Startup Succes Prediction Model II

Aprendizaje Automático II

Máster en Informática Industrial y Robótica

Víctor Caínzos López & David Adrián Rodríguez García

Introducción

El proyecto consiste en una segunda versión del código desarrollado en la primera parte del trabajo:

- Startup Succes Prediction Model

Utilizando el mismo conjunto de datos, pero ampliando el contenido con nuevas técnicas de reducción de la dimensionalidad de los datos y aprendizaje no supervisado: clustering, detección de anomalías.

En este documento se incluye la descripción de las ampliaciones realizadas y resultados obtenidos.

Índice

1. Técnicas de aprendizaje

Se han implementado técnicas de reducción de la dimensionalidad de los datos (PCA), agrupamiento (clustering) y detección de anomalías.

1.1. Reducción de la dimensionalidad

Se define una nueva clase en la que se incluyen los métodos apropiados para cada técnica.

```
class Dimension:
    """Class with implemented techniques for reducing dimensions of features"""
```

Matriz de correlación

Para estudiar la correlación entre las variables del conjunto de datos se incluye el siguiente método que cálcula la matriz de correlación.

```
@staticmethod
    def get_correlation_matrix(X: pd.DataFrame):
        """Calculates the correlation matriz of data features

        :param X: Dataset of features and samples
        :type X: pd.DataFrame
        :return: Dataframe with correlation between variables
        :rtype: pd.DataFrame
        """
        samples, nvar = X.shape
        corr_mat = np.corrcoef(np.c_[X].T)
        etiquetas = X.columns.values.tolist()
        # DataFrame with correlations between variables
        return pd.DataFrame(corr_mat, columns=etiquetas, index=etiquetas)
```

PCA

También se desarrolla un estudio de los componentes principales del dataset empleado, analizando aquellos que expliquen la mayor varianza de los mismos, para conocer qué variables son de mayor relevancia de cara a reducir la dimensionalidad del problema.

```
@staticmethod
    def get_PCA(X: pd.DataFrame, **kwargs):
        """Applies PCA technique to reduce dimensionality

        :param X: Dataset of features and samples
        :type X: pd.DataFrame
        :return: PCA results
        :rtype: obj
        """
```

```
        samples, nvar = X.shape
        pca = PCA(**kwargs)
        pca.fit(X)
        return pca
```

1.2. Clustering

Análogamente, se diseña una clase que tenga embebida las técnicas de agrupamiento de iterés cuando se afronta el proyecto en base a este problema.

```
class Clustering:
    """Class with clustering techniques implemented for unsupervised learning"""
```

K-means

Dentro de esta clase se incluyen una serie de métodos que utilizan como herramientas técnicas de aprendizaje no supervisado que son aplicables al conjunto de datos cuando no se tiene en cuenta las etiquetas. A continuación se presenta el principal método de agrupamiento por particiones, el K-means. El codigo fuente del método devuelve las inercias o distancias al cuadrado de cada punto con su centroide más cercano.

```
@staticmethod
    def perform_kmeans(X: pd.DataFrame, **kwargs):
        """Static method wrapped in Clustering class which performs Kmeans
algorithm

        :param X: Dataset with samples
        :type X: pd.DataFrame
        :return: Inertias from data samples to clusters
        :rtype: dict
        """
        k_list = np.arange(1, len(X.columns) + 1)
        inertia_dic = {}
        for k in k_list:
            kmeans = KMeans(n_clusters=k, random_state=0).fit(X)
            inertia_dic[k] = kmeans.inertia_
        return inertia_dic
```

DBSCAN

Además del K-means, también se define un método referido al agrupamiento por densidad de los datos que utiliza la técnica DBSCAN. En el siguiente bloque de código se calculan las distancias de los k-vecinos más cercanos, siendo k el doble del número de variables, y haciendo uso de este algoritmo para estudiar el valor óptimo del hiperparámetro epsilon que define la densidad de cada punto.

```
@staticmethod
    def perform_DBSCAN(X: pd.DataFrame, **kwargs):
        """Static method wrapped in Clustering class which performs

        :param X: Dataset with samples
        :type X: pd.DataFrame
        :return: Distances for all data to its k-nearest-neighbor
        :rtype: list
        """
        neighbors = NearestNeighbors(n_neighbors=2 * len(X.columns))
        neighbors_fit = neighbors.fit(X)
        distances, _ = neighbors_fit.kneighbors(X)
        distances = np.sort(distances, axis=0)
        distances = distances[:, -1]
        return distances
```

