11/2017
Datacamp - Machine Learning with the Experts: School Budgets (Data Scientist Track with Python)
[Machine Learning with the Experts: School Budgets]

Course Description

Data science isn't just for predicting ad-clicks-it's also useful for social impact! This course is a case study from a machine learning competition on DrivenData. You'll explore a problem related to school district budgeting. By building a model to automatically classify items in a school's budget, it makes it easier and faster for schools to compare their spending with other schools. In this course, you'll begin by building a baseline model that is a simple, first-pass approach. In particular, you'll do some natural language processing to prepare the budgets for modeling. Next, you'll have the opportunity to try your own techniques and see how they compare to participants from the competition. Finally, you'll see how the winner was able to combine a number of expert techniques to build the most accurate model.

Part 3 : Improving your model

Here, you'll improve on your benchmark model using pipelines. Because the budget consists of both text and numeric data, you'll learn to how build pipielines that process multiple types of data. You'll also explore how the flexibility of the pipeline workflow makes testing different approaches efficient, even in complicated problems like this one!

Instantiate pipeline

In order to make your life easier as you start to work with all of the data in your original DataFrame, df, it's time to turn to one of scikit-learn's most useful objects: the Pipeline.

For the next few exercises, you'll reacquaint yourself with pipelines and train a classifier on some synthetic (sample) data of multiple datatypes before using the same techniques on the main dataset.

The sample data is stored in the DataFrame, sample_df, which has three kinds of feature data: numeric, text, and numeric with missing values. It also has a label column with two classes, a and b.

In this exercise, your job is to instantiate a pipeline that trains using the numeric column of the sample data.

Instructions

- Import Pipeline from sklearn.pipeline.
- Create training and test sets using the numeric data only. Do this by specifying sample_df[['numeric']] in train_test_split().
- Instantiate a pipeline as pl by adding the classifier step. Use a name of 'clf' and the same classifier from Chapter 2: OneVsRestClassifier(LogisticRegression()).
- Fit your pipeline to the training data and compute its accuracy to see it in action! Since this is toy data, you'll use the default scoring method for now. In the next chapter, you'll return to log loss scoring.

```
# Import Pipeline
from sklearn.pipeline import Pipeline

# Import other necessary modules
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier

# Split and select numeric data only, no nans
X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric']],
                                                    pd.get_dummies(sample_df['label']),
                                                    random_state=22)

# Instantiate Pipeline object: pl
pl = Pipeline([
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])

# Fit the pipeline to the training data
pl.fit(X_train, y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on sample data - numeric, no nans: ", accuracy)
```

Results :

```
     numeric     text  with_missing label
0 -10.856306               4.433240     b
1   9.973454      foo      4.310229     b
2   2.829785  foo bar      2.469828     a
3 -15.062947               2.852981     b
4  -5.786003  foo bar      1.826475     a

<script.py> output:

    Accuracy on sample data - numeric, no nans:  0.62
```

Perfect. Now it’s time to incorperate numeric data with missing values by adding a preprocessing step!

Preprocessing numeric features

What would have happened if you had included the with ‘with_missing’ column in the last exercise? Without imputing missing values, the pipeline would not be happy (try it and see). So, in this exercise you’ll improve your pipeline a bit by using the Imputer() imputation transformer from scikit-learn to fill in missing values in your sample data.

By default, the imputer transformer replaces NaNs with the mean value of the column. That’s a good enough imputation strategy for the sample data, so you won’t need to pass anything extra to the imputer.

After importing the transformer, you will edit the steps list used in the previous exercise by inserting a (name, transform) tuple. Recall that steps are processed sequentially, so make sure the new tuple encoding your preprocessing step is put in the right place.

The sample_df is in the workspace, in case you’d like to take another look. Make sure to select both numeric columns- in the previous exercise we couldn’t use with_missing because we had no preprocessing step!

Instructions

- Import Imputer from sklearn.preprocessing.
- Create training and test sets by selecting the correct subset of sample_df: ‘numeric’ and ‘with_missing’.

- Add the tuple (‘imp’, Imputer()) to the correct position in the pipeline. Pipeline processes steps sequentially, so the imputation step should come before the classifier step.
- Complete the .fit() and .score() methods to fit the pipeline to the data and compute the accuracy.

```
# Import the Imputer object
from sklearn.preprocessing import Imputer

# Create training and test sets using only numeric data
X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric', 'with_missing']],
                                                    pd.get_dummies(sample_df['label']),
                                                    random_state=456)


# Insantiate Pipeline object: pl
pl = Pipeline([
        ('imp', Imputer()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])

# Fit the pipeline to the training data
pl.fit(X_train,y_train)

# Compute and print accuracy
accuracy = pl.score(X_test,y_test)
print("\nAccuracy on sample data - all numeric, incl nans: ", accuracy)
```

Results :

```
<script.py> output:

    Accuracy on sample data - all numeric, incl nans:  0.636
```

Nice! Now you know how to use preprocessing in pipelines with numeric data, and it looks like the accuracy has improved because of it! Text data preprocessing is next!

