Methods for feature importance and feature interaction detection

Step 1: Feature importance detection

Here SubmodularPick in Lime model is used to save a representative sample of explanation objects for global interpretation.

Codes:

---

lime.lime_tabular.LimeTabularExplainer(*training_data, mode='classification', training_labels=None, feature_names=None, categorical_features=None, categorical_names=None, kernel_width=None, kernel=None, verbose=False, class_names=None, feature_selection='auto', discretize_continuous=True, discretizer='quartile', sample_around_instance=False, random_state=None, training_data_stats=None*)

Description:

Explains predictions on tabular data. For numerical features, perturb them by sampling from a Normal(0,1) and doing the inverse operation of mean-centering and scaling, according to the means and stds in the training data. For categorical features, perturb by sampling according to the training distribution, and making a binary feature that is 1 when the value is the same as the instance being explained.

Arguments:

training_data	numpy 2d array
mode	“classification” or “regression”
training_labels	labels for training data. Not required, but may be used by discretizer.
feature_names	list of names (strings) corresponding to the columns in the training data.
categorical_features	list of indices (ints) corresponding to the categorical columns. Everything else will be considered continuous. Values in these columns MUST be integers.
categorical_names	map from int to list of names, where categorical_names[x][y] represents the name of the yth value of column x.
kernel_width	kernel width for the exponential kernel. If None, defaults to sqrt (number of columns) * 0.75
kernel	similarity kernel that takes euclidean distances and kernel width as input and outputs weights in (0,1). If None, defaults to an exponential kernel.
verbose	if true, print local prediction values from linear model
class_names	list of class names, ordered according to whatever the classifier is using. If not present, class names will be ‘0’, ‘1’, ...
feature_selection	feature selection method. can be ‘forward_selection’, ‘highest_weights’, ‘lasso_path’, ‘none’ or ‘auto’. See function ‘explain_instance_with_data’ in lime_base.py for details on what each of the options does.
discretize_continuous	if True, all non-categorical features will be discretized into quartiles.

Discretizer	only matters if discretize_continuous is True. Options are ‘quartile’, ‘decile’ or ‘entropy’
sample_around_instance	if True, will sample continuous features in perturbed samples from a normal centered at the instance being explained. Otherwise, the normal is centered on the mean of the feature data.
random_state	an integer or numpy.RandomState that will be used to generate random numbers. If None, the random state will be initialized using the internal numpy seed.
training_data_stats	a dict object having the details of training data statistics. If None, training data information will be used, only matters if discretize_continuous is True. Must have the following keys: means”, “mins”, “maxs”, “stds”, “feature_values”, “feature_frequencies”

Usage recommendations:

Most parameters can follow the defaults. ‘training_data’, ‘feature_names’, ‘class_names’ should be initialized. ‘random_state’ also had better to be fixed for stability.

e.g. explainer = lime.lime_tabular.LimeTabularExplainer(test_data, feature_names=feature_names, class_names=class_names, mode = 'regression', random_state=20)

---

lime.submodular_pick.SubmodularPick(*explainer, data, predict_fn, method='sample', sample_size=1000, num_exps_desired=5, num_features=10, **kwargs*)

Description:

A representative sample of explanation objects using SP-LIME is saved, as well as saving all generated explanations. Firstly, a collection of candidate explanations is generated. From these candidates, num_exps_desired are chosen using submodular pick.

Arguments:

data	a numpy array where each row is a single input into predict_fn
predict_fn	prediction function. For classifiers, this should be a function that takes a numpy array and outputs prediction probabilities. For regressors, this takes a numpy array and returns the predictions. For ScikitClassifiers, this is classifier.predict_proba(). For ScikitRegressors, this is regressor.predict(). The prediction function needs to work on multiple feature vectors (the vectors randomly perturbed from the data_row).
method	The method to use to generate candidate explanations method == ‘sample’ will sample the data uniformly at random. The sample size is given by sample_size. Otherwise if method == ‘full’ then explanations will be generated for the entire data.
sample_size	The number of instances to explain if method == ‘sample’
num_exps_desired	The number of explanation objects returned
num_features	maximum number of features present in explanation

Returns:

sp_explanations	A list of explanation objects that has a high coverage explanation: All the candidate explanations saved for potential future use.

