Notice for Machine Learning Coding in Python

Dear 5194 Scholar,

I'd like to update some important notes, which might help your project when you are using scikit-learn package in python. These notes are presented as follows:

- Split X and y in to training and testing data
- Pre-process X and y
- Train your models [problem: for multi-class classification]
  - About Classification Classifiers
  - About Cross-Validation

---

1. Split X and y in to training and testing data

Both regression and classification codes posted on Carmen have included how to split X and y into 70% training and 30% testing by using the following code:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                     test_size=0.3,
                                                     random_state=1)
```

It should work fine for most of your project data. However, some of your project data might have some columns in X that are categorical values with a huge number of different levels. For example, a column named "operating system" has the following values [a, a, b, b, b ,b, c, c, d, d, e, e, e, e, e, e, f , f, g, g, h, h]. Once you use the above code to split, the training set might only have [a, a, b, b, b ,b, c, c, d, d, e, e, e, e], the testing set might be [e, e, f , f, g, g, h, h]. Do you see what problem is here? That is, training data does not contain ALL possible levels for column `operating system`, and it ignore the "f", "g", and 'h" levels! If you train your model in this way, your model would NOT be able to make prediction when `operating system` = "f" or "g" or "h".

In order to fix it, there are generally two ways:
(1) The first option is to regroup your features with a huge number of different levels because it would result in high cardinality issue (i.e., too many unique values). **If you do have some features with too many levels, please regroup them before training models.**

(2) The second option is to split the data in a stratified fashion:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                   test_size=0.3,
                                                   random_state=1,
                                                   stratify=X['your column'])
```

We feed the column with many levels in to the argument `stratify`. It would split the X and y according to the proportion of samples of each level in 'your column', and would keep all possible levels.

2. Pre-process X and y

Both regression and classification codes posted on Carmen have included how to pre-process X and y. Yet, here are some notes that you need to pay attention when you do similar steps on your project data.

- There is a lot of functions in scikit-learn to perform pre-processing by using transformation functions. Note that for **Classification problem** some functions are only allowed to be applied X and some functions are only allowed to be applied on y.
  For example, `LabelEncoder` and `LabelBinarizer` are **NOT** intended to be used on X, and they are aimed to perform transformation on y (supervised learning target).

  If you want to know more transformation functions,
  please see here: https://scikit-learn.org/stable/modules/preprocessing.html,
  and here https://scikit-learn.org/stable/modules/preprocessing_targets.html

- Some of your project data might need to apply several transformation functions on different columns, and also want to keep the rest of the columns as their original values. To achieve this goal, you have to use ` ColumnTransformer` to wrap several transformation functions and columns' names. Let me clarify it through the following examples:

  Recall that the dataset of Indian Liver Patient Records has one categorical feature `Gender`, and the rest of columns are numerical. Assume that I want to perform `OneHotEncoder` on this `Gender` column, and want to perform `StandardScaler` on both `Age` and `Total_Bilirubin` columns. For the rest of the columns, I want to keep their original values.

  Then, the code for this combination of pre-processing is presented as follows:

```
# define the transformation methods for the columns
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import  OneHotEncoder

t = [('ohe', OneHotEncoder(), ['Gender']),
     ('scale', StandardScaler(), ['Age', 'Total_Bilirubin'])
    ]


col_trans = ColumnTransformer(transformers=t,
                              remainder="passthrough")
# fit the transformation on training data
col_trans.fit(X_train)


# apply transformation to both training and testing data
X_train_transform = col_trans.transform(X_train)
X_test_transform = col_trans.transform(X_test)
```

  The meanings of the above code are:
  (1) we first define a `list` called `t` to specify the transformation functions to be applied to which columns in the data. There are two tuples in `t`. Each tuple has the shape of (name, transformation function, columns).

