / 100)
    self.dataframe['age_cos'] = np.cos(2 * np.pi * self.dataframe['age'] / 100)

def __categorize_features_for_embedding(self):
    """Prepara variables categóricas para embeddings en la red neuronal"""
    print('\tPreparando variables categóricas para embeddings...')

    # Mapeo de variables categóricas a enteros para embeddings
    categorical_vars = [
        'status_checking_account', 'credit_history', 'purpose', 'savings_type',
        'years_of_employment', 'status_sex', 'debtors', 'property',
        'other_installments', 'housing', 'job', 'telephone', 'foreign_worker'
    ]

    for var in categorical_vars:
        if var in self.dataframe.columns:
            # Crear mapeo de categorías a enteros
            categories = sorted(self.dataframe[var].unique())
            cat_to_idx = {cat: idx for idx, cat in enumerate(categories)}
            self.dataframe[f'{var}_idx'] = self.dataframe[var].map(cat_to_idx)

    # Eliminar variables originales después de crear los índices
    self.dataframe.drop(categorical_vars, axis=1, inplace=True)

def __normalize_features(self):
    """Normaliza todas las características numéricas al rango [0,1]"""
    print('\tNormalizando características para la red neuronal...')

    # Identificar todas las columnas numéricas excepto target
    numeric_cols = self.dataframe.select_dtypes(include=[np.number]).columns.tolist()
    numeric_cols.remove('target')

    # Normalización Min-Max para todas las características numéricas
    for col in numeric_cols:
        min_val = self.dataframe[col].min()
        max_val = self.dataframe[col].max()
        if max_val != min_val:  # Evitar división por cero
            self.dataframe[col] = (self.dataframe[col] - min_val) / (max_val - min_val)
        else:
            # Si todos los valores son iguales, establecer a 0.5
            self.dataframe[col] = 0.5

    # Asegurar que no haya NaN después de la normalización
    self.dataframe[numeric_cols] = self.dataframe[numeric_cols].fillna(0.5)

def __create_risk_profiles(self):
```

```
        """Crea variables sintéticas que representen perfiles de riesgo identificados en el
EDA"""
        print('\tCreando perfiles de riesgo sintéticos...')

        # Perfil de alto riesgo según EDA (más de 50% de préstamos malos)
        self.dataframe['high_risk_profile'] = (
            (self.dataframe['credit_history'] == 'critical') |  # A30 - 62.5% bad loans
            ((self.dataframe['status_checking_account'] == 'low') &  # A11 - 49.27% bad loans
            (self.dataframe['credit_amount'] > 3972)) |
            ((self.dataframe['purpose'] == 'durable-goods') &
            (self.dataframe['month_credit_duration'] > 24))
        ).astype(int)

        # Perfil de bajo riesgo según EDA (menos de 20% de préstamos malos)
        self.dataframe['low_risk_profile'] = (
            (self.dataframe['credit_history'] == 'good') &  # A31/A32 - ~30% bad loans (mejor que
promedio)
            (self.dataframe['status_checking_account'] == 'high') &  # A13 - 22.22% bad loans
            (self.dataframe['savings_type'] == 'high') &
            (self.dataframe['years_of_employment'] == 'sr')
        ).astype(int)

        # Tendencia de riesgo acumulativo
        risk_factors = [
            (self.dataframe['credit_history'] == 'critical').astype(int),
            (self.dataframe['credit_history'] == 'warning').astype(int),
            (self.dataframe['status_checking_account'] == 'low').astype(int),
            (self.dataframe['savings_type'] == 'low').astype(int),
            (self.dataframe['years_of_employment'] == 'jr').astype(int),
            (self.dataframe['credit_amount'] > 3972).astype(int),
            (self.dataframe['month_credit_duration'] > 24).astype(int),
            (self.dataframe['debtors'] == 'co-applicant').astype(int),
            (self.dataframe['housing'] == 'rent').astype(int)
        ]

        self.dataframe['cumulative_risk_score'] = sum(risk_factors) / len(risk_factors)

    def __flujo_neuronal(self):
        print('\tPreprocesamiento optimizado para red neuronal...')

        # 1. Recodificar variables categóricas primero para mantener consistencia
        self.__cambiar_nomenclaturas()

        # 2. Suavizar outliers extremos (menos agresivo que para otros modelos)
        for col in ['credit_amount', 'month_credit_duration', 'age', 'num_dependents']:
            if col in self.dataframe.columns:
                self.dataframe[col] = self.__winsorize_outliers_neural(col)
```

```
        # 3. Crear características no lineales y polinómicas
        self.__create_polynomial_features()

        # 4. Crear perfiles de riesgo sintéticos basados en hallazgos del EDA
        self.__create_risk_profiles()

        # 5. Preparar variables categóricas para embeddings
        self.__categorize_features_for_embedding()

        # 6. Eliminar variables de muy baja relevancia según EDA bivariado
        low_relevance_vars = ['years_of_residence', 'num_dependents']
        for var in low_relevance_vars:
            if var in self.dataframe.columns:
                print(f"\t\tEliminando variable de muy baja relevancia: {var}")
                self.dataframe.drop(var, axis=1, inplace=True)

        # 7. Normalizar todas las características numéricas al rango [0,1]
        self.__normalize_features()

        # 8. Crear características adicionales específicas para redes neuronales

        # Características de interacción para capturar dependencias complejas
        if 'credit_amount' in self.dataframe.columns and 'month_credit_duration' in
self.dataframe.columns:
            self.dataframe['credit_duration_interaction'] = (
                self.dataframe['credit_amount'] * self.dataframe['month_credit_duration']
            )

        if 'age' in self.dataframe.columns and 'credit_history_idx' in self.dataframe.columns:
            self.dataframe['age_history_interaction'] = (
                self.dataframe['age'] * self.dataframe['credit_history_idx']
            )

        # Característica de estabilidad financiera
        financial_stability_factors = [
            'status_checking_account_idx', 'savings_type_idx',
            'years_of_employment_idx', 'property_idx'
        ]
        available_factors = [f for f in financial_stability_factors if f in self.dataframe.columns]

        if available_factors:
            self.dataframe['financial_stability_index'] =
self.dataframe[available_factors].mean(axis=1)

        # Características de agrupamiento para edad y monto
        # (esto ayuda a la red a aprender patrones por segmentos)
        self.dataframe['age_group'] = pd.cut(
            self.dataframe['age'],
```

```
            bins=[0, 25, 35, 45, 55, 100],
            labels=[0.2, 0.4, 0.6, 0.8, 1.0],
            include_lowest=True
        ).astype(float)

        self.dataframe['amount_group'] = pd.cut(
            self.dataframe['credit_amount'],
            bins=[0, 1365, 2319, 3972, 10000, 20000],
            labels=[0.2, 0.4, 0.6, 0.8, 1.0],
            include_lowest=True
        ).astype(float)

        # 9. Normalizar nuevamente todas las características después de crear las nuevas
        self.__normalize_features()

        print('\tPreprocesamiento para red neuronal completado. Todas las características
están en el rango [0,1].')