Usage recommendations:

'explainer' is the output of lime.lime_tabular.LimeTabularExplainer. If training data is not too large, method == 'full' is better. If data size is very large, part of the instances can be selected for training. For 'num_exps_desired', there is no special recommendation and no less than the default number is OK. For 'num_features', the best situation is that all the features or at least 95% of them are selected.

e.g.

sp = submodular_pick.SubmodularPick(explainer, data, predict_fn =model.predict, method = 'full', num_exps_desired=5, num_features=8)

#visualization

[exp.as_pyplot_figure(label=exp.available_labels()[0]) for exp in sp.sp_explanations]

---

feature_importance_table(*sbmodular, feature_names=None, threshold=0.1*)

Description:

To visualize the feature importance in a table form. *'threshold'* is to determine whether the feature is important. The table has 3 columns: 'features', 'weights', 'counts'. 'weights' shows the importance of each feature. 'counts' shows the number of features recognized as important ones among all picked data instances.

Arguments:

sbmodular	submodular pick in lime model
feature_names	the names of features
threshold	the threshold to pick up important features

Returns:

sorted_count	the table showing the feature importance in ascending order

Usage recommendations:

'threshold' should be set carefully. The recommendation is a value that is a little larger than the least important feature according to sp.sp_explanations.

e.g.

table = feature_importance_table(sp.sp_explanations, feature_names=feature_names, threshold=1)

Step 2: Feature interactions detection

Here NID or GradientNID algorithm is used to detect feature interactions.

---

application_NID(*inputs, targets, std_scale=False, detector="GradientNID", arch=[256, 128, 64], batch_size=100, device=torch.device("cpu"), weight_samples=False, add_linear=False, l1_const=None, grad_gpu=-1, seed=42, number=20, feature_names=None, decimal=4*)

Description:

To train the dataset with 'NID' and 'GradientNID' to get the feature interaction information.

Arguments:

inputs	numpy 2d array. It can be the whole dataset or part of them
targets	the target data corresponding to inputs
std_scale	standardizing the processed data by sklearn.preprocessing.StandardScaler()
detector	"NID" or "GradientNID"
arch	architecture of the neural networks, default is [256, 128,64]
batch_size	the batch size of the neural network, default is 100
device	the object of the device to which torch.Tensor is allocated, default is torch.device("cpu")
weight_samples	weight sampling with gaussian kernel, default is False
add_linear	adding the full connection layer, default is False
l1_const	the const parameter for l1 regularization, default is None
grad_gpu	if it is -1, the device is cpu, else the device is gpu. Default is -1
seed	the seed for random state, default is 42
number	the number of output interactions, default is 20
feature_names	the names of the features
decimal	the number of decimal places of the result, default is 4

Returns:

interactions	the feature interaction information
mlp_loss	the loss of NN

Usage recommendations:

Most settings can follow the defaults. ‘detector’ is recommended as ‘GradientNID’ because the high-order interactions are always the noises. Here ‘number’ is important and the recommendation is to set it as more than the number of features.

e.g.

interactions, mlp_loss = application_NID(inputs=inputs, targets=targets, feature_names=feature_names, detector="GradientNID", number=10)

Step 3: Combination and retraining

Here all explanations above are combined to create a new dataset and then it will be retrained to improve the performance.

---

combination(interactions, training_data, test_data, feature_names, delete=None, num=10)

Description:

To create new features with feature interactions by multiplication and combine them with the original dataset. If there exist some unimportant features, they should be deleted.

Arguments:

interactions	the feature interactions interpretation
training_data	the original training data
test_data	the original test data
feature_names	the names of the features

delete	the unimportant features to be deleted, default is None. If it is not None, for example, the 1st and 2nd features are deleted, delete=[1,2].
num	the number of selected features

Returns:

train_new	the new training data
test_new	the new test data
feature_names_new	the new feature names

Usage recommendations:

'delete' needs to be treated with caution. Only when the feature has much lower weight in lime explanations, it can be deleted. According to 'num', it had better follow the number of original features.

e.g.

train_new, test_new, feature_names_new = combination(interactions, train_data, test_data, feature_names, delete=[1], num=11)

---

retraining(table, model, training_data, training_target, test_data, test_target, task='regression', method='rmse and r2', multi_class='macro', rmse=None, r2=None, accuracy=None, f1=None, n=5):

Description:

After the new dataset is created, lime model will be utilized again to detect the feature importance.

