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 4 : Learning from the experts

In this chapter, you will learn the tricks used by the competition winner, and implement them yourself using scikit-learn. Enjoy!

How many tokens?

Recall from previous chapters that how you tokenize text affects the n-gram statistics used in your model.

Going forward, you'll use alpha-numeric sequences, and only alpha-numeric sequences, as tokens. Alpha-numeric tokens contain only letters a-z and numbers 0-9 (no other characters). In other words, you'll tokenize on punctuation to generate n-gram statistics.

In this exercise, you'll make sure you remember how to tokenize on punctuation.

Assuming we tokenize on punctuation, accepting only alpha-numeric sequences as tokens, how many tokens are in the following string from the main dataset?

```
'PLANNING,RES,DEV,& EVAL      '
```

If you want, we've loaded this string into the workspace as SAMPLE_STRING, but you may not need it to answer the question.

Possible Answers => 2

- 4, because RES and DEV are not tokens
- 4, because , and & are not tokens
- 7, because there are 4 different words, some commas, an & symbol, and whitespace
- 7, because there are 7 whitespaces

```

```

Results :

Yes! Commas, "&", and whitespace are not alpha-numeric tokens. Keep it up!

Deciding what's a word

Before you build up to the winning pipeline, it will be useful to look a little deeper into how the text features will be processed.

In this exercise, you will use CountVectorizer on the training data X_train (preloaded into the workspace) to see the effect of tokenization on punctuation.

Remember, since CountVectorizer expects a vector, you'll need to use the preloaded function, combine_text_columns before fitting to the training data.

Instructions

- Create text_vector by preprocessing X_train using combine_text_columns. This is important, or else you won't get any tokens!
- Instantiate CountVectorizer as text_features. Specify the keyword argument token_pattern=TOKENS_ALPHANUMERIC.
- Fit text_features to the text_vector.
- Hit 'Submit Answer' to print the first 10 tokens.

```
# Import the CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer

# Create the text vector
text_vector = combine_text_columns(X_train)


# Create the token pattern: TOKENS_ALPHANUMERIC
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

# Instantiate the CountVectorizer: text_features
text_features = CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC)

# Fit text_features to the text vector
text_features.fit(text_vector)

# Print the first 10 tokens
print(text_features.get_feature_names()[:10])
```

Results :

```
<script.py> output:
    ['00a', '12', '1st', '2nd', '3rd', '5th', '70', '70h', '8', 'a']
```

Great work! It's time to start building the winning pipeline!

Which of these is a classification problem?

In this exercise you'll insert a CountVectorizer instance into your pipeline for the main dataset, and compute multiple n-gram features to be used in the model.

In order to look for ngram relationships at multiple scales, you will use the ngram_range parameter as Peter

discussed in the video.

Special functions: You'll notice a couple of new steps provided in the pipeline in this and many of the remaining exercises. Specifically, the dim_red step following the vectorizer step , and the scale step preceeding the clf (classification) step.

These have been added in order to account for the fact that you're using a reduced-size sample of the full dataset in this course. To make sure the models perform as the expert competition winner intended, we have to apply a dimensionality reduction technique, which is what the dim_red step does, and we have to scale the features to lie between -1 and 1, which is what the scale step does.

The dim_red step uses a scikit-learn function called SelectKBest(), applying something called the chi-squared test to select the K "best" features. The scale step uses a scikit-learn function called MaxAbsScaler() in order to squash the relevant features into the interval -1 to 1.

You won't need to do anything extra with these functions here, just complete the vectorizing pipeline steps below. However, notice how easy it was to add more processing steps to our pipeline!

Instructions

- Import CountVectorizer from sklearn.feature_extraction.text.
- Add a CountVectorizer step to the pipeline with the name 'vectorizer'.
  - Set the token pattern to be TOKENS_ALPHANUMERIC.
  - Set the ngram_range to be (1, 2).