    # =============== METODOS PUBLICOS ===============
    def process(self):
        if not self.tipo_procesamiento in ['reg_logistica','arboles','ensamble','red_neuronal']:
raise Exception(f"La opción '{self.tipo_procesamiento}' no esta permitida. Solo se permite
['reg_logistica','arboles','ensamble','red_neuronal'].")
        print(f"{'='*15} Iniciando procesamiento {self.tipo_procesamiento} {'='*15}")


        self.__cargar_dataframe()
        self.__validar_tipos_de_dato()

        if self.tipo_procesamiento   == 'reg_logistica': self.__flujo_regresion_logistica()
        elif self.tipo_procesamiento == 'arboles':       self.__flujo_arboles()
        elif self.tipo_procesamiento == 'ensamble':      self.__flujo_ensamble()
        elif self.tipo_procesamiento == 'red_neuronal':  self.__flujo_neuronal()

        self.__asegurar_encoding_numerico()
        self.__explicar_dataframe()
        self.__guardar_dataframe()




preprocesador = PreprocesarData(
    tipo_procesamiento = args.tipoProcesamiento
)
preprocesador.process()
```

# LAZY MODELING

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score,
precision_recall_curve
from lazypredict.Supervised import LazyClassifier
import warnings
warnings.filterwarnings('ignore')

def load_and_evaluate_preprocessing(preprocessing_type):
    """Carga y evalúa un tipo específico de preprocesamiento"""
    print(f"\n{'='*60}")
    print(f"EVALUANDO PREPROCESAMIENTO: {preprocessing_type.upper()}")
    print(f"{'='*60}")

    try:
        # Cargar datos preprocesados
        file_path = f"Data/processed/data_preprocesada_{preprocessing_type}.csv"
        df = pd.read_csv(file_path)
        print(f"✓ Dataset cargado: {df.shape}")

        # Separar características y target
        X = df.drop('target', axis=1)
        y = df['target']

        print(f"✓ Características: {X.shape[1]}, Target: {y.shape[0]}")
        print(f"✓ Distribución del target: {y.value_counts().to_dict()}")

        # División estratificada
        X_train, X_test, y_train, y_test = train_test_split(
            X, y,
            test_size=0.2,
            stratify=y,
            random_state=42
        )

        print(f"✓ Conjunto de entrenamiento: {X_train.shape}")
        print(f"✓ Conjunto de prueba: {X_test.shape}")

        return X_train, X_test, y_train, y_test, preprocessing_type

    except Exception as e:
        print(f"✗ Error cargando {preprocessing_type}: {e}")
        return None, None, None, None, preprocessing_type
```

```
def get_best_model_with_proba(lazy_clf, models_df):
    """Selecciona el mejor modelo que tenga predict_proba"""
    proba_models = []

    for model_name, model in lazy_clf.models.items():
        try:
            # Verificar si el modelo tiene predict_proba
            if hasattr(model, 'predict_proba'):
                # Probar llamar a predict_proba en una pequeña muestra
                if hasattr(model, 'steps'):  # Si es un pipeline
                    proba_models.append(model_name)
                else:
                    # Para modelos individuales
                    proba_models.append(model_name)
        except:
            continue

    print(f"✓ Modelos con predict_proba disponibles: {len(proba_models)}")

    # Filtrar el dataframe para solo modelos con predict_proba
    proba_models_df = models_df[models_df.index.isin(proba_models)]

    if len(proba_models_df) == 0:
        return None, None

    # Seleccionar el mejor por Balanced Accuracy
    best_model_name = proba_models_df.nlargest(1, 'Balanced Accuracy').index[0]
    best_model = lazy_clf.models[best_model_name]

    return best_model, best_model_name

def evaluate_focused_metrics(model, X_test, y_test, preprocessing_type):
    """Evalúa métricas enfocadas en detectar malos pagadores"""
    print(f"\n📊 EVALUACIÓN ESPECÍFICA PARA {preprocessing_type.upper()}")

    try:
        # Predicciones
        y_pred = model.predict(X_test)
        y_pred_proba = model.predict_proba(X_test)

        # Para modelos de sklearn, las clases están ordenadas, clase 0 es la primera
        # Verificar el orden de las clases
        if hasattr(model, 'classes_'):
            class_order = model.classes_
            if class_order[0] == 0:  # Malos pagadores son clase 0
                y_pred_proba_malos = y_pred_proba[:, 0]
            else:
                y_pred_proba_malos = y_pred_proba[:, 1]
```

```
        else:
            # Asumir que la primera columna es clase 0 (malos pagadores)
            y_pred_proba_malos = y_pred_proba[:, 0]

        # Métricas básicas
        report = classification_report(y_test, y_pred, output_dict=True)
        cm = confusion_matrix(y_test, y_pred)

        # Asegurar que la matriz de confusión se interpreta correctamente
        # Clase 0: malos pagadores, Clase 1: buenos pagadores
        tn, fp, fn, tp = cm.ravel()

        # MÉTRICAS ESPECÍFICAS PARA MALOS PAGADORES (Clase 0)
        # Sensitivity/Recall para clase 0: TN / (TN + FP)
        sensitivity_0 = tn / (tn + fp) if (tn + fp) > 0 else 0

        # Precision para clase 0: TN / (TN + FN)
        precision_0 = tn / (tn + fn) if (tn + fn) > 0 else 0

        # F1-score para clase 0
        f1_0 = 2 * (precision_0 * sensitivity_0) / (precision_0 + sensitivity_0) if (precision_0 +
sensitivity_0) > 0 else 0

        # AUC-ROC
        auc_roc = roc_auc_score(y_test, y_pred_proba_malos)

        # AUC-PR (Precision-Recall) - mejor para clases desbalanceadas
        precision, recall, _ = precision_recall_curve(y_test, y_pred_proba_malos, pos_label=0)
        auc_pr = -np.trapz(precision, recall) if len(precision) > 1 else 0

        print(f"🎯 MÉTRICAS PARA MALOS PAGADORES (Clase 0):")
        print(f"   • Sensitivity/Recall: {sensitivity_0:.4f} (capacidad de detectar malos
pagadores)")
        print(f"   • Precision: {precision_0:.4f} (cuán precisos son los que marcamos como
malos)")
        print(f"   • F1-Score: {f1_0:.4f}")
        print(f"   • AUC-ROC: {auc_roc:.4f}")
        print(f"   • AUC-PR: {auc_pr:.4f} (mejor métrica para clases desbalanceadas)")
        print(f"   • Matriz de confusión:")
        print(f"                 Predicción")
        print(f"              Malo   Bueno")
        print(f"Real Malo    [{tn:3d}    {fp:3d}]")
        print(f"Real Bueno   [{fn:3d}    {tp:3d}]")

        return {
            'preprocessing': preprocessing_type,
            'sensitivity_0': sensitivity_0,
            'precision_0': precision_0,
```

```
            'f1_0': f1_0,
            'auc_roc': auc_roc,
            'auc_pr': auc_pr,