1.3 Detección de anomalías

Para finalizar con el apartado de nuevas técnicas de aprendizaje, se crea una nueva clase específica para los métodos dedicados a la detección de anomalías.

```
class Anomalies:
    """Class with clustering techniques implemented for unsupervised learning."""
```

En este caso, dado que el problema principal no fue concebido para afrontarse según este criterio, es necesario manipular el conjunto de datos considerando una clase mayoritaria como normal y otra minoritaria como anómala. Esta labor la lleva a cabo el constructor de la clase, adaptando y creando un escenario en la detección de anomalías donde Los datos de entrenamiento solo contienen datos de tipo normal, mientras que los datos de test pueden contener anomalías.

Isolation Forest

El primer método que se presenta ejecuta el procedimiento conocido como Isolation Forest, utilizando el algoritmo empleado en los bosques aleatorios para aislar los datos con distintas características del resto. El código devuelve un informe con los resultados de la clasificación.

```
def perform_IsolationForest(
        self, n_estimators=100, target_names=None, contamination=0, **kwargs
    ):
        """Isolation Forest algorithm.

        :param n_estimators: number of estimators used, defaults to 100
        :type n_estimators: int, optional
        :param contamination: percentage os samples included as anomalies in
trining set, defaults to 0
        :type contamination: int, optional
        :return: classification report with results
```

```
        :rtype: str
        """
        model = IsolationForest(
            n_estimators=n_estimators, contamination=contamination, **kwargs
        )
        model.fit(self.X_train)
        y_test = model.predict(self.X_test)
        return classification_report(
            self.t_test, y_test, target_names=target_names
        )
```

LOF

También se define un método basado que utiliza el algoritmo Local Outlier Factor (LOF) y la densidad de los datos como factor para detectar valores atípicos, comparándola con la de sus vecinos. De igual manera, el método definido devuelve un informe de clasificación.

```
def perform_LOF(self, n_neighbors=10, target_names=None, novelty=True):
        """LOF algorithm.

        :param n_neighbors: number of data neighbours used, defaults to 10
        :type n_neighbors: int, optional
        :param novelty: param necessary to detect anomalies, defaults to True
        :type novelty: bool, optional
        :return: classification report with results
        :rtype: str
        """
        model = LocalOutlierFactor(n_neighbors=n_neighbors, novelty=novelty)
        model.fit(self.X_train)
        y_test = model.predict(self.X_test)
        return classification_report(
            self.t_test, y_test, target_names=target_names
        )
```

Red de neuronas autoencoder

Terminando con los métodos definidos en la clase específica para la detección de anomalias, se incluyen una serie de funciones para la creación y utilización de una red neuronas autoencoder, que comprime los datos a una dimensión inferior, y luego a la salida del modelo, intenta reconstruir los datos de entrada, extrayendo la información esencial.

```
def perform_autoencoding(self):
        """Create an autoencoder neural network."""
        _, variables = self.X_train.shape
        autoencoder = models.Sequential()
        autoencoder.add(layers.Dense(1, input_dim=variables, activation="relu"))
        autoencoder.add(layers.Dense(variables, activation="relu"))
        opt = keras.optimizers.Adam(learning_rate=0.001)
```

```
        autoencoder.compile(loss="mean_squared_error", optimizer=opt)
        self.autoencoder = autoencoder
```

Durante el entrenamiento, la capa oculta aprenderá la representación de los datos de entrada normales y en la fase de test, una anomalía será diferente a un dato normal por lo que la red tendrá problemas para reconstruir el dato anómalo, provocando que el error a la salida sea alto.

```
def train_autoencoding(self, epochs=200, batch_size=100, **kwargs):
        """Fits autoencoder neural network.

        :param epochs: number of times all data is passed to network, defaults to
200
        :type epochs: int, optional
        :param batch_size: number of splits of data in batchs, defaults to 100
        :type batch_size: int, optional
        """
        self.history = self.autoencoder.fit(
            self.X_train,
            self.X_train,
            validation_data=(self.X_test, self.X_test),
            **kwargs
        )
```

