Use Scikit-Learn Pipelines to clean data and train models faster

A quick guide to incorporating Pipelines into your machine learning workflow

Collin Ching Sep 19, 2019 · 5 min read ★

Photo by Gerd Altmann from Pixabay

If you're looking for a way to organize your data processing workflow and decrease code redundancy, Scikit-Learn Pipelines will make a great addition to your data science

predictions.

What is a Scikit-Learn Pipeline?

Pipeline can be a pretty vague term, but it's quite apt once you realize what it does in the context of building a machine learning model. A Scikit-Learn Pipeline chains together multiple data processing steps into a single, callable method.

For example, say you want to transform continuous features from the movie data.

	budget	popularity	runtime
2545	45000000.0	14.547939	133.0
2415	0.0	10.597116	98.0
1919	0.0	4.010441	96.0
1462	8000000.0	8.172013	101.0
220	12000000.0	7.216103	132.0

Continuous features from movie data

To process continuous data for a regression model, a standard processing workflow involves imputing missing values, transforming skewed variables, and standardizing your data. You could process the data in separate steps, like so.

```
cont_vars = ['budget', 'popularity', 'runtime']

imputer = SimpleImputer(strategy = 'median')
transformer = PowerTransformer(method = 'yeo-johnson', standardize =
False)
scaler = StandardScaler()

X_train[cont_vars] = imputer.fit_transform(X_train[cont_vars])
X_train[cont_vars] = transformer.fit_transform(X_train[cont_vars])
X_train[cont_vars] = scaler.fit_transform(X_train[cont_vars])
```

```
cont_pipeline = make_pipeline(
    SimpleImputer(strategy = 'median'),
    PowerTransformer(method = 'yeo-johnson', standardize = False),
    StandardScaler()
)

X_train[cont_vars] = cont_pipeline.fit_transform(train[cont_vars],
columns = cont_vars)
```

By using a Pipeline, you can see your processing steps clearly and quickly add or remove steps. You also only have to call the `fit_transform()` once, rather than 3 times.

Use Pipelines to process different data types, in sync

I used a Pipeline to process continuous data, but there are also discrete numeric columns, categorical columns, and JSON-type columns in the movie data. Each of these data types requires a different processing method, so you can build a unique Pipeline for each data type.

```
disc_vars = list(X_train.select_dtypes(include = int).columns)

disc_pipeline = make_pipeline(
    SimpleImputer(strategy = 'constant', fill_value = -1)
)

cat_vars = ['original_language', 'release_season']

cat_pipeline = make_pipeline(
    SimpleImputer(strategy = 'constant', fill_value = 'unknown'),
    OneHotEncoder()
)

json_vars = ['Keywords', 'crew_department', 'production_countries',
'cast_name', 'crew_job', 'production_companies', 'crew_name',
'genres', 'spoken_languages']

json_pipeline = make_pipeline(
    TopCatEncoder()
)
```

`TopCatEncoder()` and some helper functions to parse JSON into categorical variables based on key, then retain top categories for each of those new categorical variables. That's all you need to know about the transformer, but if you want to learn more you can also check out the code in my GitHub.

Now that I have all the pipelines I need to process my data — `cont_pipeline`, `disc_pipeline`, `cat_pipeline` and `json_pipeline` — I can assemble these into a single Pipeline using `ColumnTransformer()` to specify which pipelines transform which variables. Transformers are specified as a list of tuples that look like this: `(name, transformer, columns)`.

```
preprocessor = ColumnTransformer(
    transformers = [
        ('continuous', cont_pipeline, cont_vars),
        ('discrete', disc_pipeline, disc_vars),
        ('categorical', cat_pipeline, cat_vars),
        ('json', json_pipeline, json_vars)
    ]
)
```

To transform all of my data using this Pipeline, I just call `preprocessor.fit_transform(X_train)`.