For example, `('ohe', OneHotEncoder(), ['Gender'])` means that this kind of transformation named as 'ohe' (you could use any name you like), then use `OneHotEncoder()` function applied on `Gender` column.
Similarly, `('scale', StandardScaler(), ['Age', 'Total_Bilirubin']` means that apply `StandardScaler()` function on both `Age` and `Total_Bilirubin` columns.

(2) Then, we feed the `t` into `ColumnTransformer` function. There are two important arguments, `transformers` and `remainder`. The `transformers` argument is assigned with `t`. The `remainder` argument is used to specify what we should do for the rest of columns that are not applied any transformation. The default setting of `remainder` is "drop", which means non-specified columns are dropped. However, we want to keep the rest of the columns as their original values. So, we should specify **remainder = "passthrough".**

(3) Next, we call `.fit` to fit those transformation rules on the training data. You can imagine that this step is kind of activating those transformation rules by following a specific data.

(4) Finally, we call `.transform` to apply those fitted transformation rules on those data that we want to make transformations, including the training data itself and the testing data.

3. Train your models

Some of you might have troubles when you follow the procedures presented in the codes posted on Carmen. I summarized some of problems that you might have and solutions to them.

3.1 About Classification Classifiers:

```
# sklearn classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
```

- `LogisticRegression(solver='lbfgs', random_state=123)`
  For Classification problem, if your target Y is "binary" (e.g., 0, 1), the above code about defining a logistic regression should work fine. However, if your target Y is "multi-class" (e.g., 0, 1, 2, 3...), the above code would raise an error since the default setting in `LogisticRegression` is for "binary". There are two ways about specifying multi-class setting in `LogisticRegression.`
  - `LogisticRegression(solver='lbfgs',random_state=123,multi_class='ovr')`
    The first version for specifying multi-class in called "One-vs-Rest" (ovr). For example, your Y value has four different classes and they are class1, class2, class3, class4. You can image this scheme is to fit four binary Logistic Regression models. Each model, for example, model 1 has the Y value = class1 and Not class1; model 2 has the Y value = class 2 and Not class2 ....
  - `LogisticRegression(solver='lbfgs',random_state=123,multi_class='multinomial')`
    The second version for specifying multi-class in called "Multi-nomial". This schema is in fact the generic version for Logistic Regression. If you know that the objective function of Binary Logistic Regression is to minimize the log likelihood, now the likelihood function could be

generalized for multi-class version. For more details, see here: https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

- About convergence. Some of you might get warnings to say that your solver does not converge if your project data is very large. You could increase the number of iterations to let the solver keep running by specifying *max_iter*. For example:

```
LogisticRegression(solver='lbfgs',random_state=123,multi_class='ovr',max_iter=
5000)
```

- For the rest of classifiers: `DecisionTreeClassifier, RandomForestClassifier, MLPClassifier`
  They support the multi-class classification in nature. So, you do not need to specify *multi_class* argument.
  There is one classifier that you might need to pay attention : `MLPClassifier`
  In Carmen's code, we are specifying
  `MLPClassifier(random_state=123, solver='lbfgs', max_iter=1000)`
  However, if your project data is very large, you might need to change `solver='lbfgs'` **to be** `solver='adam'` **and increase the number of iterations, for instance,** `max_iter=10000`

3.2 About Cross-Validation

```
from sklearn.model_selection import  cross_validate
```

- **Problem 1: train/validate folds**. there are generally two way of fold generations: KFold and StratifiedKFold.

```
from sklearn.model_selection import  StratifiedKFold, KFold
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=123)
cv = KFold(n_splits=5, shuffle=True, random_state=123)
```

  - "KFold" - divides all the samples in groups of samples, called folds of equal sizes (if possible).
  - "StratifiedKFold" - a variation of k-fold which returns stratified folds: each fold contains approximately the same percentage of samples of each target class as the complete set.

    In our Classification code, you could see that we are using "**StratifiedKFold**" function to split the training data and perform cross-validation. It requires that your **Y value is either binary** (e.g., 0, 1) or **multi-class** (e.g., 1, 2, 3, 4, ...)