Se define una función encargada de graficar la curva de validación con los resultados de entrenamiento y test, utilizada posteriormente a la hora de generar el report donde se recogen todos los resultados de los experimentos realizados.

```
def plot_autoencoder_validation(
        self,
        xlabel: str = "Mean Square Error (MSE)",
        ylabel: str = "Iteration (epoch)",
        legend: tuple = ("Entrenamiento", "Test"),
        figsize: tuple = (12, 4),
    ):
        """Plot autoencoder validation curve.

        :param xlabel: plot xlabel, defaults to "Mean Square Error (MSE)"
        :type xlabel: str, optional
        :param ylabel: plot ylabel, defaults to "Iteration (epoch)"
        :type ylabel: str, optional
        :param legend: plot legend, defaults to ("Entrenamiento", "Test")
        :type legend: tuple, optional
        :param figsize: size of plot, defaults to (12, 4)
        :type figsize: tuple, optional
        :return: plot figure
        :rtype: obj
        """
        fig, ax = plt.subplots(figsize=figsize)
        ax.plot(self.history.history["loss"])
```

```
        ax.plot(self.history.history["val_loss"])
        ax.set_ylabel(xlabel)
        ax.set_xlabel(ylabel)
        ax.legend(legend, loc="upper right")
        return fig
```

Para etiquetar un nuevo dato como anómalo se emplea un umbral para el error de reconstrucción, determinado en base a los errores de entrenamiento.

```
def plot_autoencoder_threshold(
        self,
        xlabel: str = "Reconstruction error (training)",
        ylabel: str = "Number of data",
        legend: tuple = ("Threshold"),
        figsize: tuple = (12, 4),
    ):
        """Predicts anomalies by recontructing samples passed to encoder.

        :param xlabel: xlabel plot, defaults to "Reconstruction error (training)"
        :type xlabel: str, optional
        :param ylabel: ylabel plot, defaults to "Number of data"
        :type ylabel: str, optional
        :param legend: legend plot, defaults to ("Threshold")
        :type legend: tuple, optional
        :param figsize: size of plot, defaults to (12, 4)
        :type figsize: tuple, optional
        :return: figure
        :rtype: obj
        """
        y_train = self.autoencoder.predict(self.X_train)
        self.mse_train = np.mean(np.power(self.X_train - y_train, 2), axis=1)
        threshold = np.max(self.mse_train)

        fig, ax = plt.subplots(figsize=figsize)
        ax.hist(self.mse_train, bins=50)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.axvline(threshold, color="r", linestyle="--")
        ax.legend(legend, loc="upper center")
        return fig
```

Adicionalmente, se define una función gráfica para la detección en función del umbral seleccionado.

```
def plot_autoencoder_error(
        self,
        xlabel: str = "Data Index",
        ylabel: str = "Reconstruction Error",
        legend: tuple = ("Training", "Test", "Threshold"),
        figsize: tuple = (12, 4),
```

```
    ):
        """Plot autoencoder error.

        :param xlabel: xlabel plot, defaults to "Data Index"
        :type xlabel: str, optional
        :param ylabel: ylabel plot, defaults to "Reconstruction Error"
        :type ylabel: str, optional
        :param legend: legend plot, defaults to ("Training", "Test", "Threshold")
        :type legend: tuple, optional
        :param figsize: size of figure, defaults to (12, 4)
        :type figsize: tuple, optional
        :return: figure
        :rtype: obj
        """
        e_test = self.autoencoder.predict(self.X_test)
        self.mse_test = np.mean(np.power(self.X_test - e_test, 2), axis=1)
        threshold = np.max(self.mse_train)
        fig, ax = plt.subplots(figsize=(12, 4))
        ax.plot(range(1, self.X_train.shape[0] + 1), self.mse_train, "b.")
        ax.plot(
            range(
                self.X_train.shape[0] + 1,
                self.X_train.shape[0] + self.X_test.shape[0] + 1,
            ),
            self.mse_test,
            "r.",
        )
        ax.axhline(threshold, color="r", linestyle="--")
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.legend(legend, loc="upper left")
        return fig
```

Y para generar un informe con los resultados de clasificación.

```
def get_autoencoder_clreport(self, target_names=None):
        """Generate classification report with autoencoders results.

        :return: classification report with results
        :rtype: str
        """
        y_test = np.ones((self.t_test.shape))
        threshold = np.max(self.mse_train)
        y_test[self.mse_test > threshold] = -1

        return classification_report(
            self.t_test, y_test, target_names=target_names
        )
```

2. Ejecución de tareas en paralelo

En este apartado se explican las partes del programa en las que se ha implementado la posibilidad de realizar las declaraciones en paralelo. Concretamente utilizando multithreading o programación en hilos de ejecución.