            'tn': tn, 'fp': fp, 'fn': fn, 'tp': tp
        }

    except Exception as e:
        print(f"✗ Error en evaluación específica: {e}")
        import traceback
        traceback.print_exc()
        return None

def lazy_modeling_analysis():
    """Análisis completo con LazyPredict para los 4 tipos de preprocesamiento"""

    preprocessing_types = ['reg_logistica', 'arboles', 'ensamble', 'red_neuronal']
    all_results = []

    for preproc_type in preprocessing_types:
        # Cargar datos
        X_train, X_test, y_train, y_test, preproc_name =
load_and_evaluate_preprocessing(preproc_type)

        if X_train is None:
            continue

        print(f"\n🚀 ENTRENANDO CON LAZYPREDICT PARA {preproc_name.upper()}...")

        try:
            # Configurar LazyClassifier
            lazy_clf = LazyClassifier(
                verbose=0,
                ignore_warnings=True,
                custom_metric=None,
                random_state=42,
                classifiers='all'
            )

            # Entrenar y evaluar modelos
            models, predictions = lazy_clf.fit(X_train, X_test, y_train, y_test)

            print(f"✓ Modelos evaluados: {len(models)}")

            # Filtrar top 10 modelos por Balanced Accuracy
            top_models = models.nlargest(10, 'Balanced Accuracy')
            print(f"\n🏆 TOP 10 MODELOS PARA {preproc_name.upper()}:")
            print(top_models[['Balanced Accuracy', 'ROC AUC', 'F1 Score', 'Time Taken']])
```

```
            # Obtener el mejor modelo CON predict_proba
            best_model, best_model_name = get_best_model_with_proba(lazy_clf, models)

            if best_model is None:
                print(f"⚠ No se encontraron modelos con predict_proba para {preproc_name}")
                continue

            print(f"\n⭐ MEJOR MODELO CON predict_proba: {best_model_name}")

            # Evaluación específica para detección de malos pagadores
            focused_metrics = evaluate_focused_metrics(best_model, X_test, y_test,
preproc_name)

            if focused_metrics:
                focused_metrics['best_model'] = best_model_name
                focused_metrics['balanced_accuracy'] = models.loc[best_model_name, 'Balanced
Accuracy']
                all_results.append(focused_metrics)

        except Exception as e:
            print(f"✗ Error en LazyPredict para {preproc_type}: {e}")
            import traceback
            traceback.print_exc()
            continue

    return all_results

def final_recommendation(results):
    """Genera recomendación final basada en todos los resultados"""
    print(f"\n{'='*80}")
    print("🎯 RECOMENDACIÓN FINAL")
    print(f"{'='*80}")

    if not results:
        print("No se pudieron obtener resultados válidos de ningún preprocesamiento.")
        return None

    # Convertir a DataFrame para análisis
    results_df = pd.DataFrame(results)

    # Ponderar métricas (énfasis en sensitivity y AUC-PR para detectar malos pagadores)
    results_df['weighted_score'] = (
        results_df['sensitivity_0'] * 0.35 +  # Mayor peso a sensitivity (detectar malos)
        results_df['auc_pr'] * 0.30 +         # AUC-PR muy importante para clases
desbalanceadas
        results_df['auc_roc'] * 0.20 +
        results_df['f1_0'] * 0.15
    )
```

```
    # Mejor combinación
    best_idx = results_df['weighted_score'].idxmax()
    best_combo = results_df.loc[best_idx]

    print("📈 RESULTADOS COMPARATIVOS:")
    print("-" * 80)
    for idx, row in results_df.iterrows():
        print(f"{row['preprocessing'].upper():<15} | "
              f"Sens: {row['sensitivity_0']:.3f} | "
              f"AUC-PR: {row['auc_pr']:.3f} | "
              f"AUC-ROC: {row['auc_roc']:.3f} | "
              f"F1: {row['f1_0']:.3f} | "
              f"Modelo: {row['best_model']}")

    print(f"\n🏅 MEJOR COMBINACIÓN:")
    print(f"   • Preprocesamiento: {best_combo['preprocessing']}")
    print(f"   • Modelo: {best_combo['best_model']}")
    print(f"   • Sensitivity (detección malos pagadores): {best_combo['sensitivity_0']:.3f}")
    print(f"   • AUC-PR: {best_combo['auc_pr']:.3f}")
    print(f"   • AUC-ROC: {best_combo['auc_roc']:.3f}")
    print(f"   • F1-Score (malos pagadores): {best_combo['f1_0']:.3f}")
    print(f"   • Puntuación ponderada: {best_combo['weighted_score']:.3f}")

    print(f"\n💡 RECOMENDACIÓN ESTRATÉGICA:")
    print(f"   Para detectar eficientemente malos pagadores, use:")
    print(f"   → Preprocesamiento: {best_combo['preprocessing']}")
    print(f"   → Algoritmo: {best_combo['best_model']}")
    print(f"   → Este enfoque detecta correctamente el {best_combo['sensitivity_0']*100:.1f}%
de los malos pagadores")

    # Análisis del costo/beneficio
    fn_cost = best_combo['fn']  # Malos pagadores clasificados como buenos (PÉRDIDA)
    fp_cost = best_combo['fp']  # Buenos pagadores clasificados como malos
(OPORTUNIDAD PERDIDA)

    print(f"\n💰 ANÁLISIS DE COSTOS:")
    print(f"   • Malos pagadores no detectados (FN): {fn_cost} → PÉRDIDA DIRECTA")
    print(f"   • Buenos pagadores rechazados (FP): {fp_cost} → OPORTUNIDAD PERDIDA")

    # Recomendación basada en el tipo de preprocesamiento
    if 'arboles' in best_combo['preprocessing'] or 'ensamble' in best_combo['preprocessing']:
        print(f"   → Ventaja: Los modelos basados en árboles capturan mejor interacciones no
lineales")
        print(f"   → Ideal para: Detectar patrones complejos en el comportamiento de pago")
    elif 'red_neuronal' in best_combo['preprocessing']:
        print(f"   → Ventaja: La red neuronal maneja bien relaciones complejas y embeddings")
        print(f"   → Ideal para: Capturar relaciones no lineales sofisticadas")
```

```
    else:
        print(f"   → Ventaja: La regresión logística ofrece buena interpretabilidad")
        print(f"   → Ideal para: Entender factores específicos que afectan el riesgo")

    return best_combo

if __name__ == "__main__":
    print("🤖 INICIANDO ANÁLISIS COMPARATIVO CON LAZYPREDICT")
    print("🎯 Objetivo: Encontrar la mejor combinación preprocesamiento-modelo")
    print("   para detectar malos pagadores (clase 0)")
    print("📊 Métricas clave: Sensitivity y AUC-PR (prioridad detectar malos pagadores)")

    # Ejecutar análisis completo
    results = lazy_modeling_analysis()

    # Generar recomendación final
    best_combo = final_recommendation(results)

    if best_combo is not None:
        print(f"\n✅ ANÁLISIS COMPLETADO - MEJOR ELECCIÓN IDENTIFICADA")
    else:
        print(f"\n❌ ANÁLISIS COMPLETADO - NO SE ENCONTRÓ COMBINACIÓN
VÁLIDA")
```