    However, when you want to perform Regression models, generally, your **Y value is strictly continuous** such as 0.1, 0.005, 1.2, ... Therefore, you have to use "**KFold**" to do the cross-validation.

- **Problem 2: the metrics/scorings to evaluate Cross-Validation**

  You might have seen that for Binary Classification, the scorings are :

```
scoring = ['accuracy', 'precision', 'recall', 'f1', 'roc_auc']
```

```
# perform the 5-fold CV and get the metrics results
cv_results = cross_validate(estimator=clf,
                            X=X_train_transform,
                            y=y_train_transform,
                            scoring=scoring,
                            cv=cv,
                            return_train_score=False)
```

If your Y value is binary, these scorings would work fine in cross-validation. However, if your target Y is "multi-class" (e.g., 0, 1, 2, 3…), the above code would cause error when you perform cross-validation because some of the scorings are needed to be specified as a multi-class version! Using the following settings would solve your problem.

```
scoring = ['accuracy', 'precision_micro', 'recall_micro', 'f1_micro', 'roc_auc_ovr']
```

For more details, see here : https://scikit-learn.org/stable/modules/model_evaluation.html

- **Problem 3: the metrics/scorings to evaluate Testing data**

You might have seen that for Binary Classification, the evaluation code for testing data is as follows:

```
# ======== Step 2: Evaluate the model using testing data =======

# fit the Logistic Regression model
clf.fit(X=X_train_transform, y=y_train_transform)

# predition on testing data
y_pred_class = clf.predict(X=X_test_transform)
y_pred_score = clf.predict_proba(X=X_test_transform)[:, 1]

# AUC of ROC
auc_ontest = roc_auc_score(y_true=y_test_transform, y_score=y_pred_score)
# confusion matrix
cm_ontest = confusion_matrix(y_true=y_test_transform, y_pred=y_pred_class)
# precision score
precision_ontest = precision_score(y_true=y_test_transform, y_pred=y_pred_class)
# recall score
recall_ontest = recall_score(y_true=y_test_transform, y_pred=y_pred_class)
# classification report
cls_report_ontest = classification_report(y_true=y_test_transform,
                                          y_pred=y_pred_class)
```

The above code would raise errors when your Y value are multi-class. The reason is that the function used to calculate those metrics/scores are needed to specify the multi-class schema.

I listed code lines that have to be changed if your Y is multi-class.

-
  ```
  y_pred_score = clf.predict_proba(X=X_test_transform)[:, 1]
  ```

  This line is used to calculate the probabilities for each class in Y.
  If your Y value is binary (Y=0, 1), "[:, 1]" at the end of this line is used to extract the probabilities for Y=1.
  If your Y value is multi-class, you need to delete "[:, 1]".
  So, this line needed to be changed as

  ```
  y_pred_score = clf.predict_proba(X=X_test_transform)
  ```

-
  ```
  auc_ontest = roc_auc_score(y_true=y_test_transform, y_score=y_pred_score)
  ```

  This line is used to calculate the AUC value. If your Y is multi-class, you should specify two more arguments:

  ```
  auc_ontest = roc_auc_score(y_true=y_test_transform, y_score=y_pred_score,
                             average='micro', multi_class='ovr')
  ```

-
  ```
  precision_ontest = precision_score(y_true=y_test_transform, y_pred=y_pred_clas
  s)
  ```

  This line is used to calculate the precision value. If your Y is multi-class, you should specify one more argument:

  ```
  precision_ontest = precision_score(y_true=y_test_transform,
                                     y_pred=y_pred_class,
                                     average='micro')
  ```

-
  ```
  recall_ontest = recall_score(y_true=y_test_transform, y_pred=y_pred_class)
  ```

  This line is used to calculate the recall value. If your Y is multi-class, you should specify one more argument:

  ```
  recall_ontest = recall_score(y_true=y_test_transform, y_pred=y_pred_class,
                               average='micro')
  ```