METIS

# Day 11: Pickles, Grids and Pipelines

John Navarro
john.navarro@thisismetis.com
https://www.linkedin.com/in/johnnavarro/

METIS

# What is a pipeline?

A pipeline is a way of linking the data wrangling and model training stages, and/or the data wrangling and model prediction stages, of the machine learning process in Python, so that they can be run together with a single line of code.

For example, data wrangling in the form of scaling and dimensionality reduction many be linked with model training for a logistic regression model. Or data wrangling in the form of scaling and dimensionality reduction may be linked with model prediction by a logistic regression model.

For more details, see http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

METIS

# Why use a pipeline?

- It makes code more readable
- You don't have to worry about keeping track data during intermediate steps, for example between transforming and estimating.
- It makes it trivial to move ordering of the pipeline pieces, or to swap pieces in and out.
- It allows you to do GridSearchCV on your workflow

METIS

# Without Pipeline

```
#get categorical features
#drop off last column because its unnecessary
X_categorical =
pd.get_dummies(abalone_data[categorical_columns]).astype(int).iloc[:,:
-1]

#get and transform numeric features
X_numeric = abalone_data[numeric_columns]
X_numeric[numeric_columns] = StandardScaler().fit_transform(X_numeric)

#get outcome variable
y = abalone_data[target]

#combine transformed categorical and numeric features
X_final = pd.concat((X_numeric,X_categorical),axis=1)
```

# Without Pipeline

```
#create rf regressor and check 10-fold RMSE
rf = RandomForestRegressor()
cross_val_scores = np.abs(cross_val_score(rf,X_final,y,scoring =
"neg_mean_squared_error", cv=10))
rmse_cross_val_scores = np.sqrt(cross_val_scores)
```

METIS

# With Pipeline

```
from sklearn.base import BaseEstimator, TransformerMixin

class ItemSelector(BaseEstimator, TransformerMixin):
    def __init__(self, key):
        self.key = key

    def fit(self, x, y=None):
        return self

    def transform(self, data_dict):
        return data_dict[self.key]
```

# With Pipeline

```
from sklearn.pipeline import FeatureUnion, Pipeline
from sklearn.preprocessing import OneHotEncoder

#encode the categorical column from strings to ints
le = LabelEncoder()
abalone_data["sex_encoded"] = abalone_data[[categorical_columns]].apply(le.fit_transform)

#extract the y
y = abalone_data.age

#create the feature union for the features
X_transformed_pipe = FeatureUnion(
        transformer_list=[
            # Pipeline for one hot encoding categorical column
            ('sexes', Pipeline([
                ('selector', ItemSelector(key=["sex_encoded"])),
                ('encoder', OneHotEncoder())
            ])),
            # Pipeline for pulling out numeric features and scaling them
            ('numeric', Pipeline([
                ('selector', ItemSelector(key=numeric_columns)),
                #('polyfeatures', PolynomialFeatures(degree=2,interaction_only=True)),
                ('scaler', StandardScaler()),
            ]))])
#create the full final pipeline
full_pipeline = Pipeline([("all_features",X_transformed_pipe),
("rf_regressor",RandomForestRegressor(n_estimators=100))])
```

# With Pipeline

```
cross_val_scores = 
np.abs(cross_val_score(full_pipeline,abalone_data,y,cv=10,scorin
g="neg_mean_squared_error"))
rmse_cross_val_scores = np.sqrt(cross_val_scores)

>> Mean 10-fold rmse:  2.13930248305
>> Std 10-fold rmse:  0.622005786443
```

METIS

# With Pipeline

```
full_pipeline.steps

[('all_features', FeatureUnion(n_jobs=1,
         transformer_list=[('categoricals', Pipeline(memory=None,
       steps=[('selector', ItemSelector(key=['rbc', 'pc', 'pcc', 'ba', 'htn',
'dm', 'cad', 'appet', 'pe', 'ane'])), ('imputer', Imputer(axis=0, copy=True,
missing_values=0, strategy='most_frequent',
      verbose=0)), ('encoder', OneHotEncoder(cat...tegy='median', verbose=0)),
('scaler', StandardScaler(copy=True, with_mean=True, with_std=True))]))],
         transformer_weights=None)),
 ('rf_classifier',
  RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
              max_depth=None, max_features='auto', max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
              oob_score=False, random_state=None, verbose=0,
              warm_start=False))]
```