Este procedimiento se implementa en aquellas zonas en las que se llevan a cabo tareas recurrentes como el entrenamiento de los modelos o la red neuronal utilizando validación cruzada. De esta manera se adapta y amplia el programa para aceptar la ejecución de esta parte del codigo fuente de forma secuencial o en varios hilos, según especifique el usuario por teclado junto con el número de intentos por modelo y particiones o kfolds en la validación cruzada. Es importante anotar en este caso el interés de introducir unos datos sustaciales forma que se obtengan soluciones representativas para los modelos evaluados.

Para el caso de los modelos de Scikit-Learn, se define una versión de la función original de optimización donde se utilizan tantos hilos como intentos y modelos se tengan, es decir, un hilo por cada versión del modelo a optimizar.

```
def perform_optimizing_models_multithread(
        self,
        tag,
        X: np.ndarray,
        t: np.ndarray,
        train_size: float = 0.85,
        scoring: dict = {
            "accuracy": "accuracy",
            "recall": "recall",
            "specificity": make_scorer(recall_score, pos_label=0),
            "precision": "precision",
            "f1": "f1",
            "roc_auc": "roc_auc",
        },
        cv: int = 10,
        trials: int = 25,
    ) -> dict:
        """Optimize hyperparameters of the given model and returns the best of
them using cross validation and multithreading.

        :param tag: model to be evaluated.
        :type tag: str
        :param X: Characteristic matrix numpy array of the dataset which will be
evaluated
        :type X: np.ndarray
        :param t: Vector labels numpy array of the dataset which will be evaluated
        :type t: np.ndarray
        :param train_size: % of the data to be splitted into train and test
values, defaults to 0.85
        :type train_size: float, optional
        :param scoring: A dictionary with the wanted metrics to compare and
evaluate the different models, defaults to { "accuracy": "accuracy", "recall":
"recall", "specificity": make_scorer(recall_score, pos_label=0), "precision":
"precision", "f1": "f1", "roc_auc": "roc_auc", }
        :type scoring: dict, optional
```

```
        :param cv: Number of folds for the cross validation algorithm, defaults to
10
        :type cv: int, optional
        :param trials: Number of trials used to generate random models with
different hyperparameters, defaults to 25
        :type trials: int, optional
        :return: A dictionary with the scores and models trained using cross
validation
        :rtype: dict
        """
        if "accuracy" not in scoring:
            scoring["accuracy"] = "accuracy"

        X_train, _, t_train, _ = train_test_split(X, t, train_size=train_size)

        print(f"\n***Optimizing {tag} hyperparameters***")
        self.threads_dict[tag] = []
        for _ in range(trials):
            current_model = HpModels.random_model(tag)
            macro_model = HpModels.build_macro_model(current_model)
            model_thread = mth.ModelThread(
                X_train, np.ravel(t_train), macro_model, cv, scoring
            )
            model_thread.start()
            self.threads_dict[tag].append(model_thread)

        last_accuracy = 0
        print(f"\n***Cross validation results for {tag}***")
        for thread in self.threads_dict[tag]:
            thread.join()
            try:
                mean_accuracy = np.mean(thread.scores["test_accuracy"])
                if mean_accuracy > last_accuracy:
                    self.best_models[tag] = (thread.scores, thread.model)
                    last_accuracy = mean_accuracy
                    print("\nBest accuracy so far: ", last_accuracy)
            except:
                raise Exception(
                    f"No solution valid for {tag}. Increase number of trials or
kfolds."
                )
        try:
            print(
                "\nScore:",
                round(np.mean(last_accuracy), 4),
                "-",
                self.best_models[tag][1].steps[1][1],
            )
        except:
            raise Exception(
                f"No solution valid for {tag}. Increase number of trials or
kfolds."
            )
```