# MODELING

```
# modeling.py
"""
Modelo de riesgo crediticio para deployment web
Autor: [Tu nombre]
Fecha: [Fecha actual]

Este script entrena, evalúa y guarda un modelo de machine learning para predecir
riesgo crediticio. Utiliza el mejor preprocesamiento identificado en el benchmarking
y un modelo XGBoost optimizado para balancear rendimiento, interpretabilidad y capacidad
de implementación en web.

Flujo del pipeline:
1. Cargar datos preprocesados
2. Dividir en conjunto de entrenamiento y prueba
3. Optimizar hiperparámetros con RandomizedSearchCV
4. Entrenar modelo final
5. Calibrar probabilidades
6. Evaluar métricas de rendimiento
7. Analizar equidad y sesgos
8. Generar visualizaciones explicativas
9. Guardar artefactos para deployment web
"""

import os
import json
import pickle
import warnings
import numpy as np
import pandas as pd
import matplotlib
# Establecer backend de matplotlib adecuado para entornos sin GUI
matplotlib.use('Agg')  # Usar backend no interactivo
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime
from typing import Dict, List, Tuple, Any

# Modelado y optimización
from sklearn.model_selection import train_test_split, RandomizedSearchCV, StratifiedKFold
from sklearn.calibration import CalibratedClassifierCV, calibration_curve
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score, roc_auc_score,
    average_precision_score, confusion_matrix, classification_report, roc_curve,
    precision_recall_curve
)
```

```
from sklearn.utils.class_weight import compute_sample_weight

# Modelos
import xgboost as xgb
from xgboost import XGBClassifier

# Explicabilidad
import shap

# Fairness (si está disponible, sino manejar excepción)
try:
    from aif360.datasets import BinaryLabelDataset #type: ignore
    from aif360.metrics import ClassificationMetric #type: ignore
    AIF360_AVAILABLE = True
except ImportError:
    AIF360_AVAILABLE = False
    print("⚠️ Librería aif360 no disponible. El análisis de fairness podría estar limitado.")

# Configuración inicial
warnings.filterwarnings('ignore')
sns.set(style='whitegrid', palette='muted', font_scale=1.1)
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['figure.dpi'] = 100
plt.rcParams['savefig.bbox'] = 'tight'

# Directorios del proyecto
PROJECT_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
DATA_DIR = os.path.join(PROJECT_DIR, 'Data', 'processed')
MODELS_DIR = os.path.join(PROJECT_DIR, 'Docs', 'models')
PLOTS_DIR = os.path.join(PROJECT_DIR, 'Docs')
LOGS_DIR = os.path.join(PROJECT_DIR, 'Logs')

# Crear directorios si no existen
os.makedirs(MODELS_DIR, exist_ok=True)
os.makedirs(PLOTS_DIR, exist_ok=True)
os.makedirs(LOGS_DIR, exist_ok=True)

def print_step(message: str, is_header: bool = False) -> None:
    """Imprime un mensaje con formato para indicar los pasos del pipeline."""
    if is_header:
        print(f"\n{'='*60}")
        print(f"{message.upper()}")
        print(f"{'='*60}")
    else:
        print(f"\n🔍 {message}")
    # Guardar en log
    log_file = os.path.join(LOGS_DIR,
f"modeling_log_{datetime.now().strftime('%Y%m%d')}.txt")
```

```
    os.makedirs(LOGS_DIR, exist_ok=True)
    with open(log_file, 'a') as f:
        f.write(f"{datetime.now().strftime('%Y-%m-%d %H:%M:%S')} - {message}\n")

def load_preprocessed_data(file_path: str) -> Tuple[pd.DataFrame, pd.Series]:
    """Carga los datos preprocesados y separa en features y target."""
    print_step("Cargando datos preprocesados para árboles")

    if not os.path.exists(file_path):
        raise FileNotFoundError(f"No se encontró el archivo: {file_path}")

    df = pd.read_csv(file_path)
    print_step(f"✓ Dataset cargado exitosamente: {df.shape[0]} filas, {df.shape[1]} columnas")

    # Separar características y target
    X = df.drop('target', axis=1)
    y = df['target']

    # Verificación de calidad
    if X.isnull().any().any():
        null_cols = X.columns[X.isnull().any()].tolist()
        print_step(f"⚠️ Advertencia: Se encontraron valores nulos en las columnas:
{null_cols}")
        # Rellenar nulos con la mediana de cada columna
        for col in null_cols:
            if pd.api.types.is_numeric_dtype(X[col]):
                X[col].fillna(X[col].median(), inplace=True)
            else:
                X[col].fillna(X[col].mode()[0], inplace=True)

    print_step(f"✓ Características: {X.shape[1]} features, Target: {y.shape[0]} muestras")
    print_step(f"✓ Distribución del target: {y.value_counts().to_dict()}")

    return X, y

def split_data(X: pd.DataFrame, y: pd.Series, test_size: float = 0.2) -> Tuple:
    """Divide los datos en conjuntos de entrenamiento y prueba de forma estratificada."""
    print_step("Dividiendo datos en conjuntos de entrenamiento y prueba")

    X_train, X_test, y_train, y_test = train_test_split(
        X, y,
        test_size=test_size,
        stratify=y,
        random_state=42
    )

    # Calcular pesos para clases desbalanceadas
    sample_weights = compute_sample_weight('balanced', y_train)
```

```
    print_step(f"✓ Conjunto de entrenamiento: {X_train.shape[0]} muestras")
    print_step(f"✓ Conjunto de prueba: {X_test.shape[0]} muestras")
    print_step(f"✓ Distribución en entrenamiento:
{pd.Series(y_train).value_counts().to_dict()}")
    print_step(f"✓ Distribución en prueba: {pd.Series(y_test).value_counts().to_dict()}")

    return X_train, X_test, y_train, y_test, sample_weights

def optimize_model(X_train: pd.DataFrame, y_train: pd.Series, sample_weights: np.ndarray)
-> XGBClassifier:
    """Optimiza los hiperparámetros del modelo XGBoost usando RandomizedSearchCV."""
    print_step("Optimizando hiperparámetros del modelo XGBoost", is_header=True)

    # Definir espacio de búsqueda para hiperparámetros
    param_dist = {
        'n_estimators': [50, 100, 150, 200],
        'max_depth': [3, 5, 7, 9],
        'learning_rate': [0.01, 0.05, 0.1, 0.2],
        'subsample': [0.6, 0.8, 1.0],
        'colsample_bytree': [0.6, 0.8, 1.0],
        'gamma': [0, 0.1, 0.2, 0.5],
        'min_child_weight': [1, 3, 5, 7],
        'scale_pos_weight': [len(y_train[y_train==1])/len(y_train[y_train==0])]  # Para balancear
clases (inverso al benchmark)
    }

    # Modelo base
    xgb_base = XGBClassifier(
        random_state=42,