Preprocessing text features

Here, you’ll perform a similar preprocessing pipeline step, only this time you’ll use the text column from the sample data.

To preprocess the text, you’ll turn to CountVectorizer() to generate a bag-of-words representation of the data, as in Chapter 2. Using the default arguments, add a (step, transform) tuple to the steps list in your pipeline.

Make sure you select only the text column for splitting your training and test sets.

As usual, your sample_df is ready and waiting in the workspace.

Instructions

- Import CountVectorizer from sklearn.feature_extraction.text.
- Create training and test sets by selecting the correct subset of sample_df: ‘text’.
- Add the ‘CountVectorizer’ step (with the name ‘vec’) to the correct position in the pipeline.
- Fit the pipeline to the training data and compute its accuracy.

```
# Import FunctionTransformer
from sklearn.preprocessing import FunctionTransformer

# Obtain the text data: get_text_data
get_text_data = FunctionTransformer(lambda x: x['text'], validate=False)

# Obtain the numeric data: get_numeric_data
get_numeric_data = FunctionTransformer(lambda x: x[['numeric', 'with_missing']], validate=False)

# Fit and transform the text data: just_text_data
just_text_data = get_text_data.fit_transform(sample_df)

# Fit and transform the numeric data: just_numeric_data
just_numeric_data = get_numeric_data.fit_transform(sample_df)

# Print head to check results
print('Text Data')
print(just_text_data.head())
print('\nNumeric Data')
print(just_numeric_data.head())
```

Results :

```
<script.py> output:
    Text Data
    0
    1        foo
    2    foo bar
    3
    4    foo bar
    Name: text, dtype: object

    Numeric Data
     numeric  with_missing
    0 -10.856306      4.433240
    1   9.973454      4.310229
    2   2.829785      2.469828
    3 -15.062947      2.852981
    4  -5.786003      1.826475
```

Nice. You can see in the shell that fit and transform are now available to the selectors. Let's put the selectors to work!

Multiple types of processing: FeatureUnion

Now that you can separate text and numeric data in your pipeline, you're ready to perform separate steps on each by nesting pipelines and using FeatureUnion().

These tools will allow you to streamline all preprocessing steps for your model, even when multiple datatypes are involved. Here, for example, you don't want to impute our text data, and you don't want to create a bag-of-words with our numeric data. Instead, you want to deal with these separately and then join the results together using FeatureUnion().

In the end, you'll still have only two high-level steps in your pipeline: preprocessing and model instantiation. The difference is that the first preprocessing step actually consists of a pipeline for numeric data and a pipeline for text data. The results of those pipelines are joined using FeatureUnion().

Instructions

- In the process_and_join_features:
  - Add the steps (‘selector’, get_numeric_data) and (‘imputer’, Imputer()) to the ‘numeric_features’ preprocessing step.
  - Add the equivalent steps for the text_features preprocessing step. That is, use get_text_data and a CountVectorizer step with the name ‘vectorizer.
- Add the transform step process_and_join_features to ‘union’ in the main pipeline, pl.
- Hit ‘Submit Answer’ to see the pipeline in action!

```
# Import FeatureUnion
from sklearn.pipeline import FeatureUnion

# Split using ALL data in sample_df
X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric', 'with_missing', 'text']
],
                                                    pd.get_dummies(sample_df['label']),
                                                    random_state=22)

# Create a FeatureUnion with nested pipeline: process_and_join_features
process_and_join_features = FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer())
                ]))
             ]
        )

# Instantiate nested pipeline: pl
pl = Pipeline([
        ('union', process_and_join_features),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])


# Fit pl to the training data
pl.fit(X_train, y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on sample data - all data: ", accuracy)
```

Results :

```
<script.py> output:

    Accuracy on sample data - all data:  0.928
```

Crushed it! You now know more about pipelines than many practicing data scientists. You're on fire!

Using FunctionTransformer on the main dataset

In this exercise you're going to use FunctionTransformer on the primary budget data, before instantiating a multiple-datatype pipeline in the next exercise.

Recall from Chapter 2 that you used a custom function combine_text_columns to select and properly format text data for tokenization; it is loaded into the workspace and ready to be put to work in a function transformer!

Concerning the numeric data, you can use NUMERIC_COLUMNS, preloaded as usual, to help design a subset-selecting lambda function.

You're all finished with sample data. The original df is back in the workspace, ready to use.