# With Pipeline

```
full_pipeline.fit(X,y)
```

# Pipelines

```
import sklearn.pipeline

select = sklearn.feature_selection.SelectKBest(k=100)
clf = sklearn.ensemble.RandomForestClassifier()

steps = [('feature_selection', select),
        ('random_forest', clf)]

pipeline = sklearn.pipeline.Pipeline(steps)
```

# Pipelines

```
X_transformed_pipe = FeatureUnion(
  transformer_list=[
  ('categoricals', Pipeline([
     ('selector', ItemSelector(key=kidney_columns[14:-1])),
     ('imputer',
 Imputer(missing_values=0,strategy="most_frequent",axis=0)),
    ('encoder', OneHotEncoder())
 ])) ...
```

# Pipelines

```
pipeline.fit( X_train, y_train )

y_prediction = pipeline.predict( X_test )

report = sklearn.metrics.classification_report( y_test,
y_prediction )

print(report)
```

METIS

# Grid Search for Hyper-parameter Optimization

A hyper-parameter is a parameter of a machine learning model which is not set by the model training process itself, but which is a constraint on that process and which defines the structure of the model. For example, for a linear regression model, the number (and nature) of the predictors $\{X_i\}$ can be considered hyper-parameters, whereas the values of the coefficients $\{\beta_i\}$ are not hyper-parameters. For a decision tree, the depth of the tree is a hyper-parameter, whereas the thresholds upon which decisions are made are not hyper-parameters.

Grid search is a brute-force method of selecting the hyper-parameters by simply trying many different values and combinations thereof, and choosing those which perform best. For more details, see http://scikit-learn.org/stable/modules/grid_search.html

METIS

# Grid Search

```
RandomForestClassifier(bootstrap=True,
class_weight=None,
criterion='gini',
max_depth=None,
max_features='auto',
max_leaf_nodes=None,
min_impurity_decrease=0.0,
min_impurity_split=None,
min_samples_leaf=1,
min_samples_split=2,
min_weight_fraction_leaf=0.0,
n_estimators=10,
n_jobs=1,
oob_score=False
)
```

METIS

# Tuning these parameters

- GridSearchCV:  You provide a list of possible parameters
- RandomizedSearchCV:  Random combinations are searched

# Grid Search

```
param_grid = [
  {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
  {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
'kernel': ['rbf']},
 ]
```

# Grid Search

```
# use a full grid over all parameters
param_grid = {"max_depth": [3, None],
              "max_features": [1, 3, 10],
              "min_samples_split": [2, 3, 10],
              "min_samples_leaf": [1, 3, 10],
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}

# run grid search
grid_search = GridSearchCV(clf, param_grid=param_grid)
start = time()
grid_search.fit(X, y)

print("GridSearchCV took %.2f seconds for %d candidate parameter settings."
      % (time() - start, len(grid_search.cv_results_['params'])))
report(grid_search.cv_results_)
```

*http://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html*

METIS

# Random Search

```
# specify parameters and distributions to sample from
param_dist = {"max_depth": [3, None],
              "max_features": sp_randint(1, 11),
              "min_samples_split": sp_randint(2, 11),
              "min_samples_leaf": sp_randint(1, 11),
              "bootstrap": [True, False],
              "criterion": ["gini", "entropy"]}

# run randomized search
n_iter_search = 20
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
                                   n_iter=n_iter_search)

start = time()
random_search.fit(X, y)
print("RandomizedSearchCV took %.2f seconds for %d candidates"
      " parameter settings." % ((time() - start), n_iter_search))
report(random_search.cv_results_)
```

*http://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html*

METIS

# Pickling Your Model

- Pickling your model allows you to preserve your model to be used later.

Pickling is a way of saving the parameters and state of a machine learning model in Python, and packaging it in a form which can be retrieved later without repeating the process of training the model or running it until it reaches a desired state.

For more details, see https://docs.python.org/3/library/pickle.html

# Pickling Your Model

In order to rebuild a similar model with future versions of scikit-learn, additional metadata should be saved along the pickled model:

- The training data, e.g. a reference to a immutable snapshot
- The python source code used to generate the model
- The versions of scikit-learn and its dependencies
- The cross validation score obtained on the training data

# Pickling Your Model

```
#Saving your model
from sklearn.externals import joblib
joblib.dump(clf, 'filename.pkl')

#Loading your model
clf = joblib.load('filename.pkl')
```