        eval_metric='aucpr',  # Priorizar precision-recall para clases desbalanceadas
        use_label_encoder=False,
        tree_method='hist',   # Más rápido y eficiente
        n_jobs=-1             # Usar todos los cores
    )

    # Configurar búsqueda aleatoria
    random_search = RandomizedSearchCV(
        estimator=xgb_base,
        param_distributions=param_dist,
        n_iter=25,            # Número de combinaciones a probar
        cv=StratifiedKFold(n_splits=3, shuffle=True, random_state=42),
        scoring='average_precision',  # Métrica apropiada para clases desbalanceadas
        n_jobs=-1,
        verbose=1,
        random_state=42
    )
```

```
    print_step("Iniciando búsqueda aleatoria de hiperparámetros...")
    print_step(f"Espacio de búsqueda: {len(param_dist)} dimensiones")

    # Entrenar con búsqueda aleatoria
    random_search.fit(X_train, y_train, sample_weight=sample_weights)

    # Mostrar resultados
    print_step("✓ Búsqueda de hiperparámetros completada")
    print_step(f"Mejores hiperparámetros: {random_search.best_params_}")
    print_step(f"Mejor puntuación (average_precision): {random_search.best_score_:.4f}")

    # Entrenar modelo final con los mejores hiperparámetros
    best_model = random_search.best_estimator_
    print_step("Entrenando modelo final con mejores hiperparámetros...")
    best_model.fit(X_train, y_train, sample_weight=sample_weights)

    return best_model

def calibrate_model(model: XGBClassifier, X_train: pd.DataFrame, y_train: pd.Series) ->
CalibratedClassifierCV:
    """Calibra las probabilidades del modelo usando calibración isotónica."""
    print_step("Calibrando probabilidades del modelo", is_header=True)
    print_step("Utilizando calibración isotónica para mejorar la confiabilidad de las
probabilidades")

    # Calibración isotónica (mejor para datasets pequeños/moderados)
    calibrated_model = CalibratedClassifierCV(
        model,
        method='isotonic',  # Mejor para datasets no muy grandes
        cv='prefit'         # Ya pre-entrenado
    )

    calibrated_model.fit(X_train, y_train)
    print_step("✓ Modelo calibrado exitosamente")

    return calibrated_model

def evaluate_model(model: Any, X_test: pd.DataFrame, y_test: pd.Series, model_name: str
= "XGBoost") -> Dict:
    """Evalúa el modelo y genera métricas de rendimiento."""
    print_step("Evaluando rendimiento del modelo", is_header=True)

    # Predicciones
    y_pred = model.predict(X_test)
    y_pred_proba = model.predict_proba(X_test)[:, 1]  # Probabilidades para clase positiva
(buenos pagadores)

    # Métricas básicas
```

```
    metrics = {
        'accuracy': accuracy_score(y_test, y_pred),
        'precision': precision_score(y_test, y_pred),
        'recall': recall_score(y_test, y_pred),
        'f1': f1_score(y_test, y_pred),
        'roc_auc': roc_auc_score(y_test, y_pred_proba),
        'pr_auc': average_precision_score(y_test, y_pred_proba)
    }

    print_step("📊 Métricas de rendimiento:")
    for metric, value in metrics.items():
        print_step(f"• {metric.replace('_', ' ').title()}: {value:.4f}")

    # Matriz de confusión
    cm = confusion_matrix(y_test, y_pred)
    print_step("Matriz de confusión:")
    print(f"              Predicción")
    print(f"           Malo (0)  Bueno (1)")
    print(f"Real Malo    [{cm[0,0]:3d}      {cm[0,1]:3d}]")
    print(f"Real Bueno   [{cm[1,0]:3d}      {cm[1,1]:3d}]")

    # Reporte de clasificación
    print_step("Reporte de clasificación detallado:")
    print(classification_report(y_test, y_pred, target_names=['Malo (0)', 'Bueno (1)']))

    # Curva precision-recall para encontrar umbral óptimo
    precisions, recalls, thresholds = precision_recall_curve(y_test, y_pred_proba)

    # Calcular F1 para cada umbral
    f1_scores = 2 * (precisions * recalls) / (precisions + recalls + 1e-10)

    # Encontrar umbral que maximiza F1
    opt_idx = np.argmax(f1_scores[:-1])  # Excluir último elemento para coincidir con
thresholds
    optimal_threshold = thresholds[opt_idx] if opt_idx < len(thresholds) else 0.5
    print_step(f"\n🔍 Umbral óptimo encontrado para maximizar F1: {optimal_threshold:.4f}")

    # Evaluar con umbral óptimo
    y_pred_opt = (y_pred_proba >= optimal_threshold).astype(int)
    metrics_opt = {
        'accuracy_opt': accuracy_score(y_test, y_pred_opt),
        'precision_opt': precision_score(y_test, y_pred_opt),
        'recall_opt': recall_score(y_test, y_pred_opt),
        'f1_opt': f1_score(y_test, y_pred_opt)
    }

    print_step("📊 Métricas con umbral óptimo:")
    for metric, value in metrics_opt.items():
```

```
        print_step(f"• {metric.replace('_opt', '').replace('_', ' ').title()}: {value:.4f}")

    return {
        'raw_metrics': metrics,
        'opt_metrics': metrics_opt,
        'optimal_threshold': optimal_threshold,
        'y_pred': y_pred,
        'y_pred_proba': y_pred_proba,
        'y_pred_opt': y_pred_opt,
        'thresholds': thresholds,  # Guardar thresholds para las gráficas
        'precisions': precisions,  # Guardar precisions para las gráficas
        'recalls': recalls         # Guardar recalls para las gráficas
    }

def plot_evaluation_curves(model: Any, X_test: pd.DataFrame, y_test: pd.Series, results:
Dict) -> None:
    """Genera gráficos de evaluación: ROC, PR y curva de calibración."""
    print_step("Generando gráficas de evaluación", is_header=True)

    y_pred_proba = results['y_pred_proba']
    optimal_threshold = results['optimal_threshold']
    thresholds = results['thresholds']
    precisions = results['precisions']
    recalls = results['recalls']

    # 1. Curva ROC
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
    plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC =
{results["raw_metrics"]["roc_auc"]:.4f})')
    plt.plot([0, 1], [0, 1], color='gray', lw=1, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('Tasa de Falsos Positivos (1 - Especificidad)')
    plt.ylabel('Tasa de Verdaderos Positivos (Sensibilidad)')
    plt.title('Curva ROC')
    plt.legend(loc="lower right")
    plt.grid(True, alpha=0.3)

    # 2. Curva Precision-Recall
    plt.subplot(1, 2, 2)
    plt.plot(recalls, precisions, color='green', lw=2, label=f'PR curve (AUC =
{results["raw_metrics"]["pr_auc"]:.4f})')