Instructions

- Complete the call to multilabel_train_test_split() by selecting df[NON_LABELS].
- Compute get_text_data by using FunctionTransformer() and passing in combine_text_columns. Be sure to also specify validate=False.
- Use FunctionTransformer() to compute get_numeric_data. In the lambda function, select out the NUMERIC_COLUMNS of x. Like you did when computing get_text_data, also specify validate=False.

```python
# Import FunctionTransformer
from sklearn.preprocessing import FunctionTransformer

# Get the dummy encoding of the labels
dummy_labels = pd.get_dummies(df[LABELS])

# Get the columns that are features in the original df
NON_LABELS = [c for c in df.columns if c not in LABELS]

# Split into training and test sets
X_train, X_test, y_train, y_test = multilabel_train_test_split(df[NON_LABELS],
                                                               dummy_labels,
                                                               0.2,
                                                               seed=123)

# Preprocess the text data: get_text_data
get_text_data = FunctionTransformer(combine_text_columns,validate=False)

# Preprocess the numeric data: get_numeric_data
get_numeric_data = FunctionTransformer(lambda x: x[NUMERIC_COLUMNS], validate=False)
```

Results :

At this point we're not even surprised - you rule! Now go forth and build a full pipeline!

Add a model to the pipeline

You're about to take everything you've learned so far and implement it in a Pipeline that works with the real, DrivenData budget line item data you've been exploring.

Surprise! The structure of the pipeline is exactly the same as earlier in this chapter:

- the preprocessing step uses FeatureUnion to join the results of nested pipelines that each rely on FunctionTransformer to select multiple datatypes
- the model step stores the model object

You can then call familiar methods like .fit() and .score() on the Pipeline object pl.

Instructions

- Complete the 'numeric_features' transform with the following steps:
  - get_numeric_data, with the name 'selector'.
  - Imputer(), with the name 'imputer'.
- Complete the 'text_features' transform with the following steps:
  - get_text_data, with the name 'selector'.
  - CountVectorizer(), with the name 'vectorizer'.
- Fit the pipeline to the training data.
- Hit 'Submit Answer' to compute the accuracy!

```
# Complete the pipeline: pl
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer())
                ]))
             ]
        )),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])

# Fit to the training data
pl.fit(X_train,y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on budget dataset: ", accuracy)
```

Results :

```
<script.py> output:

    Accuracy on budget dataset:  0.203846153846
```

Great work! Now that you've built the entire pipeline, you can easily start trying out different models by just modifying the 'clf' step.

Try a different class of model

Now you’re cruising. One of the great strengths of pipelines is how easy they make the process of testing different models.

Until now, you’ve been using the model step (‘clf’, OneVsRestClassifier(LogisticRegression())) in your pipeline.

But what if you want to try a different model? Do you need to build an entirely new pipeline? New nests? New FeatureUnions? Nope! You just have a simple one-line change, as you’ll see in this exercise.

In particular, you’ll swap out the logistic-regression model and replace it with a random forest classifier, which uses the statistics of an ensemble of decision trees to generate predictions.

Instructions

- Import the RandomForestClassifier from sklearn.ensemble.
- Add a RandomForestClassifier() step named ‘clf’ to the pipeline.
- Hit ‘Submit Answer’ to fit the pipeline to the training data and compute its accuracy.

```
# Import random forest classifer
from sklearn.ensemble import RandomForestClassifier

# Edit model step in pipeline
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer())
                ]))
             ]
        )),
        ('clf', RandomForestClassifier())
    ])

# Fit to the training data
pl.fit(X_train, y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on budget dataset: ", accuracy)
```

Results :

```
<script.py> output:

    Accuracy on budget dataset:  0.269230769231
```

An accuracy improvement- amazing! All your work building the pipeline is paying off. It’s now very simple to test different models!

Can you adjust the model or parameters to improve

accuracy?

You just saw a substantial improvement in accuracy by swapping out the model. Pipelines are amazing!

Can you make it better? Try changing the parameter n_estimators of RandomForestClassifier(), whose default value is 10, to 15.

Instructions

- Import the RandomForestClassifier from sklearn.ensemble.
- Add a RandomForestClassifier() step with n_estimators=15 to the pipeline with a name of ‘clf’.
- Hit ‘Submit Answer’ to fit the pipeline to the training data and compute its accuracy.

```
# Import RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier

# Add model step to pipeline: pl
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer())
                ]))
             ]
        )),
        ('clf', RandomForestClassifier(n_estimators=15))
    ])

# Fit to the training data
pl.fit(X_train, y_train)

# Compute and print accuracy
accuracy = pl.score(X_test, y_test)
print("\nAccuracy on budget dataset: ", accuracy)
```

Results :

```
<script.py> output:

    Accuracy on budget dataset:  0.305769230769
```

Wow, you’re becoming a master! It’s time to get serious and work with the log loss metric. You’ll learn expert techniques in the next chapter to take the model to the next level.