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 2 : Creating a simple first model

In this chapter, you'll build a first-pass model. You'll use numeric data only to train the model. Spoiler alert - throwing out all of the text data is bad for performance! But you'll learn how to format your predictions. Then, you'll be introduced to natural language processing (NLP) in order to start working with the large amounts of text in the data.

Setting up a train-test split in scikit-learn

Alright, you've been patient and awesome. It's finally time to start training models!

The first step is to split the data into a training set and a test set. Some labels don't occur very often, but we want to make sure that they appear in both the training and the test sets. We provide a function that will make sure at least min_count examples of each label appear in each split: multilabel_train_test_split.

Feel free to check out the full code for multilabel_train_test_split here.

You'll start with a simple model that uses just the numeric columns of your DataFrame when calling multilabel_train_test_split. The data has been read into a DataFrame df and a list consisting of just the numeric columns is available as NUMERIC_COLUMNS.

Instructions

- Create a new DataFrame named numeric_data_only by applying the .fillna(-1000) method to the numeric columns (available in the list NUMERIC_COLUMNS) of df.
- Convert the labels (available in the list LABELS) to dummy variables. Save the result as label_dummies.
- In the call to multilabel_train_test_split(), set the size of your test set to be 0.2. Use a seed of 123.
- Fill in the .info() method calls for X_train, X_test, y_train, and y_test.

```
# Create the new DataFrame: numeric_data_only
numeric_data_only = df[NUMERIC_COLUMNS].fillna(-1000)

# Get labels and convert to dummy variables: label_dummies
label_dummies = pd.get_dummies(df[LABELS])

# Create training and test sets
X_train, X_test, y_train, y_test = multilabel_train_test_split(numeric_data_only,
```

```
                                                    label_dummies,
                                                    size=0.2,
                                                    seed=123)

# Print the info
print("X_train info:")
print(X_train.info())
print("\nX_test info:")
print(X_test.info())
print("\ny_train info:")
print(y_train.info())
print("\ny_test info:")
print(y_test.info())
```

Results :

```
<script.py> output:
    X_train info:
    <class 'pandas.core.frame.DataFrame'>
    Int64Index: 1040 entries, 198 to 101861
    Data columns (total 2 columns):
    FTE      1040 non-null float64
    Total    1040 non-null float64
    dtypes: float64(2)
    memory usage: 24.4 KB
    None

    X_test info:
    <class 'pandas.core.frame.DataFrame'>
    Int64Index: 520 entries, 209 to 448628
    Data columns (total 2 columns):
    FTE      520 non-null float64
    Total    520 non-null float64
    dtypes: float64(2)
    memory usage: 12.2 KB
    None

    y_train info:
    <class 'pandas.core.frame.DataFrame'>
    Int64Index: 1040 entries, 198 to 101861
    Columns: 104 entries, Function_Aides Compensation to Operating_Status_PreK-12 Operating
    dtypes: float64(104)
    memory usage: 853.1 KB
    None

    y_test info:
    <class 'pandas.core.frame.DataFrame'>
    Int64Index: 520 entries, 209 to 448628
    Columns: 104 entries, Function_Aides Compensation to Operating_Status_PreK-12 Operating
    dtypes: float64(104)
    memory usage: 426.6 KB
    None
```

Nice! With the data split, you can now train a model!

Training a model

With split data in hand, you're only a few lines away from training a model.

In this exercise, you will import the logistic regression and one versus rest classifiers in order to fit a multi-class logistic regression model to the NUMERIC_COLUMNS of your feature data.

Then you'll test and print the accuracy with the .score() method to see the results of training.

Before you train! Remember, we're ultimately going to be using logloss to score our model, so don't worry too much about the accuracy here. Keep in mind that you're throwing away all of the text data in the dataset - that's by far most of the data! So don't get your hopes up for a killer performance just yet. We're just interested in getting things up and running at the moment.

All data necessary to call multilabel_train_test_split() has been loaded into the workspace.

Instructions

- Import LogisticRegression from sklearn.linear_model and OneVsRestClassifier from sklearn.multiclass.
- Instantiate the classifier clf by placing LogisticRegression() inside OneVsRestClassifier().
- Fit the classifier to the training data X_train and y_train.
- Compute and print the accuracy of the classifier using its .score() method, which accepts two arguments: X_test and y_test.