    # Encontrar el índice del umbral óptimo en la curva PR
    opt_idx = np.argmin(np.abs(thresholds - optimal_threshold))
    if opt_idx < len(recalls):
```

```
        plt.scatter(recalls[opt_idx], precisions[opt_idx], color='red', s=100,
                    label=f'Umbral óptimo ({optimal_threshold:.2f})', zorder=5)

    plt.xlabel('Recall')
    plt.ylabel('Precisión')
    plt.title('Curva Precision-Recall')
    plt.legend(loc="lower left")
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig(os.path.join(PLOTS_DIR, 'roc_pr_curves.png'), dpi=300, bbox_inches='tight')
    plt.close()
    print_step("✅ Gráficas ROC y PR generadas y guardadas")

    # 3. Matriz de confusión
    plt.figure(figsize=(8, 6))
    cm = confusion_matrix(y_test, results['y_pred'])
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=['Malo (0)', 'Bueno (1)'],
                yticklabels=['Malo (0)', 'Bueno (1)'])
    plt.xlabel('Predicción')
    plt.ylabel('Real')
    plt.title('Matriz de Confusión')
    plt.savefig(os.path.join(PLOTS_DIR, 'confusion_matrix.png'), dpi=300,
bbox_inches='tight')
    plt.close()
    print_step("✅ Matriz de confusión generada y guardada")

    # 4. Curva de calibración
    plt.figure(figsize=(10, 6))
    prob_true, prob_pred = calibration_curve(y_test, y_pred_proba, n_bins=10)
    plt.plot(prob_pred, prob_true, marker='o', label='Modelo calibrado')
    plt.plot([0, 1], [0, 1], linestyle='--', color='gray', label='Perfectamente calibrado')
    plt.xlabel('Probabilidad promedio predicha')
    plt.ylabel('Fracción de positivos')
    plt.title('Curva de Calibración')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.savefig(os.path.join(PLOTS_DIR, 'calibration_curve.png'), dpi=300,
bbox_inches='tight')
    plt.close()
    print_step("✅ Curva de calibración generada y guardada")

    # 5. Distribución de probabilidades
    plt.figure(figsize=(10, 6))
    plt.hist(y_pred_proba[y_test == 0], bins=30, alpha=0.5, label='Malo (0)', color='red')
    plt.hist(y_pred_proba[y_test == 1], bins=30, alpha=0.5, label='Bueno (1)', color='green')
```

```
    plt.axvline(x=optimal_threshold, color='black', linestyle='--', label=f'Umbral óptimo =
{optimal_threshold:.2f}')
    plt.xlabel('Probabilidad predicha de ser Bueno (1)')
    plt.ylabel('Frecuencia')
    plt.title('Distribución de Probabilidades por Clase')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.savefig(os.path.join(PLOTS_DIR, 'probability_distribution.png'), dpi=300,
bbox_inches='tight')
    plt.close()
    print_step("✅ Distribución de probabilidades generada y guardada")

def analyze_fairness(model: Any, X_test: pd.DataFrame, y_test: pd.Series, y_pred:
np.ndarray, threshold: float) -> None:
    """Analiza la equidad del modelo en diferentes grupos demográficos."""
    print_step("Analizando equidad del modelo", is_header=True)

    # Crear una copia del dataset de prueba para análisis
    df = X_test.copy()
    df['target'] = y_test
    df['predicted'] = y_pred

    # Identificar columnas relacionadas con edad y género
    age_cols = [col for col in X_test.columns if 'age' in col.lower()]
    sex_cols = [col for col in X_test.columns if ('sex' in col.lower() or 'gender' in col.lower() or
'status_sex' in col.lower())]

    # Variables de sustitución para análisis de fairness
    substitute_cols = {
        'age': age_cols[0] if age_cols else None,
        'sex': sex_cols[0] if sex_cols else None
    }

    # Si no encontramos columnas específicas, buscar alternativas
    if not substitute_cols['age']:
        # Buscar columna con rango de edad o similar
        age_related = [col for col in X_test.columns if any(term in col.lower() for term in ['years',
'duration', 'residence'])]
        substitute_cols['age'] = age_related[0] if age_related else None

    if not substitute_cols['sex']:
        # Buscar columnas categóricas que puedan indicar género indirectamente
        categorical_cols = X_test.select_dtypes(include=['object', 'category']).columns.tolist()
        if categorical_cols:
            substitute_cols['sex'] = categorical_cols[0]

    # Realizar análisis de fairness básico incluso sin aif360
    if substitute_cols['age']:
```

```
        print_step(f"\n👥 Análisis de equidad usando '{substitute_cols['age']}' como proxy de edad:")

        # Dividir en grupos por percentiles
        df['age_group'] = pd.qcut(df[substitute_cols['age']], q=2, labels=['Grupo 1', 'Grupo 2'])

        groups = df['age_group'].unique()
        for group in groups:
            group_data = df[df['age_group'] == group]
            if len(group_data) > 10:
                group_pred = group_data['predicted']
                group_target = group_data['target']

                dp = group_pred.mean()  # Demographic Parity (tasa de positivos)
                eo = recall_score(group_target, group_pred)  # Equal Opportunity (TPR)

                print_step(f"  • {group} ({len(group_data)} muestras):")
                print_step(f"    - Demographic Parity (tasa de aprobación): {dp:.4f}")
                print_step(f"    - Equal Opportunity (recall): {eo:.4f}")
    else:
        print_step("⚠️ No se encontró una columna adecuada para analizar equidad por edad.")

    if substitute_cols['sex']:
        print_step(f"\n👥 Análisis de equidad usando '{substitute_cols['sex']}' como proxy de género:")

        # Tomar los dos valores más comunes para crear grupos
        common_values = df[substitute_cols['sex']].value_counts().index[:2]
        df['sex_group'] = df[substitute_cols['sex']].apply(lambda x: 'Grupo A' if x == common_values[0] else ('Grupo B' if x == common_values[1] else 'Otros'))

        for group in ['Grupo A', 'Grupo B']:
            group_data = df[df['sex_group'] == group]
            if len(group_data) > 10:
                group_pred = group_data['predicted']
                group_target = group_data['target']

                dp = group_pred.mean()  # Demographic Parity (tasa de positivos)
                eo = recall_score(group_target, group_pred)  # Equal Opportunity (TPR)

                print_step(f"  • {group} ({len(group_data)} muestras):")
                print_step(f"    - Demographic Parity (tasa de aprobación): {dp:.4f}")
                print_step(f"    - Equal Opportunity (recall): {eo:.4f}")