Análogamente para la red de neuronas profunda de Keras, se crea una versión de la optimización específica para multithreading en la que se utilizan tantos hilos como intentos y kfolds se definan, en otras palabras, para cada versión de la red de neuronas, con sus correspondientes hiperparámetros, se ejecutan en paralelo las kfolds de la validación cruzada.

```
def perform_optimize_DNN_multithread(
        self,
        X: np.ndarray,
        t: np.ndarray,
        kfolds: int = 10,
        train_size: float = 0.8,
        trials: int = 5,
        epochs: int = 50,
        batch_size: int = 40,
        metrics: tuple = (
            "accuracy",
            "Recall",
            cm.specificity,
            "Precision",
            cm.f1_score,
            "AUC",
        ),
    ) -> tuple:
        """Train the current model using cross validation and multithreading and
register its score comparing with new ones in each trial.

        :param X: Characteristic matrix numpy array of the dataset which will be
evaluated
        :type X: np.ndarray
        :param t: Vector labels numpy array of the dataset which will be evaluated
        :type t: np.ndarray
        :param kfolds: Number of folds for the cross validation algorithm,
defaults to 10
        :type kfolds: int, optional
        :param train_size: % of the data to be splitted into train and test
values, defaults to 0.8
        :type train_size: float, optional
        :param trials: Number of trials used to generate random neural networks
with different hyperparameters, defaults to 5
        :type trials: int, optional
        :param epochs: Number of maximum iterations allow to converge the
algorithm, defaults to 50
        :type epochs: int, optional
        :param batch_size: Size of the batch used to calculate lossing function,
defaults to 40
        :type batch_size: int, optional
        :param metrics: Tuple with the metrics to compare and evaluate the
differene neural networks, defaults to
("accuracy","Recall",cm.specificity,"Precision",cm.f1_score,"AUC")
        :type metrics: tuple, optional
        :return: A tuple containing a tuple with the model and the history of the
```

```
training, and another tuple with the X_test and t_test arrays to evaluate the
model
        :rtype: tuple
        """
        # Its needed the accuracy metric, if it is not passed it will be auto-
included
        if "accuracy" not in metrics:
            metrics.append("accuracy")

        _, m = X.shape
        n_classes = len(np.unique(t))
        cv = KFold(kfolds)
        last_mean = 0

        # Split the data into train and test sets. Note that test set will be
reserved for evaluate the best model later with data unseen previously for it
        X_train_set, X_test, t_train_set, t_test = train_test_split(
            X, t, train_size=train_size
        )

        # Loop for different trials or models to train in order to find the best
        threads_dict = {}
        for row in range(trials):

            model_aux, optimizer, loss = HpDNN.create_random_network(
                m, n_classes, metrics=metrics
            )
            params, comp = HpDNN.get_hyperparams(model_aux)

            print(f"\n***Trial {row+1} hyperparameters***", end="\n\n")
            print(params, "\n", comp)
            # Lists to store threads to manage them
            threads_dict[row] = []
            # Loop that manage cross validation using training set
            for train_index, test_index in cv.split(X_train_set):
                # This sentence carefully clones the untrained model in each fold
in order to avoid unwanted learning weights between them
                model = keras.models.clone_model(model_aux)
                model.compile(optimizer=optimizer, loss=loss, metrics=metrics)

                X_train, X_val = (
                    X_train_set[train_index],
                    X_train_set[test_index],
                )
                t_train, t_val = (
                    t_train_set[train_index],
                    t_train_set[test_index],
                )

                t_train = to_categorical(t_train, num_classes=n_classes)
                t_val = to_categorical(t_val, num_classes=n_classes)

                # Training of the model using multithreading
                thread = mth.DnnThread(
```

```
                    X_train, t_train, X_val, t_val, model, epochs, batch_size
                )
                thread.start()
                threads_dict[row].append(thread)

        # Save the train and test results,of the current model
        for row in range(trials):
            # Lists to store historical data and means per fold
            historial = []
            mean_folds = []
            print(f"\n***Trial {row+1} results***", end="\n\n")
            for thread in threads_dict[row]:
                thread.join()
                historial.append(thread.history.history)
                mean = np.mean(thread.history.history["val_accuracy"])
                mean_folds.append(mean)
                print(
                    f"\nKFold{threads_dict[row].index(thread)+1} --- Current
score: {mean}"
                )

            # Criterion to choose the best model with the highest accuracy score
in validation
            means = np.mean(mean_folds)
            history_metrics = HpDNN.split_history_metrics(historial)
            print(
                f"\n\tMin. Train score:
{min(np.concatenate(history_metrics['accuracy']))} | Max. Train score:
{max(np.concatenate(history_metrics['accuracy']))}"
            )
            if means > last_mean:
                # Save the model and its historial results
                self.best_model = (thread.model, history_metrics)
                self.test_set = (X_test, t_test)
                last_mean = means
```