Use Pipelines to benchmark machine learning algorithms

Here, I use a utility function called `quick_eval()` to train my model and make test predictions.

```
def quick_eval(pipeline, X_train, y_train, X_test, y_test, verbose=True):
    """
    Quickly trains modeling pipeline and evaluates on test data.      Returns original model, tr
    RMSE as a tuple.
    """

    pipeline.fit(X_train, y_train)
    y_train_pred = pipeline.predict(X_train)
    y_test_pred = pipeline.predict(X_test)
```

```
    test_score = np.sqrt(mean_squared_error(y_test, y_test_pred))

    if verbose:
        print(f"Regression algorithm: {pipeline.named_steps['regressor'].__class__.__name__}")
        print(f"Train RMSE: {train_score}")
        print(f"Test RMSE: {test_score}")

    return pipeline.named_steps['regressor'], train_score, test_score

regressors = [
    LinearRegression(),
    Lasso(alpha=.5),
    Ridge(alpha=.1),
    LassoLars(alpha=.1),
    DecisionTreeRegressor(),
    RandomForestRegressor(),
    AdaBoostRegressor(),
    GradientBoostingRegressor()
]

for r in regressors:
    pipe = Pipeline(steps = [
        ('preprocessor', preprocessor),
        ('regressor', r)
    ])

    quick_eval(pipe, X_train, y_train, X_test, y_test)
```

pipe_benchmark.py hosted with ♡ by GitHub view raw

By combining the `processor` pipeline with a regression model, `pipe` handles data processing, model training, and model evaluation all at once, so that we can quickly compare baseline model performance for 8 different models. The output looks like this.

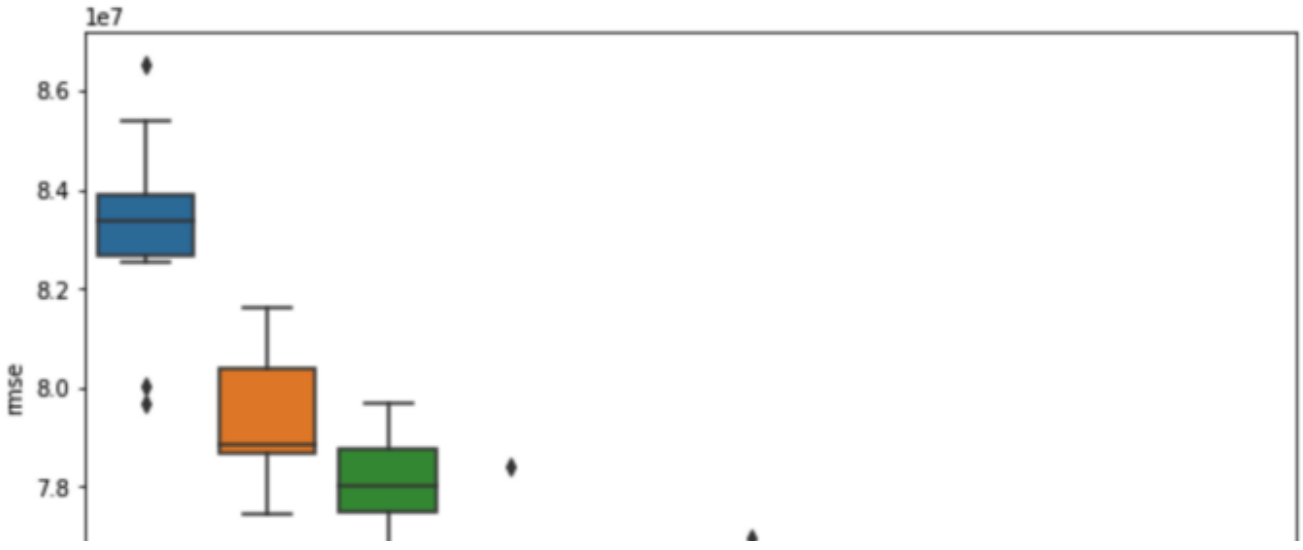
```
Regression algorithm: LinearRegression
Train RMSE: 90680698.96219109
Test RMSE: 233337173.59506416

Regression algorithm: Lasso
Train RMSE: 90680698.6002461
Test RMSE: 233337843.81985235
```