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

# Create the DataFrame: numeric_data_only
numeric_data_only = df[NUMERIC_COLUMNS].fillna(-1000)

# Get labels and convert to dummy variables: label_dummies
label_dummies = pd.get_dummies(df[LABELS])

# Create training and test sets
X_train, X_test, y_train, y_test = multilabel_train_test_split(numeric_data_only,
                                                               label_dummies,
                                                               size=0.2,
                                                               seed=123)

# Instantiate the classifier: clf
clf = OneVsRestClassifier(LogisticRegression())

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

# Print the accuracy
print("Accuracy: {}".format(clf.score(X_test,y_test)))
```

Results :

```
<script.py> output:
    Accuracy: 0.0
```

Ok! The good news is that your workflow didn't cause any errors. The bad news is that your model scored the lowest possible accuracy: 0.0! But hey, you just threw away ALL of the text data in the budget. Later, you won't. Before you add the text data, let's see how the model does when scored by log loss.

Use your model to predict values on holdout data

You're ready to make some predictions! Remember, the train-test-split you've carried out so far is for model development. The original competition provides an additional test set, for which you'll never actually see the correct labels. This is called the "holdout data."

The point of the holdout data is to provide a fair test for machine learning competitions. If the labels aren't known by anyone but DataCamp, DrivenData, or whoever is hosting the competition, you can be sure that no one submits a mere copy of labels to artificially pump up the performance on their model.

Remember that the original goal is to predict the probability of each label. In this exercise you'll do just that by using the .predict_proba() method on your trained model.

First, however, you'll need to load the holdout data, which is available in the workspace as the file HoldoutData.csv.

Instructions

- Read HoldoutData.csv into a DataFrame called holdout. Specify the keyword argument index_col=0 in your call to read_csv().
- Generate predictions using .predict_proba() on the numeric columns (available in the NUMERIC_COLUMNS list) of holdout. Make sure to fill in missing values with -1000!

```
# Instantiate the classifier: clf
clf = OneVsRestClassifier(LogisticRegression())

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

# Load the holdout data: holdout
holdout = pd.read_csv('HoldoutData.csv',index_col=0)

# Generate predictions: predictions
predictions = clf.predict_proba(holdout[NUMERIC_COLUMNS].fillna(-1000))
```

Results :

Awesome! Now you can write the predictions to a .csv and submit for scoring!

Writing out your results to a csv for submission

At last, you're ready to submit some predictions for scoring. In this exercise, you'll write your predictions to a .csv using the .to_csv() method on a pandas DataFrame. Then you'll evaluate your performance according to the LogLoss metric discussed earlier!

You'll need to make sure your submission obeys the correct format.

To do this, you'll use your predictions values to create a new DataFrame, prediction_df.

Interpreting LogLoss & Beating the Benchmark:

When interpreting your log loss score, keep in mind that the score will change based on the number of samples tested. To get a sense of how this very basic model performs, compare your score to the DrivenData benchmark

model performance: 2.0455, which merely submitted uniform probabilities for each class.

Remember, the lower the log loss the better. Is your model's log loss lower than 2.0455?

Instructions

- Create the prediction_df DataFrame by specifying the following arguments to the provided parameters pd.DataFrame():
  - pd.get_dummies(df[LABELS]).columns.
  - holdout.index.
  - predictions.
- Save prediction_df to a csv file called 'predictions.csv' using the .to_csv() method.
- Submit the predictions for scoring by using the score_submission() function with pred_path set to 'predictions.csv'.

```
# Generate predictions: predictions
predictions = clf.predict_proba(holdout[NUMERIC_COLUMNS].fillna(-1000))

# Format predictions in DataFrame: prediction_df
prediction_df = pd.DataFrame(columns=pd.get_dummies(df[LABELS]).columns,
                             index=holdout.index,
                             data=predictions)


