Lab 3 - Explainable and Trustworthy AI

Teaching Assistant: Eleonora Poeta (eleonora.poeta@polito.it)

Lab 3: Local post-hoc explainable models on structured data

LIME

LIME is a **local surrogate model**. It tests **what happens to the predictions** when you **give variations of your data** into the machine learning model.

The main steps are:

- LIME generates **a new dataset** consisting of **perturbed samples** and the corresponding **predictions** of the black box model.
- On the new dataset → LIME trains an **interpretable model** (weighted by the proximity of the sampled instances to the instance of interest).
- The learned model should be a **good approximation** of the **machine learning model** predictions **locally**, but it does not have to be a good global approximation.

Exercise 1:

The **Titanic** dataset describes the survival status of individual passengers on the Titanic. In this exercise you have to:

- **Preprocess** the Titanic dataset. Please, follow these main steps:

> - **Load** the dataset
> - **Split** the dataset into training and test set using the **80/20** ratio. **Shuffle** the dataset and **stratify** it using the target variable.
> - Fill **null** values. `age` column with the mean, `fare` with the median and `embarked` with the most frequent values.
> - **Remove** columns that are *not informative for the final task*, or that *contain information about target variable*.
> - **Encoding**: in this exercise, the encoding of the dataset ***will be different from previous exercises of the past labs.***

> - Follow the **step-by-step procedure** that is written in the Exercise.

- Fit the **RandomForestClassifier()** with `n_estimators=500`
  - Calculate the predictions with `.predict()`
  - Calculate the `accuracy_score()`

Solution:

Imports

```
# Import the required libraries for this exercise

from sklearn.datasets import fetch_openml, make_classification
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, MinMaxScaler
from sklearn.impute import SimpleImputer
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier
import xgboost
from sklearn.metrics import accuracy_score
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

Data Preprocessing - Until Encoding part

Load the dataset

```
# Load input features and target variable
df, y = fetch_openml("titanic", version=1, as_frame=True, parser='auto', return_X

# The "survived" column contains the target variable
df["survived"] = y
```

Split the dataset - 80/20 train/test ratio.

```
# Split the dataset. 80% for training data and 20% for test data. Shuffle the dat

df_train, df_test = train_test_split(df, test_size=0.2, shuffle=True, random_stat
```

Fill Null Values - `age` column

```
print(f'Number of null values in Train before pre-processing: {df_train.age.isnul
print(f'Number of null values in Test before pre-processing: {df_test.age.isnull(


df_train['age'] = df_train['age'].fillna(df_train['age'].mean())
df_test['age'] = df_test['age'].fillna(df_train['age'].mean())

print(f'Number of null values in Train after pre-processing: {df_train.age.isnull
print(f'Number of null values in Test after pre-processing: {df_test.age.isnull()
```

```
Number of null values in Train before pre-processing: 209/1047
Number of null values in Test before pre-processing: 54/262
Number of null values in Train after pre-processing: 0/1047
Number of null values in Test after pre-processing: 0/262
```

Fill Null Values - `fare` column

```
print(f'Number of null values in Train before pre-processing: {df_train.fare.isnu
print(f'Number of null values in Test before pre-processing: {df_test.fare.isnull


df_train['fare'] = df_train['fare'].fillna(df_train['fare'].median())
df_test['fare'] = df_test['fare'].fillna(df_train['fare'].median())


print(f'Number of null values in Train after pre-processing: {df_train.fare.isnul
print(f'Number of null values in Test after pre-processing: {df_test.fare.isnull(
```

```
Number of null values in Train before pre-processing: 1/1047
Number of null values in Test before pre-processing: 0/262
Number of null values in Train after pre-processing: 0/1047
Number of null values in Test after pre-processing: 0/262
```

Fill Null Values - `embarked` column

```
print(f'Number of null values in Train before pre-processing: {df_train.embarked.
print(f'Number of null values in Test before pre-processing: {df_test.embarked.is


imp = SimpleImputer(missing_values=np.nan, strategy='most_frequent')

df_train[['embarked']] = imp.fit_transform(df_train[['embarked']])

df_test[['embarked']] = imp.transform(df_test[['embarked']])


print(f'Number of null values in Train after pre-processing: {df_train.embarked.i
print(f'Number of null values in Test after pre-processing: {df_test.embarked.isn
```

```
Number of null values in Train before pre-processing: 0/1047
Number of null values in Test before pre-processing: 2/262
Number of null values in Train after pre-processing: 0/1047
Number of null values in Test after pre-processing: 0/262
```

Drop useless columns - `name`, `ticket`

```
df_train = df_train.drop(columns=['name','ticket'])
df_test = df_test.drop(columns=['name','ticket'])

df_train.head()
```

	pclass	sex	age	sibsp	parch	fare	cabin	embarked	boat	body
999	3	female	29.604316	0	0	7.7500	NaN	Q	15 16	NaN
392	2	female	24.000000	1	0	27.7208	NaN	C	12	NaN
628	3	female	11.000000	4	2	31.2750	NaN	S	NaN	NaN

Drop columns that contains info of the target classe (survived) - `cabin` , `body` , `boat` , `home.dest` .