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

# Import classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier

# Import CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer

# Import other preprocessing modules
from sklearn.preprocessing import Imputer
from sklearn.feature_selection import chi2, SelectKBest

# Select 300 best features
chi_k = 300

# Import functional utilities
from sklearn.preprocessing import FunctionTransformer, MaxAbsScaler
from sklearn.pipeline import FeatureUnion

# Perform preprocessing
get_text_data = FunctionTransformer(combine_text_columns, validate=False)
get_numeric_data = FunctionTransformer(lambda x: x[NUMERIC_COLUMNS], validate=False)

# Create the token pattern: TOKENS_ALPHANUMERIC
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

# Instantiate 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(token_pattern=TOKENS_ALPHANUMERIC,ngram_range=(
1, 2))),
                    ('dim_red', SelectKBest(chi2, chi_k))
                ]))
            ]
        )),
        ('scale', MaxAbsScaler()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```

Results :

Log loss score: 1.2681. Great work! You'll now add some additional tricks to make the pipeline even better.

Which models of the data include interaction terms?

Recall from the video that interaction terms involve products of features.

Suppose we have two features x and y, and we use models that process the features as follows:

```
βx + βy + ββ
βxy + βx + βy
βx + βy + βx^2 + βy^2
```

where β is a coefficient in your model (not a feature).

Which expression(s) include interaction terms?

Possible Answers => 2

- The first expression
- The second expression
- The third expression
- The first and third expressions.

```

```

Results :

Yes! An xy term is present, which represents interactions between features. Nice work, lets implement this!

Implement interaction modeling in scikit-learn

It's time to add interaction features to your model. The PolynomialFeatures object in scikit-learn does just that, but here you're going to a custom interaction object, SparseInteractions. Interaction terms are a statistical tool that lets your model express what happens if two features appear together in the same row.

SparseInteractions does the same thing as PolynomialFeatures, but it uses sparse matrices to do so. You can get the code for SparseInteractions at this GitHub Gist.

PolynomialFeatures and SparseInteractions both take the argument degree, which tells them what polynomia degree of interactions to compute.

You're going to consider interaction terms of degree=2 in your pipeline. You will insert these steps after the preprocessing steps you've built out so far, but before the classifier steps.

Pipelines with interaction terms take a while to train (since you're making n features into n-squared features!), so as long as you set it up right, we'll do the heavy lifting and tell you what your score is!

Instructions

- Add the interaction terms step using SparseInteractions() with degree=2. Give it a name of 'int', and make sure it is after the preprocessing step but before scaling.

```
# Instantiate 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(token_pattern=TOKENS_ALPHANUMERIC,
                                                   ngram_range=(1, 2))),
                    ('dim_red', SelectKBest(chi2, chi_k))
                ]))
             ]
        )),
        ('int', SparseInteractions(degree=2)),
        ('scale', MaxAbsScaler()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```

Results :

Log loss score: 1.2256. Nice improvement from 1.2681! The student is becoming the master!

Why is hashing a useful trick?

In the video, Peter explained that a hash function takes an input, in your case a token, and outputs a hash value. For example, the input may be a string and the hash value may be an integer.

We've loaded a familiar python datatype, a dictionary called hash_dict, that makes this mapping concept a bit more explicit. In fact, python dictionaries ARE hash tables!

Print hash_dict in the IPython Shell to get a sense of how strings can be mapped to integers.

By explicitly stating how many possible outputs the hashing function may have, we limit the size of the objects that need to be processed. With these limits known, computation can be made more efficient and we can get results faster, even on large datasets.

Using the above information, answer the following:

Why is hashing a useful trick?

Possible Answers => 3

- Hashing isn't useful unless you're working with numbers.
- Some problems are memory-bound and not easily parallelizable, but hashing parallelizes them.
- Some problems are memory-bound and not easily parallelizable, and hashing enforces a fixed length computation instead of using a mutable datatype (like a dictionary).
- Hashing enforces a mutable length computation instead of using a fixed length datatype, like a dictionary.