    else:
        print_step("⚠️ No se encontró una columna adecuada para analizar equidad por género.")
```

```
def explain_model(model: XGBClassifier, X_train: pd.DataFrame, X_test: pd.DataFrame,
feature_names: List[str]) -> None:
    """Genera explicaciones globales y locales del modelo usando SHAP."""
    print_step("Generando explicaciones del modelo con SHAP", is_header=True)

    # Explicabilidad global con SHAP
    print_step("Calculando valores SHAP para explicabilidad global...")
    try:
        explainer = shap.TreeExplainer(model)

        # Tomar muestra para cálculo eficiente
        sample_size = min(100, len(X_test))
        X_sample = shap.sample(X_test, sample_size, random_state=42)

        shap_values = explainer.shap_values(X_sample)

        # Resumen de importancia global
        plt.figure(figsize=(12, 8))
        shap.summary_plot(shap_values, X_sample, feature_names=feature_names,
plot_type="bar", show=False)
        plt.title('Importancia Global de Características')
        plt.tight_layout()
        plt.savefig(os.path.join(PLOTS_DIR, 'feature_importance_global.png'), dpi=300,
bbox_inches='tight')
        plt.close()
        print_step("✅ Importancia global de características generada y guardada")

        # Gráfico de resumen SHAP
        plt.figure(figsize=(12, 10))
        shap.summary_plot(shap_values, X_sample, feature_names=feature_names,
show=False)
        plt.title('Valores SHAP - Resumen')
        plt.tight_layout()
        plt.savefig(os.path.join(PLOTS_DIR, 'shap_summary.png'), dpi=300,
bbox_inches='tight')
        plt.close()
        print_step("✅ Gráfico de resumen SHAP generado y guardado")

        # Explicaciones locales para casos específicos
        print_step("\n🔍 Explicaciones locales para casos representativos:")

        # Convertir a DataFrame para facilitar el manejo
        X_sample_df = X_sample.copy() if isinstance(X_sample, pd.DataFrame) else
pd.DataFrame(X_sample, columns=feature_names)
        y_test_sample = model.predict(X_sample)
        y_pred_proba_sample = model.predict_proba(X_sample)[:, 1]

        # Caso 1: Alto riesgo (mal pagador predicho correctamente)
```

```
        high_risk_indices = np.where((y_test_sample == 0))[0]
        if len(high_risk_indices) > 0:
            high_risk_idx = high_risk_indices[0]
            print_step(f"  • Caso de alto riesgo (índice {high_risk_idx})")

            plt.figure(figsize=(10, 6))
            shap.waterfall_plot(
                shap.Explanation(
                    values=shap_values[high_risk_idx],
                    base_values=explainer.expected_value,
                    data=X_sample_df.iloc[high_risk_idx].values,
                    feature_names=feature_names
                ),
                max_display=10,
                show=False
            )
            plt.title(f'Explicación SHAP - Alto Riesgo (Índice {high_risk_idx})')
            plt.tight_layout()
            plt.savefig(os.path.join(PLOTS_DIR, 'shap_alto_riesgo.png'), dpi=300,
bbox_inches='tight')
            plt.close()
            print_step("✅ Explicación SHAP para caso de alto riesgo generada y guardada")

        # Caso 2: Frontera (probabilidad cerca del umbral óptimo)
        border_indices = np.argsort(np.abs(y_pred_proba_sample - 0.5))[:3]
        if len(border_indices) > 0:
            border_idx = border_indices[0]
            print_step(f"  • Caso frontera (índice {border_idx},
prob={y_pred_proba_sample[border_idx]:.4f})")

            plt.figure(figsize=(10, 6))
            shap.waterfall_plot(
                shap.Explanation(
                    values=shap_values[border_idx],
                    base_values=explainer.expected_value,
                    data=X_sample_df.iloc[border_idx].values,
                    feature_names=feature_names
                ),
                max_display=10,
                show=False
            )
            plt.title(f'Explicación SHAP - Caso Frontera (Índice {border_idx})')
            plt.tight_layout()
            plt.savefig(os.path.join(PLOTS_DIR, 'shap_caso_frontera.png'), dpi=300,
bbox_inches='tight')
            plt.close()
            print_step("✅ Explicación SHAP para caso frontera generada y guardada")
```

```
        # Caso 3: Error (seleccionar un caso incorrectamente clasificado si existe)
        wrong_indices = np.where(y_test_sample != model.predict(X_sample))[0]
        if len(wrong_indices) > 0:
            wrong_idx = wrong_indices[0]
            print_step(f"  • Caso mal clasificado (índice {wrong_idx})")

            plt.figure(figsize=(10, 6))
            shap.waterfall_plot(
                shap.Explanation(
                    values=shap_values[wrong_idx],
                    base_values=explainer.expected_value,
                    data=X_sample_df.iloc[wrong_idx].values,
                    feature_names=feature_names
                ),
                max_display=10,
                show=False
            )
            plt.title(f'Explicación SHAP - Mal Clasificado (Índice {wrong_idx})')
            plt.tight_layout()
            plt.savefig(os.path.join(PLOTS_DIR, 'shap_mal_clasificado.png'), dpi=300,
bbox_inches='tight')
            plt.close()
            print_step("✅ Explicación SHAP para caso mal clasificado generada y guardada")

    except Exception as e:
        print_step(f"⚠️ Error al generar explicaciones SHAP: {str(e)}")
        print_step("Continuando sin explicabilidad detallada...")

def save_artifacts(model: Any, calibrated_model: Any, X_train: pd.DataFrame, results: Dict,
feature_names: List[str]) -> Dict:
    """Guarda artefactos necesarios para deployment web."""
    print_step("Guardando artefactos para deployment web", is_header=True)

    # 1. Guardar modelo calibrado
    model_path = os.path.join(MODELS_DIR, 'credit_risk_model.pkl')
    with open(model_path, 'wb') as f:
        pickle.dump(calibrated_model, f)
    print_step(f"✓ Modelo guardado en: {model_path}")

    # 2. Guardar diccionario de características
    if hasattr(calibrated_model._get_estimator, 'feature_importances_'):
        feature_importances = calibrated_model._get_estimator.feature_importances_.tolist()
    else:
        feature_importances = [1/len(feature_names)] * len(feature_names)  # Importancias