# Save prediction_df to csv
prediction_df.to_csv('predictions.csv')

# Submit the predictions for scoring: score
score = score_submission(pred_path='predictions.csv')

# Print score
print('Your model, trained with numeric data only, yields logloss score: {}'.format(score))
```

Results :

```
<script.py> output:
    Your model, trained with numeric data only, yields logloss score: 1.9067227623381413
```

Incredible! Even though your basic model scored 0.0 accuracy, it nevertheless performs better than the benchmark score of 2.0455. You've now got the basics down and have made a first pass at this complicated supervised learning problem. It's time to step up your game and incorporate the text data.

Tokenizing text

As we talked about in the video, tokenization is the process of chopping up a character sequence into pieces called tokens.

How do we determine what constitutes a token? Often, tokens are separated by whitespace. But we can specify other delimiters as well. For example, if we decided to tokenize on punctuation, then any punctuation mark would be treated like a whitespace. How we tokenize text in our DataFrame can affect the statistics we use in our model.

A particular cell in our budget DataFrame may have the string content Title I - Disadvantaged Children/Targeted

Assistance. The number of n-grams generated by this text data is sensitive to whether or not we tokenize on punctuation, as you'll show in the following exercise.

How many tokens (1-grams) are in the string

```
Title I - Disadvantaged Children/Targeted Assistance
```

if we tokenize on punctuation?

Possible Answers => 2

- 4
- 6
- 8
- 13

```

```

Results :

Yes! Tokenizing on punctuation means that Children/Targeted becomes two tokens and - is dropped altogether. Nice work!

Testing your NLP credentials with n-grams

You're well on your way to NLP superiority. Let's test your mastery of n-grams!

In the workspace, we have the loaded a python list, one_grams, which contains all 1-grams of the string petro-vend fuel and fluids, tokenized on punctuation. Specifically,

```
one_grams = ['petro', 'vend', 'fuel', 'and', 'fluids']
```

In this exercise, your job is to determine the sum of 1-grams, 2-grams and 3-grams generated by the string petro-vend fuel and fluids, tokenized on punctuation.

Recall that the n-gram of a sequence consists of all ordered subsequences of length n.

Possible Answers => 4

- 3.
- 4.
- 7.
- 12.
- 15.

```
def ngrams(list,n):
    return len(list)-n+1

def sumngrams(list,n):
    res = 0
    for i in range(1,n+1):
```

```
        res=res+len(list)-i+1
    return res
```

Results :

```
    In [2]: def ngrams(list,n):
    ...     return len(list)-n+1

    In [3]: ngrams(one_grams,2)
    Out[3]: 4

    In [4]: ngrams(one_grams,3)
    Out[4]: 3

    In [5]: ngrams(one_grams,1)
    Out[5]: 5

    In [13]: def sumngrams(list,n):
...     res = 0
...     for i in range(1,n+1):
...         res=res+len(list)-i+1
...     return res

In [14]: sumngrams(one_grams,3)
Out[14]: 12
```

Bingo! The number of 1-grams + 2-grams + 3-grams is 5 + 4 + 3 = 12. NLP champion!

Creating a bag-of-words in scikit-learn

In this exercise, you'll study the effects of tokenizing in different ways by comparing the bag-of-words representations resulting from different token patterns.

You will focus on one feature only, the Position_Extra column, which describes any additional information not captured by the Position_Type label.

For example, in the Shell you can check out the budget item in row 8960 of the data using df.loc[8960]. Looking at the output reveals that this Object_Description is overtime pay. For who? The Position Type is merely "other", but the Position Extra elaborates: "BUS DRIVER". Explore the column further to see more instances. It has a lot of NaN values.

Your task is to turn the raw text in this column into a bag-of-words representation by creating tokes that contain only alphanumeric characters.

For comparison purposes, the first 15 tokens of vec_basic, which splits df.Position_Extra into tokens when it encounters only whitespace characters, have been printed along with the length of the representation.

Instructions

- Import CountVectorizer from sklearn.feature_extraction.text.
- Fill missing values in df.Position_Extra using .fillna('') to replace NaNs with empty strings. Specify the additional keyword argument inplace=True so that you don't have to assign the result back to df.
- Instantiate the CountVectorizer as vec_alphanumeric by specifying the token_pattern to be TOKENS_ALPHANUMERIC.

- Fit vec_alphanumeric to df.Position_Extra.
- Hit 'Submit Answer' to see the len of the fitted representation as well as the first 15 elements, and compare to vec_basic.

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

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

# Fill missing values in df.Position_Extra
df.Position_Extra.fillna('', inplace=True)

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

# Fit to the data
vec_alphanumeric.fit(df.Position_Extra)

# Print the number of tokens and first 15 tokens
msg = "There are {} tokens in Position_Extra if we split on non-alpha numeric"
print(msg.format(len(vec_alphanumeric.get_feature_names())))
print(vec_alphanumeric.get_feature_names()[:15])
```

Results :

```
<script.py> output:
    There are 123 tokens in Position_Extra if we split on non-alpha numeric
    ['1st', '2nd', '3rd', 'a', 'ab', 'additional', 'adm', 'administrative', 'and', 'any', 'art',
'assessment', 'assistant', 'asst', 'athletic']
```

Great work! Treating only alpha-numeric characters as tokens gives you a smaller number of more meaningful tokens. You've got bag-of-words in the bag!