```
```

Results :

Yes! Enforcing a fixed length can speed up calculations drastically, especially on large datasets!

Implementing the hashing trick in scikit-learn

In this exercise you will check out the scikit-learn implementation of HashingVectorizer before adding it to your pipeline later.

As you saw in the video, HashingVectorizer acts just like CountVectorizer in that it can accept token_pattern and ngram_range parameters. The important difference is that it creates hash values from the text, so that we get all the computational advantages of hashing!

Instructions

- Import HashingVectorizer from sklearn.feature_extraction.text.
- Instantiate the HashingVectorizer as hashing_vec using the TOKENS_ALPHANUMERIC pattern.
- Fit and transform hashing_vec using text_data. Save the result as hashed_text.
- Hit 'Submit Answer' to see some of the resulting hash values.

```
# Import HashingVectorizer
from sklearn.feature_extraction.text import HashingVectorizer

# Get text data: text_data
text_data = combine_text_columns(X_train)

# Create the token pattern: TOKENS_ALPHANUMERIC
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

# Instantiate the HashingVectorizer: hashing_vec
hashing_vec = HashingVectorizer(token_pattern=TOKENS_ALPHANUMERIC)

# Fit and transform the Hashing Vectorizer
hashed_text = hashing_vec.fit_transform(text_data)

# Create DataFrame and print the head
hashed_df = pd.DataFrame(hashed_text.data)
print(hashed_df.head())
```

Results :

```
<script.py> output:
              0
    0 -0.160128
    1  0.160128
    2 -0.480384
    3 -0.320256
    4  0.160128
```

Nice! As you can see, some text is hashed to the same value, but as Peter mentioned in the video, this doesn't neccessarily hurt performance.

Build the winning model

You have arrived! This is where all of your hard work pays off. It's time to build the model that won DrivenData's competition.

You've constructed a robust, powerful pipeline capable of processing training and testing data. Now that you understand the data and know all of the tools you need, you can essentially solve the whole problem in a relatively small number of lines of code. Wow!

All you need to do is add the HashingVectorizer step to the pipeline to replace the CountVectorizer step.

The parameters non_negative=True, norm=None, and binary=False make the HashingVectorizer perform similarly to the default settings on the CountVectorizer so you can just replace one with the other.

Instructions

- Import HashingVectorizer from sklearn.feature_extraction.text.
- Add a HashingVectorizer step to the pipeline.
  - Name the step 'vectorizer'.
  - Use the TOKENS_ALPHANUMERIC token pattern.
  - Specify the ngram_range to be (1, 2)

```
# Import the hashing vectorizer
from sklearn.feature_extraction.text import HashingVectorizer

# Instantiate the winning model 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', HashingVectorizer(token_pattern=TOKENS_ALPHANUMERIC,
                                                     non_negative=True, norm=None, binary=False,
                                                     ngram_range=(1, 2))),
                    ('dim_red', SelectKBest(chi2, chi_k))
                ]))
             ]
```

```
        )),
        ('int', SparseInteractions(degree=2)),
        ('scale', MaxAbsScaler()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```

Results :

Well done! Log loss: 1.2258. Looks like the performance is about the same, but this is expected since the HashingVectorizer should work the same as the CountVectorizer. Try this pipeline out on the whole dataset on your local machine to see its full power!

What tactics got the winner the best score?

Now you've implemented the winning model from start to finish. If you want to use this model locally, this Jupyter notebook contains all the code you've worked so hard on. You can now take that code and build on it!

Let's take a moment to reflect on why this model did so well. What tactics got the winner the best score?

Possible Answers => 3

- The winner used a 500 layer deep convolutional neural network to master the budget data.
- The winner used an ensemble of many models for classification, taking the best results as predictions.
- The winner used skillful NLP, efficient computation, and simple but powerful stats tricks to master the budget data.

```

```

Results :

Yep! Often times simpler is better, and understanding the problem in depth leads to simpler solutions!
Final notebook : https://github.com/datacamp/course-resources-ml-with-experts-budgets
Data : https://www.drivendata.org/competitions/46/box-plots-for-education-reboot/data/