```
df_train = df_train.drop(columns=['cabin', 'body', 'boat', 'home.dest'])

df_test = df_test.drop(columns=['cabin', 'body', 'boat', 'home.dest'])


df_train.head(2)
```

	pclass	sex	age	sibsp	parch	fare	embarked	survived
999	3	female	29.604316	0	0	7.7500	Q	1
392	2	female	24.000000	1	0	27.7208	C	1

Extract target variable and input features for the training and test data

```
y_train = df_train['survived']               # Target variable trainig set
X_train = df_train.drop('survived', axis=1)  # Features training set


y_test = df_test['survived']                 # Target variable test set
X_test = df_test.drop('survived', axis=1)    # Features test set
```

Encoding

Our **LIME explainer** (and most classifiers) takes in **numerical data**, **even if the features are categorical**.

- We thus **transform all of the string attributes into integers**, using sklearn's **LabelEncoder**.
- We *use a dictionary to save the correspondence between the integer values and the original strings* so we can present this later in the explanations.

1. **Identify** the **categorical columns** in the dataset and save them into a list.
    - They are the same for training and test data.
    - In this case, both `category` and `object` dtype represent categorical columns.

```
X_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1047 entries, 999 to 668
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   pclass    1047 non-null   int64
 1   sex       1047 non-null   category
 2   age       1047 non-null   float64
 3   sibsp     1047 non-null   int64
 4   parch     1047 non-null   int64
 5   fare      1047 non-null   float64
 6   embarked  1047 non-null   object
dtypes: category(1), float64(2), int64(3), object(1)
memory usage: 58.4+ KB
```

```
X_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 262 entries, 1028 to 203
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   pclass    262 non-null    int64
 1   sex       262 non-null    category
 2   age       262 non-null    float64
 3   sibsp     262 non-null    int64
 4   parch     262 non-null    int64
 5   fare      262 non-null    float64
 6   embarked  262 non-null    object
dtypes: category(1), float64(2), int64(3), object(1)
memory usage: 14.7+ KB
```

```
# Identify categorical columns in train dataset --- they are the same for test da
# You have to indicate the index of the categorical columns
categorical_cols = [0, 1, 6]
print(categorical_cols)
```

```
[0, 1, 6]
```

2. Create a dictionary of categorical_names. `categorical_names = {}`
3. Create a dictionary of the LabelEncoders. `le_dict = {}`
4. For each categorical feature, you have to:
    - Instanciate the **LabelEncoder()** from sklearn. `le = LabelEncoder()`
    - Fit the **LabelEncoder()** over the categorical feature of interest.
    - Transform the the categorical feature of interest.
    - Keep trace of the transformation done as follows: `categorical_names[feature] = le.classes_`
    - Save the label encoder in the dictionary above as follows: `le_dict[feature] = le`

> Do this procedure **only for the train set**. Then, **for the test set**, you will **apply** only `.transform()` .Rember to use the right label encoder for the right categorical feature that you just saved in the `le_dict` .

```
categorical_names = {}
le_dict = {}
for feature in categorical_cols:
    le = LabelEncoder()
    le.fit(X_train.iloc[:, feature])
    X_train.iloc[:, feature] = le.transform(X_train.iloc[:, feature])
    categorical_names[feature] = le.classes_
    le_dict[feature] = le
```

```
categorical_names
```

```
{0: array([1, 2, 3]),
 1: array(['female', 'male'], dtype=object),
 6: array(['C', 'Q', 'S'], dtype=object)}
```

```
categorical_names_test = {}
for feature in categorical_cols:
    le = le_dict[feature]
    X_test.iloc[:, feature] = le.transform(X_test.iloc[:, feature])
    categorical_names_test[feature] = le.classes_
```

```
categorical_names_test
```

```
{0: array([1, 2, 3]),
 1: array(['female', 'male'], dtype=object),
```

```
    6: array(['C', 'Q', 'S'], dtype=object)}
```

Now, **use a One-hot encoder**, so that our **classifier does not take the categorical features as continuous features.**

> ***We will use this encoder only for the classifier***, *not for the explainer* - and the reason is that the **explainer must make sure that a categorical feature only has one value.**

1. Instanciate the **OneHotEncoder()** to encode the categorical variables.
2. Apply the **MinMaxScaler()** to the numerical features.
3. Use the **ColumnTransformer()**.

