julia_ml_2_production_model

July 24, 2019

[1]:
```
using Images
using JLD
using PyCall
using PyCallJLD
using PyPlot
using Statistics

using ScikitLearn
using ScikitLearn.CrossValidation: cross_val_score
using ScikitLearn.GridSearch: GridSearchCV
using ScikitLearn.Pipelines: Pipeline, FeatureUnion, make_pipeline



@sk_import datasets: load_breast_cancer
@sk_import decomposition: PCA
@sk_import ensemble: RandomForestClassifier
@sk_import feature_selection: SelectPercentile
@sk_import metrics: f1_score
@sk_import metrics: make_scorer
@sk_import preprocessing: PolynomialFeatures
@sk_import preprocessing: MinMaxScaler
```

```
 Warning: `getindex(o::PyObject, s::AbstractString)` is deprecated in favor of
dot overloading (`getproperty`) so elements should now be accessed as e.g.
`o."s"` instead of `o["s"]`.
   caller = __init__() at PyCallJLD.jl:12
 @ PyCallJLD /opt/julia/packages/PyCallJLD/Tfc36/src/PyCallJLD.jl:12
 Warning: `getindex(o::PyObject, s::AbstractString)` is deprecated in favor of
dot overloading (`getproperty`) so elements should now be accessed as e.g.
`o."s"` instead of `o["s"]`.
   caller = __init__() at PyCallJLD.jl:13
 @ PyCallJLD /opt/julia/packages/PyCallJLD/Tfc36/src/PyCallJLD.jl:13
```

[1]:
```
PyObject <class 'sklearn.preprocessing.data.MinMaxScaler'>
```

1 How to Use a Julia Machine Learning Model in Production

Take advantage of: - docker (jupyter Julia, Python, R) container - scikitlearn.jl - genie.jl (Model View Controller (MVC) framework)

[ ]:
```
img = load("overviews.png")
```

2 Start with SCIKIT LEARN for Julia

[ ]:
```
img = load("scikit.png")
```

3 Get UCI ML Breast Cancer Wisconsin (Diagnostic) dataset

https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html

3.0.1 Good Example of Classic Binary Classification dataset

- 569 observations
- 30 features
- Samples per class 212-Malignant(1) and 357-Benign(0)

[ ]:
```
cancer = load_breast_cancer(); #load data as dictionary
```

[ ]:
```
cancer["data"];
cancer["feature_names"];
```

[ ]:
```
# Set data features values to X's
X = cancer["data"]
size(X)
```

[ ]:
```
# Set Target values to y
y = cancer["target"]
size(y)
```

4 Apply Feature Engineering

We think the model may perform better with polynomial features like: x^2, xy and y^2

Sometimes these feature combinations have more important impact than the original features. Note, they expand number of features 30 to 496

If the `interactions_only` option is not used, the number of produced features is:

$$\#Features = N + N + \frac{N \times (N-1)}{2} + 1$$

E.g. for degree=30, it is 496; for degree=100, it is 5151.

[ ]:
```
# Add Polynomial Features
poly = PolynomialFeatures(2)
X_poly = poly.fit_transform(X)
```

```
size(X_poly)
```

[ ]:
```
# Look at the raw data to see that we might need to scale it.
X_poly[1:4,:]
```

4.0.1 Scale the data

Scale the dat for better performance in subsequent models

[ ]:
```
# compute minimum and maximum on the training data
scaler = MinMaxScaler()
scaler.fit(X_poly)

#rescale training data
X_poly_scaled = scaler.transform(X_poly)
size(X_poly_scaled)
```

[ ]:
```
# Look at the scaled Poly expanded traning data
X_poly_scaled[1:4,:]
```

4.0.2 Now Reduce the Number of Important Features – Select Most Important Engineered Features

Because we increased the number of features (30 to 496), now we can go back and select the features that have most importance using a selection tool called `SelectPrecentile`.

SelectPercentile is a univariate feature selector which says what percentage of features to keep. Think Principal Component Analysis (PCA) for multivariate data

[ ]:
```
#?SelectPercentile
```

[ ]:
```
select = SelectPercentile(percentile=20)
select.fit(X_poly_scaled, y) #need both scaled training and target for fit
X_selected = select.transform(X_poly_scaled)
size(X_selected)
```

[ ]:
```
# Look at the selected  expanded traning data of the most dominant features␣
 ↪looks like column 2 from X_poly_selected
# starts out
X_selected[1:4,:]
```

5 Test Feature Engineered Data Against Known Model : RandomForestClassifier

5.0.1 F1 Score

There are several different algorithms that attempt to *blend* precision and recall to produce a single "score." Scikit-learn provides a number of other scalar scores that are useful for differing purposes (and other libraries are similar), but F1 score is one that is used very frequently. It is simply:

$$\text{F1} = 2 \times \frac{precision \times recall}{precision + recall}$$

We get precision and recall from Confusion Table/Contingency Matrix entries
Consider a binary problem though:

Predict/Actual	Positive	Negative
Positive	some_val	some_val
Negative	some_val	some_val

Here, Precision is:

$$\text{Precision} = \frac{true\ positive}{true\ positive + false\ positive}$$

Generalizing that to the multi-class case, the formula is as follows (for i being the index of the class):

$$\text{Precision}_i = \frac{M_{ii}}{\sum_i M_{ij}}$$

And, Recall is:

$$\text{Recall} = \frac{true\ positive}{true\ positive + false\ negative}$$

Generalizing that to the multi-class case:

$$\text{Recall}_i = \frac{M_{ii}}{\sum_j M_{ij}}$$

F1 score can be generalized to multi-class models by averaging the F1 score across each class, counting only correct/incorrect per class.