Combining text columns for tokenization

In order to get a bag-of-words representation for all of the text data in our DataFrame, you must first convert the text data in each row of the DataFrame into a single string.

In the previous exercise, this wasn't necessary because you only looked at one column of data, so each row was already just a single string. CountVectorizer expects each row to just be a single string, so in order to use all of the text columns, you'll need a method to turn a list of strings into a single string.

In this exercise, you'll complete the function definition combine_text_columns(). When completed, this function will convert all training text data in your DataFrame to a single string per row that can be passed to the vectorizer object and made into a bag-of-words using the .fit_transform() method.

Note that the function uses NUMERIC_COLUMNS and LABELS to determine which columns to drop. These lists have been loaded into the workspace.

Instructions

- Use the .drop() method on data_frame with to_drop and axis= as arguments to drop the non-text data. Save

the result as text_data.
- Fill in missing values (inplace) in text_data with blanks (“”), using the .fillna() method.
- Complete the .apply() method by writing a lambda function that uses the .join() method to join all the items in a row with a space in between.

```
# Define combine_text_columns()
def combine_text_columns(data_frame, to_drop=NUMERIC_COLUMNS + LABELS):
    """ converts all text in each row of data_frame to single vector """

    # Drop non-text columns that are in the df
    to_drop = set(to_drop) & set(data_frame.columns.tolist())
    text_data = data_frame.drop(to_drop,axis=1)

    # Replace nans with blanks
    text_data.fillna('',inplace=True)

    # Join all text items in a row that have a space in between
    return text_data.apply(lambda x: " ".join(x), axis=1)
```

Results :

```
In [1]: NUMERIC_COLUMNS
Out[1]: ['FTE', 'Total']

In [2]: LABELS
Out[2]:
['Function',
 'Use',
 'Sharing',
 'Reporting',
 'Student_Type',
 'Position_Type',
 'Object_Type',
 'Pre_K',
 'Operating_Status']
```

Good job! You’ll put this function to use in the next exercise to tokenize the data!

What’s in a token?

Now you will use combine_text_columns to convert all training text data in your DataFrame to a single vector that can be passed to the vectorizer object and made into a bag-of-words using the .fit_transform() method.

You’ll compare the effect of tokenizing using any non-whitespace characters as a token and using only alphanumeric characters as a token.

Instructions

- Import CountVectorizer from sklearn.feature_extraction.text.
- Instantiate vec_basic and vec_alphanumeric using, respectively, the TOKENS_BASIC and TOKENS_ALPHANUMERIC patterns.
- Create the text vector by using the combine_text_columns() function on df.
- Using the .fit_transform() method with text_vector, fit and transform first vec_basic and then vec_alphanumeric. Print the number of tokens they contain.

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

# Create the basic token pattern
TOKENS_BASIC = '\\S+(?=\\s+)'

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

# Instantiate basic CountVectorizer: vec_basic
vec_basic = CountVectorizer(token_pattern=TOKENS_BASIC)

# Instantiate alphanumeric CountVectorizer: vec_alphanumeric
vec_alphanumeric = CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC)

# Create the text vector
text_vector = combine_text_columns(df)

# Fit and transform vec_basic
vec_basic.fit_transform(text_vector)

# Print number of tokens of vec_basic
print("There are {} tokens in the dataset".format(len(vec_basic.get_feature_names())))

# Fit and transform vec_alphanumeric
vec_alphanumeric.fit_transform(text_vector)

# Print number of tokens of vec_alphanumeric
print("There are {} alpha-numeric tokens in the dataset".format(len(vec_alphanumeric.get_feature_
names())))
```

Results :

```
<script.py> output:
    There are 1405 tokens in the dataset
    There are 1117 alpha-numeric tokens in the dataset
```

Wow, you're on your way to complete Data Domination! Notice that tokenizing on alpha-numeric tokens reduced the number of tokens, just as in the last exercise. We'll keep this in mind when building a better model with the Pipeline object next. See you in the next chapter!