```
# Identify the numerical columns - you must save the index of the column!
numerical_columns = numeric_features = [0, 2, 6]
print(numerical_columns)
```

```
[0, 2, 6]
```

```
# Initialize OneHotEncoder
onehot_encoder = OneHotEncoder(handle_unknown="ignore")

# Initialize MinMaxScaler
minmax_s = MinMaxScaler()


# Create ColumnTransformer
ct = ColumnTransformer(
    transformers=[
        ('onehot', onehot_encoder, categorical_cols),
        ('num', minmax_s, numerical_columns)
    ],
    remainder='passthrough'
)

# Apply ColumnTransformer to your train data
encoded_X_train = ct.fit_transform(X_train)

# Apply ColumnTransformer to your test data
encoded_X_test = ct.transform(X_test)
```

Fit the RandomForestClassifier with `n_estimators=500`

```
rf = RandomForestClassifier(n_estimators=500)
rf.fit(encoded_X_train, y_train)
```

```
▾ RandomForestClassifier
RandomForestClassifier(n_estimators=500)
```

Calculate the y_pred with the `.predict()` function from sklearn

```
y_pred = rf.predict(encoded_X_test)
```

Calculate the Accuracy Score

```
accuracy_score(y_test, y_pred)
```

```
0.7900763358778626
```

Exercise 1b:

Let's now explain the predictions obtained in the Exercise 1a using **LIME**. Before starting the exercise you have to:

- Install the lime library running the following command in a cell `!pip install lime`
- Import the module for tabular data as: `from lime import lime_tabular`

Then, the goal of this exercise is to explain an individual prediction of interest. To get you started in understanding how the library works, this part of the exercise will be mostly guided. You have to:

- Fix the random seed.
- Instanciate the explainer as: `explainer = lime_tabular.LimeTabularExplainer`.
  - Read the [documentation] and try to understand the role of each parameter.
  - In this case, the prediction function `pred_fn` has to be custom. *Follow the guide in the notebook.*
  - Now, try to explain the `instance i=0` with `explainer.explain_instance`. *What can you infer? What is the predicted class for that instance?*

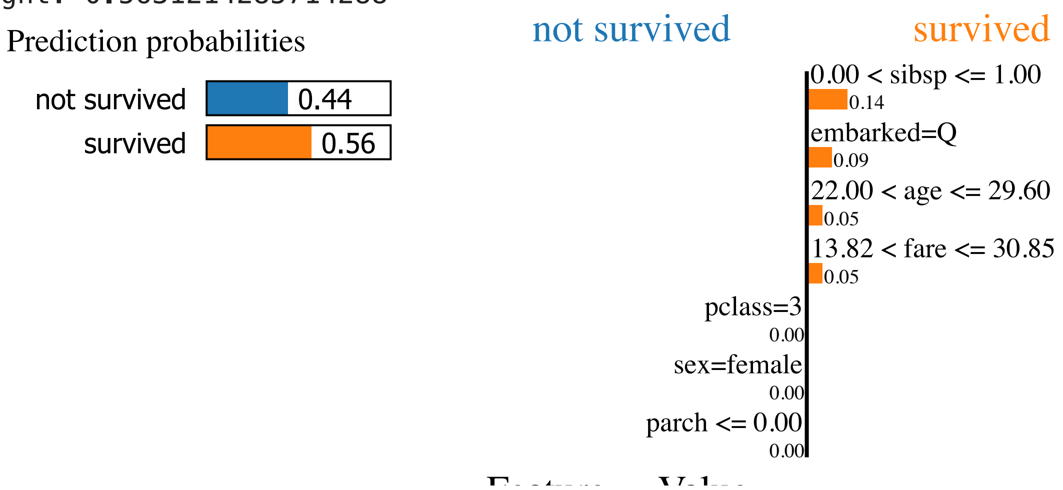
```
!pip install lime
from lime import lime_tabular
```

Explaining predictions

Fix the random seed with `np.random.seed(42)`

```
np.random.seed(42)


explainer = lime_tabular.LimeTabularExplainer(X_train.values,
                                              mode = 'classification',
                                              class_names=['not survived' , 'surv
                                              feature_names = X_train.columns,
                                              categorical_features=categorical_co
                                              categorical_names=categorical_names
                                              kernel_width=3,
                                              verbose=True)



def predict_fn(x):
  temporary_df = pd.DataFrame(x, columns=X_train.columns, dtype='object')
  print(temporary_df.head(2))
  transf = ct.transform(temporary_df)
  pred = rf.predict_proba(transf).astype(float)
  return pred


i = 1
exp = explainer.explain_instance(X_test.values[i],
                                 predict_fn,
                                 num_samples=3)
exp.show_in_notebook()
```