Cabe recordar que las versiones secuenciales de las funciones que contienen el código fuente que definen las intrucciones de los procedimientos de optimización tanto para los modelos Scikit-Learn como para la red neuronal de Keras siguen disponibles tras la ampliación de esta parte del programa conservando la estructura principal de la primera parte del trabajo:

- Startup Succes Prediction

A excepción de la implementación en clases, que es de utilidad para la medición de los tiempo de ejecución tal y como se describe en el siguiente apartado.

3. Medición de tiempos

Se lleva a cabo una ampliación que permite realizar mediciones de tiempos de entrenamiento medios de los modelos de aprendizaje desarrollados para la clasificación supervisada. Para ello se utiliza una función envoltorio que decora aquellas funciones en las que resulta de interés medir el tiempo de ejecución, de forma que una vez decoradas, con solo llamarlas ya se calcularía el tiempo de ejecución.

```
import logging
import time

from functools import wraps

logger = logging.getLogger(__name__)


# Misc logger setup so a debug log statement gets printed on stdout.
logger.setLevel("DEBUG")
handler = logging.StreamHandler()
log_format = "%(asctime)s %(levelname)s -- %(message)s"
formatter = logging.Formatter(log_format)
handler.setFormatter(formatter)
logger.addHandler(handler)
```

Como se venía diciendo al final del apartado anterior, se ha optado por reestructurar los módulos de optimización de los modelos en clases con el fin de hacer uso de los atributos de instacia para referenciar los resultados con los mejores modelos y facilitando así la devolución de las mediciones de los tiempos de ejecución utilizando la función decoradora.

```
def timed(func):
        """This decorator prints the execution time for the decorated function."""

        @wraps(func)
        def wrapper(*args, **kwargs):
            start = time.time()
            result = func(*args, **kwargs)
            end = time.time()
            logger.debug(
                "{} ran in {}s".format(func.__name__, round(end - start, 2))
            )
            return round(end - start, 2)

        return wrapper
```

De la misma forma, para poder medir los tiempos de cada modelo por separado se generaliza la función de optimización original empleada para los modelos de Scikit-Learn, que se encarga de realizar el proceso de optimización y se define una nueva función dedicada a la llamada y recogida de los tiempos de ejecución de la anterior para cada modelo. Esta modificación no es necesaria para la red de neuronas puesto que es exclusiva el módulo específico.

Se decoran las funciones de optimización tanto secuencial como multihilo utlizando la función envoltorio anterior.

```
@timed
    def perform_optimizing_model
```

```
@timed
    def perform_optimizing_models_multithread
```

Y se llaman a las funciones de nivel superior que gestionan la medición de los tiempos.

```
def optimizing_models(self, models: list, *args, **kwargs) -> dict:

        time_models = dict()
        for tag in models:
            time_models[tag] = self.perform_optimizing_model(
                tag, *args, **kwargs
            )
        return self.best_models, pd.DataFrame(
            time_models, index=["Time Models"]
        )
```

```
def optimizing_models_multithread(self, models: list, *args, **kwargs):

        time_models = dict()
        for tag in models:
            time_models[tag] = self.perform_optimizing_models_multithread(
                tag, *args, **kwargs
            )

        return self.best_models, pd.DataFrame(
            time_models, index=["Time Models"]
        )
```

Para la red de neuronas el modo de operación sería el mismo. Decorar las funciones de optimización con la función envoltorio de la medición de tiempos:

```
@timed
    def perform_optimizing_DNN(
```

```
@timed
    def perform_optimize_DNN_multithread
```

Y ejecutar las funciones que llaman a la optimización y guardan el tiempo de ejecución:

```
def optimize_DNN(self, *args, **kwargs):
        timeDNN_seq = self.perform_optimizing_DNN(*args, **kwargs)
        return (self.best_model, self.test_set), timeDNN_seq
```

```
def optimize_DNN_multithread(self, *args, **kwargs):
        timeDNN_thr = self.perform_optimize_DNN_multithread(*args, **kwargs)
        return (self.best_model, self.test_set), timeDNN_thr
```

4. Resultados obtenidos

Todos los resultados extraídos de los métodos explicados anteriormente se recogen en un report de manera automática.