uniformes como fallback

    feature_dict = {
        'features': feature_names,
```

```
    'feature_importances': feature_importances,
    'optimal_threshold': results['optimal_threshold'],
    'class_names': ['Malo (0)', 'Bueno (1)'],
    'timestamp': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
    'model_type': 'XGBoost Calibrado'
}

feature_dict_path = os.path.join(MODELS_DIR, 'feature_dictionary.csv')
pd.DataFrame([feature_dict]).to_csv(feature_dict_path, index=False)
print_step(f"✓ Diccionario de características guardado en: {feature_dict_path}")

# 3. Guardar configuración para web
web_config = {
    'model_name': 'Modelo de Riesgo Crediticio',
    'version': '1.0',
    'threshold': float(results['optimal_threshold']),
    'features': [
        {'name': col, 'type': 'numerical', 'description': f'Variable {col}'}
        for col in feature_names
    ],
    'target_description': {
        '0': 'Alto riesgo - No aprobar crédito',
        '1': 'Bajo riesgo - Aprobar crédito'
    },
    'metrics': {
        'test_recall': float(results['raw_metrics']['recall']),
        'test_precision': float(results['raw_metrics']['precision']),
        'test_f1': float(results['raw_metrics']['f1']),
        'test_roc_auc': float(results['raw_metrics']['roc_auc']),
        'test_pr_auc': float(results['raw_metrics']['pr_auc'])
    }
}

web_config_path = os.path.join(MODELS_DIR, 'web_config.json')
with open(web_config_path, 'w') as f:
    json.dump(web_config, f, indent=2)
print_step(f"✓ Configuración para web guardada en: {web_config_path}")

# 4. Guardar todas las métricas
metrics_path = os.path.join(MODELS_DIR, 'model_metrics.json')
with open(metrics_path, 'w') as f:
    json.dump({
        'model_name': 'XGBoost Calibrado',
        'timestamp': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
        **results['raw_metrics'],
        **results['opt_metrics'],
        'optimal_threshold': float(results['optimal_threshold'])
    }, f, indent=2)
```

```
    print_step(f"✓ Métricas del modelo guardadas en: {metrics_path}")

    return web_config

def generate_report(web_config: Dict) -> None:
    """Genera un reporte resumen del modelo para documentación."""
    print_step("Generando reporte final del modelo", is_header=True)

    report = f"""
    REPORTE DEL MODELO DE RIESGO CREDITICIO
    {'='*60}
    Fecha de generación: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}
    Modelo: {web_config['model_name']} (versión {web_config['version']})

    DESEMPEÑO EN CONJUNTO DE PRUEBA:
    {'-'*60}
    • Recall (Capacidad para detectar malos pagadores):
{web_config['metrics']['test_recall']:.4f}
    • Precisión (Exactitud en aprobaciones): {web_config['metrics']['test_precision']:.4f}
    • F1-Score (Balance): {web_config['metrics']['test_f1']:.4f}
    • AUC-ROC (Capacidad de discriminación): {web_config['metrics']['test_roc_auc']:.4f}
    • AUC-PR (Robustez ante desbalance): {web_config['metrics']['test_pr_auc']:.4f}

    CONFIGURACIÓN DEL MODELO:
    {'-'*60}
    • Umbral de decisión óptimo: {web_config['threshold']:.4f}
    • Características utilizadas: {len(web_config['features'])}
    • Clases objetivo: {', '.join(web_config['target_description'].values())}

    RECOMENDACIONES OPERATIVAS:
    {'-'*60}
    • Utilizar el umbral óptimo de {web_config['threshold']:.4f} para balancear detección de
riesgo y oportunidades de negocio
    • Monitorear periódicamente las métricas de desempeño y equidad
    • Realizar recalibración cada 6 meses o cuando se observe desviación significativa en las
métricas
    • Revisar las decisiones en los casos frontera (probabilidades cercanas al umbral)
    • Auditar las decisiones para diferentes segmentos demográficos para garantizar equidad

    ARTEFACTOS GENERADOS:
    {'-'*60}
    • Modelo entrenado: {os.path.join(MODELS_DIR, 'credit_risk_model.pkl')}
    • Configuración para web: {os.path.join(MODELS_DIR, 'web_config.json')}
    • Diccionario de características: {os.path.join(MODELS_DIR, 'feature_dictionary.csv')}
    • Métricas completas: {os.path.join(MODELS_DIR, 'model_metrics.json')}
    • Visualizaciones: {PLOTS_DIR}
    """
```

```
    report_path = os.path.join(MODELS_DIR, 'model_report.txt')
    with open(report_path, 'w') as f:
        f.write(report)

    print_step(report)
    print_step(f"✅ Reporte completo guardado en: {report_path}")

if __name__ == "__main__":
    # Inicio del pipeline
    print_step("INICIANDO PIPELINE DE MODELADO PARA RIESGO CREDITICIO",
is_header=True)
    print_step("Objetivo: Desarrollar un modelo web-friendly para predecir riesgo crediticio")

    try:
        # 1. Cargar datos
        file_path = os.path.join(DATA_DIR, 'data_preprocesada_arboles.csv')
        X, y = load_preprocessed_data(file_path)

        # 2. Dividir datos
        X_train, X_test, y_train, y_test, sample_weights = split_data(X, y)

        # 3. Optimizar modelo
        model = optimize_model(X_train, y_train, sample_weights)

        # 4. Calibrar modelo
        calibrated_model = calibrate_model(model, X_train, y_train)

        # 5. Evaluar modelo
        results = evaluate_model(calibrated_model, X_test, y_test)

        # 6. Generar gráficas de evaluación
        plot_evaluation_curves(calibrated_model, X_test, y_test, results)

        # 7. Analizar equidad
        analyze_fairness(calibrated_model, X_test, y_test, results['y_pred'],
results['optimal_threshold'])

        # 8. Explicabilidad con SHAP
        explain_model(model, X_train, X_test, X_train.columns.tolist())

        # 9. Guardar artefactos para web
        web_config = save_artifacts(model, calibrated_model, X_train, results,
X_train.columns.tolist())

        # 10. Generar reporte
        generate_report(web_config)
```

```
        print_step("PIPELINE DE MODELADO COMPLETADO EXITOSAMENTE",
is_header=True)
        print_step("El modelo está listo para ser implementado en la aplicación web")
        print_step(f"Artefactos guardados en: {MODELS_DIR}")

    except Exception as e:
        print_step(f"❌ ERROR CRÍTICO DURANTE LA EJECUCIÓN: {str(e)}",
is_header=True)
        import traceback
        traceback.print_exc()
        print_step("Se ha generado un log de errores en el directorio Logs/")
```