```
   pclass  sex        age sibsp parch       fare embarked
0     2.0  0.0  29.604316   1.0   0.0      24.15      1.0
1     2.0  0.0  40.003295   0.0   0.0  10.194764      1.0
Intercept 0.08663431052879247
Prediction_local [0.42233048]
Right: 0.5631214285714288
```



Exercise 1.c

It's time to play with LIME! 😀

The purpose of this exercise is to make you familiar with the LIME library and make you understand the main features.

- Instanciate **a new LimeTabularExplainer**
- Use the **same predict_fn** as before
- `explain_instance` for the instance `i=1`.
  - **Run** this for **5 times** and **pay attention** to the part about what features and to what extent they contributed to that prediction (explanation).
  - *Did you always obtain the same explanation?* If no, *what is the missing step?*
- Let's now change the parameter num_samples to `num_samples=15`.
  - Can you guess what is the role of this parameter?
- The parameter `num_features` indicates the maximum number of features present in explanation.
  - Try to vary this number between 1 and 6. Where can you see a change?
- Change the distance parameter to `distance_metric='l2'`.
  - Where is the distance used?

Setting the **random seed** is extremely important for LIME explainer. In fact, as you may have noticed, by running the LimeTabularExplainer cell several times, the **explanations** obtained, *for the same instance*, **change**.

```
np.random.seed(42)


explainer_new = lime_tabular.LimeTabularExplainer(X_train.values,
                                                   mode = 'classification',
                                                   class_names=['not survived' , 'surv
                                                   feature_names = X_train.columns,
                                                   categorical_features=categorical_co
                                                   categorical_names=categorical_names
                                                   kernel_width=15,
                                                   verbose=True)
```

```
def predict_fn(x):
  temporary_df = pd.DataFrame(x, columns=X_train.columns, dtype='object')
  print(temporary_df.head(2))
  transf = ct.transform(temporary_df)
  pred = rf.predict_proba(transf).astype(float)
  return pred
```

The `num_samples` parameter, as stated in the documentation, indicates the size of the neighborhood to learn the linear model.

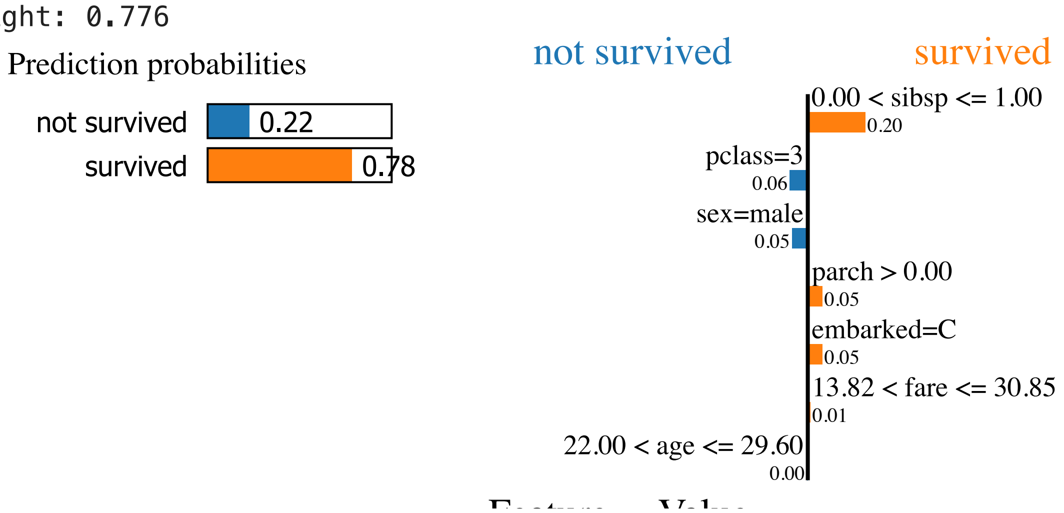
A **higher** `num_samples` value generally leads to a **more accurate approximation** of the model's behavior but *also increases computational cost.*

The `num_features` parameter specifies the maximum number of features that will be used in the explanation. LIME selects the most important features based on their influence on the model's predictions for the instance being explained.

This parameter allows you to **control the complexity** of the explanation by limiting the number of features considered.

```
i = 1
exp = explainer_new.explain_instance(X_test.values[i],
                                     predict_fn,
                                     num_samples=8, distance_metric='euclidean')
exp.show_in_notebook()
```

```
   pclass  sex        age  sibsp    parch       fare  embarked
0     2.0  1.0  29.604316    1.0      1.0    22.3583       0.0
1     2.0  0.0  28.094439    1.0  2.06693  10.512244       0.0
Intercept 0.508116161476772
Prediction_local [0.69869987]
Right: 0.776
```



LIME generates perturbed samples by randomly perturbing features within a specified range around the instance to be explained. The `distance_metric` determines how LIME measures the similarity between these perturbed samples and the original instance.