Arguments:

table	the feture importance table
model	the model used before
training_data	the new training data
training_target	the targets of training data
test_data	the new test data
test_target	the targets of the teat data
task	'regression', 'binary_classification' or 'multi_classification'
method	'rmse and r2', 'rmse' or 'r2' in regression task; 'acc and f1', 'acc' or 'f1' in classification tasks
multi_class	'micro', 'macro', 'samples' or 'weighted', default is 'macro'
rmse	root mean squared error
r2	r2 score
accuracy	accuracy score
f1	f1 score
n	the number of candidate features to be deleted

Returns:

rmse_best	the optimized rmse score
r2_best	the optimized r2 score
acc_best	the optimized accuracy score
f1_best	the optimized f1 score
feature_deleted	the deleted unimportant features

Usage recommendations:

For ‘method’, select the suitable method according to the task. For ‘n’, the number of candidate features to be deleted should be less than half of the feature numbers.

e.g.

rmse_best, r2_best, feature_deleted = retraining(table, model, training_data, training_target, test_data, test_target, task='regression', method='rmse and r2', rmse=rmse, r2=r2, n=6)

Example:

Datasert; IKV_dataset - Young's Modulus

Codes:

```
train, test = train_test_split(dataset, train_size=0.8, random_state=20)

# extracting the training data, test data and the feature names

train_data = train.iloc[:, 1:8].values

train_target = train.iloc[:, 8:].values

test_data = test.iloc[:, 1:8].values

test_target = test.iloc[:, 8:].values

feature_names=list(dataset.columns)[1:8]

target_names = list(dataset.columns)[8:]


# random forest model to predict the young's modulus

model_1 = RandomForestRegressor(n_estimators=300, random_state=20)

model_1.fit(train_data, train_target[:,0])

Young_pred = model_1.predict(test_data)

# RMSE and R2 score

rmse = np.sqrt(mean_squared_error(Young_pred, test_target[:,0]))

r2 = r2_score(Young_pred, test_target[:,0])


# local explanation in LIME algorithm

explainer = lime.lime_tabular.LimeTabularExplainer(test_data,
feature_names=feature_names, class_names=target_names[0], verbose = False, mode =
'regression', random_state=20)

# submodular pick based on the local explainer above to create the global explainer

sp = submodular_pick.SubmodularPick(explainer, test_data, model_1.predict, method = 'full')

# visualizing the feature importance for each picked data instance
```

```
[exp.as_pyplot_figure() for exp in sp.sp_explanations]

a = feature_importance_table(sp.sp_explanations, feature_names=feature_names,
threshold=3)


interactions, mlp_loss = application_NID(dataset.iloc[:, 1:8].values, dataset.iloc[:, 8].values,
feature_names=feature_names, detector="GradientNID", number=20)


# create new datasets with feature interactions and delete unimportant features

train_new, test_new, feature_names_new = combination(interactions, train_data, test_data,
feature_names, num=11)

# reuse the lime model for the new dataset

model_12 = RandomForestRegressor(n_estimators=300, random_state=20)

model_12.fit(train_new,train_target[:,0])

explainer_12 = lime.lime_tabular.LimeTabularExplainer(test_new,
feature_names=feature_names_new, class_names=target_names[0], verbose = False, mode =
'regression', random_state=20)

# submodular pick for the new dataset

sp_new = submodular_pick.SubmodularPick(explainer_12, test_new, model_12.predict,
num_features=len(feature_names_new), method = 'full', num_exps_desired=10)

# visualizing

[exp.as_pyplot_figure() for exp in sp_new.sp_explanations]

a12 = feature_importance_table(sp_new.sp_explanations,
feature_names=feature_names_new, threshold=3)


#retraining to optimize both rmse and r2

rmse_best, r2_best, feature_deleted = retraining(table=a12, model=model_12,
training_data=train_new, training_target=train_target[:,0], test_data=test_new,
test_target=test_target[:,0], task='regression', method='rmse and r2', rmse=rmse, r2=r2, n=8)

#retraining to optimize only rmse

rmse_best, feature_deleted = retraining(table=a12, model=model_12,
training_data=train_new, training_target=train_target[:,0], test_data=test_new,
test_target=test_target[:,0], task='regression', method='rmse', rmse=rmse, r2=r2, n=8)

#retraining to optimize only r2

r2_best, feature_deleted = retraining(table=a12, model=model_12, training_data=train_new,
training_target=train_target[:,0], test_data=test_new, test_target=test_target[:,0],
task='regression', method='r2', rmse=rmse, r2=r2, n=8)
```