5. Conclusiones

De acuerdo con las novedades incluidas en el presente documento, extensible de la primera parte:

- Startup Success Prediction

Se pueden añadir un grupo de conclusiones.

En cuanto a las técnicas de reducción de dimensionalidad, se puede estudiar el grado de influencia entre las variables del conjunto de datos mediante la matriz de correlación y analizar los componentes principales que explican la mayor varianza de los mismos desarrollando una PCA.

Tanto las técnicas de agrupamiento por particiones como por densidad, explicadas anteriormente: K-means y DBSCAN, permiten tratar el estudio como un modelo de aprendizaje no supervisado donde no se utilizan etiquetas y el objetivo consiste en encontrar agrupaciones óptimas para los datos. Para determinar el número de clusters necesarios, el K-means se basa en el conocido como método elbow para calcular las inercias o distancias al cuadrado de los datos con sus respectivos centroides, indentificando el número para el cual se produce la mayor disminución consecutiva. Análogamente, el DBSCAN utiliza el algoritmo de K-Nearest Neighbours para extraer el valor del hiperparámetro epsilon del modelo, que hace referencia a la densidad de punto empleada.

Por lo que a las funciones declaradas en la la clase dedicada a la detección de anomalías respecta, merece la pena destacar la necesidad de adaptar el conjunto de datos para poder tratarlo como un problema de este tipo. Esto pasa por escoger una clase mayoritaria como normal, en este caso que las empresas tengan éxito, aprovechando que se tienen más muestras globales con esta etiqueta, y otra minoritaria que se considera la clase anómala y hace referencia al fracaso de las startup. Se trata de un modelo de aprendizaje semisupervisado, ya que se utliza una partición de los datos para entrenamiento que no contiene muestras anómalas y una para test que presenta una pequeña porción de las mismas. Dentro de este apartado se definen una serie de funciones que proponen la operación de algunos de los principales métodos empleados en este tipo de problemas: Isolation Forest, basado en los algoritmos utilizados en los bosques aleatorios para aislar las muestras con características diferentes al resto; LOF, utiliza la densidad de los datos para identificar las muestras atípicas comparándola con la de sus vecinos y una red de neuronas autoencoder, que durante el proceso de entrenamiento extrae las características esenciales de los datos, como consecuecia de la definición de la capa oculta de menor dimensión, y las utiliza para reconstruirlos a la salida, de esta manera las muestras anómalas presentes en el conjunto de test producirán un error de rescontrucción alto pudiendo ser identificadas en base a un umbral.

Si se analizan las partes del programa multihilo en comparativa con sus versiones secuenciales, se puede concluir una aceleración del proceso, como es el caso de la ejecución paralela en varios cores de los k experimentos de la validación cruzada para la red neuronal, o el proceso de optimización por modelo, teniendo en cuenta el número de intentos para cada uno. Esta ampliación favorece notablemente los aspectos del programa que abordan tareas recursivas, mejorando la organización de la ejecución de las instrucciones, que es donde se implentan. Este análisis tiene en cuenta los hilos virtuales configurados en el el código fuente, sin embargo y como cabría esperar, los resultados pueden diferir de una CPU a otra según el hardware.

Para terminar con el apartado de conlusiones, no se puede dejar de lado el interés y la importancia que presenta el hecho de realizar mediciones de tiempos de ejecución, especialmente allí donde el proceso demanda más recursos a la CPU, como es el caso del proceso de optimizización donde se desarrollan varios

entrenamientos utilizando validación cruzada. Esta mejora, proporciona mayor trazabilidad al sistema y explica una comparativa con los modelos más dificilmente escalables de cara a encontrar una solución representativa al problema de optimización, pues el tiempo de ejecución sería superior.