[ ]:
```
rfc = RandomForestClassifier(max_depth=7, random_state=1)
```

[ ]:
```
typeof(rfc)
```

[ ]:
```
scorer = make_scorer(f1_score)
```

[ ]:
```
#Cross-valdiation Model Check
#Cross-validation is to help train and test different groupings of the data so␣
↪we ensure better model performance
# reduce bias of data, help make sure we do not miss patterns or trends
# identify overfitting

cv_scores = cross_val_score(rfc, X_selected, y, scoring=scorer, cv = 5)#␣
↪Stratified kfold done here.
println(" CV scores: ", cv_scores)
println("Mean score: ", mean(cv_scores))
```

6 Using Pipelines - Bundle all Your Operations

A pipeline is simply an abstraction in scikit-learn to bundle together steps like those used above into a single model interface, following the same APIs as a model itself. A particular pipeline is likely to be somewhat domain specific in that you may learn that those particular steps are useful for e.g. cancer data, but not as useful for data with very different characteristics.

[ ]:
```
img = load("pipeline-diagram.png")
#Image credit (CC-BY-NA): [Karl Rosaen](http://karlrosaen.com/ml/learning-log/
 ↪2016-06-20/)
```

6.1 Pipelines with Grid Search

Grid Search is a way of testing out our hyperparameters to suss out the most ideal model parameters.

First create the pipe or the list of procedures with some defaults

[ ]:
```
pipe = make_pipeline(
    PolynomialFeatures(2),
    MinMaxScaler(),
    SelectPercentile(percentile=20),
    RandomForestClassifier(max_depth=7))
#pipe.steps
```

6.1.1 Now add the GridSearch part

[ ]:
```
@time begin
   params = Dict("polynomialfeatures__degree"=> [1, 2, 3],
              "selectpercentile__percentile"=> [10, 15, 20, 50],
              "randomforestclassifier__max_depth"=> [5, 7, 9],
              "randomforestclassifier__criterion"=> ["entropy", "gini"])

    grid = GridSearchCV(pipe, params, cv=5)
    fit!(grid, X, y)

    print("best cross-validation accuracy:", grid.best_score_)
    #print("best dataset score: ", grid.grid_scores_)    # Overfitting against␣
 ↪entire dataset
    print("best parameters: $(grid.best_params_)")
end
```

6.2 Choose best model as overall learning model… THE MODEL!

[ ]:
```
print("best parameters: $(grid.best_params_)")
```

6.3 Apply ideal model to original dataset X not X_selected

[ ]:
```
model = RandomForestClassifier(max_depth=9, criterion="entropy", random_state=1)
cv_scores = cross_val_score(model, X, y, scoring=scorer, cv=5)# Stratified␣
 ↪kfold done here.
println(" CV scores: ", cv_scores)
println("Mean score: ", mean(cv_scores))
```

6.3.1 Check that Classifier [1-Malignant , 0-Benign] works

X[50] was Malignant
X[11] was Benign

[ ]:
```
transpose([X[11,:] X[50,:]])
```

[ ]:
```
#r_model = fit!(model, X, y)
fit!(model, X, y)

#results = predict(model,transpose([X[50,:] X[11,:]]))
results = predict(model,transpose([X[50,:]]))
```

7 Now we can Serialize the Model and store it for usage in other .jl files

8 JLD

Julia's Data Format

uses Hierarchical Data Format 5 (HDF5) https://en.wikipedia.org/wiki/Hierarchical_Data_Format open source file formats (HDF4, HDF5) designed to store and organize large amounts of data. open source file format for storing huge amounts of numerical data. It's typically used in research applications (meteorology, astronomy, genomics etc.) to distribute and access very large datasets without using a database. One can use HDF5 data format for pretty fast serialization to large datasets.

Serialization is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later (possibly in a different computer environment) This process of serializing an object is also called marshalling an object. The opposite operation, extracting a data structure from a series of bytes, is deserialization (also called unmarshalling).

$array = array("a" => 1, "b" => 2, "c" => array("a" => 1, "b" => 2));

serialized in JSON to

`$json = json_encode($array);` will give you this:

{"a":1,"b":2,"c":{"a":1,"b":2}}

[ ]:
```
#import JLD, PyCallJLD
JLD.save("cancer_model.jld", "model", model)
```