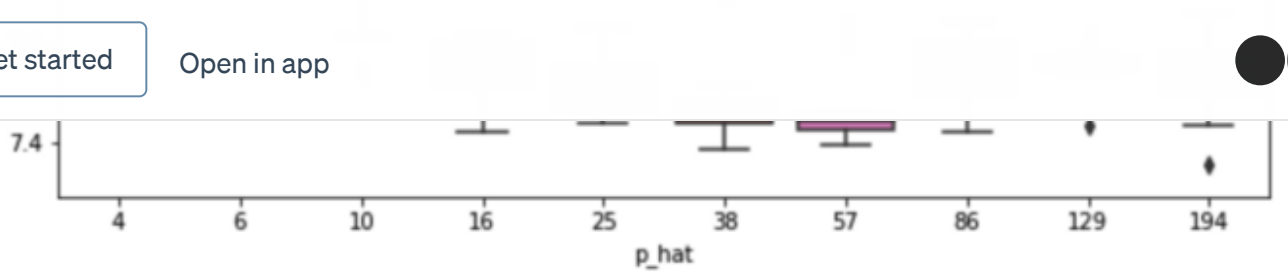
```
Regression algorithm: LassoLars
Train RMSE: 90680698.6003168
Test RMSE: 233337797.46264017

Regression algorithm: DecisionTreeRegressor
Train RMSE: 0.0
Test RMSE: 112298067.15162858

Regression algorithm: RandomForestRegressor
Train RMSE: 33382792.78936104
Test RMSE: 84013907.54387705

Regression algorithm: AdaBoostRegressor
Train RMSE: 111256825.70390953
Test RMSE: 118403122.15438783

Regression algorithm: GradientBoostingRegressor
Train RMSE: 45758323.39709334
Test RMSE: 80865500.98218118
```

Next steps: model selection, feature selection, and a final model

With the data processing and model prototyping steps completed, we can select a subset of models that to focus on. I used the Random Forest regressor because it performed comparatively pretty well and is pretty simple to interpret.

From there, we can proceed with feature engineering, feature selection, and hyperparameter tuning to arrive at a final model. I used a cross-validated variable selection procedure based on Random Forests to whittle my data from 194 features down to 57 features and improve test RMSE.





5-fold CV model performance, p_hat is number of features fed into Random Forest regressor

This variable selection implementation isn't the focus of this post, but you can find the implementation of it in my code in the function `rf_variable_selection()`.

```
# 57 features
final_pipeline = Pipeline(steps = [
                    ('preprocessor', preprocessor),
                    ('selector', TopFeatureSelector(feature_importances,57)),
                    ('regressor', RandomForestRegressor(n_estimators=200,max_features=20))
                ])

quick_eval(final_pipeline, X_train, y_train, X_test, y_test);
```

movies_final_pipe.py hosted with ♡ by GitHub view raw

Above, `TopFeatureSelector()` is another custom transformer that selects the top k features to keep using pre-computed feature importances.

```
Regression algorithm: RandomForestRegressor
Train RMSE: 27724591.63292086
Test RMSE: 79698047.57052584
```

And there you have it, the final model! It performs a little bit better than the baseline with fewer features.

Additional steps to take

Since I focused on Scikit-Learn Pipelines, I skipped a few steps like **incorporating external data, feature engineering, and hyperparameter tuning.** If I were to revisit this project to make my models stronger, I would focus on these three things. I also

Thanks for reading! I'm new to Scikit-Learn Pipelines and this blog post helped me solidify what I've done so far. I'll update this document with new use cases I come across.

You can check out the Jupyter Notebook for this project here. For more info about Pipelines, checkout Rebecca Vickery's post and Scikit-Learn's official guide to Pipelines.

Machine Learning Scikit Learn Regression Data Processing Python