Python Applied to Machine Learning and Statistics

Lecture 05: scikit-learn

Pedro Costa Ricardo Cruz Kelwin Fernandes Adrián Galdrán
Chetak Kandaswamy

September 12, 2016

Agenda

Day 1
1. Classification
2. Regression
3. Clustering
4. Dimensionality Reduction

Day 2
1. Model Selection
2. Preprocessing

Interface

- model.fit(data, [labels])
- **Supervised:**
  - **model.predict(data)**: An array containing the classes of every example in *data*.
  - **model.predict_proba(data)**: The probability of belonging to each of the classes.
  - **model.score(data, labels)**: A measure of how good the fit was. Scores lie between 0 and 1.
- **Unsupervised:**
  - **model.predict(data)**: The cluster of each example.
  - **model.transform(data)**: Maps the data into the model's space.

Some estimator implement the **fit_predict** or **fit_transform** for efficiency reasons.

Cross-Validation

- It is always a good practice to evaluate your model on a different data set than the one you trained;
- You can do that on scikit-learn by using the *train_test_split* method:

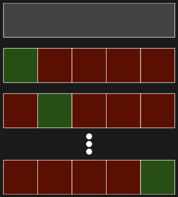
```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
    train_size=0.75, random_state=0)

clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

Cross-Validation

- If we are using the test set to tune the model's hyper-parameters, there is the risk of overfitting to the test set. We need a third set called the validation set, which is used to find the best hyper-parameters of the model;
- By dividing the data into three sets we may end up with few examples to train. The solution is to train the model with cross-validation;
- We divide the training set into k-folds and perform k fits, where each fold takes turns as the validation set.

```
from sklearn.cross_validation import cross_val_score
cross_val_score(clf, X, y, cv=5)
# [0.92, 1., 0.97, 0.95, 0.93]
```



cross_val_score

```python
scores = cross_val_score(clf, X, y, cv=5)
```

- We can use the mean of the scores as the model's score;
- The standard deviation gives us the confidence in the score;

Score estimate and the 95% confidence interval:

```python
"Accuracy = {0:.2f} +/- {1:2f}".format(scores.mean(),
      scores.std() * 2)
# "Accuracy = 0.95 +/- 0.01"
```

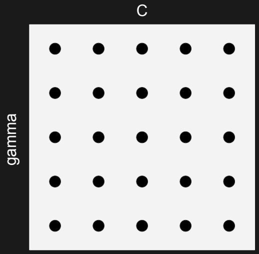
cross_val_score

- When the estimator derives from the *ClassifierMixin*, the strategy for dividing the dataset is the *StratifiedKFold*. This means that the percentage of samples of each class is preserved inside each fold;
- Otherwise, it uses the *KFold* strategy, which divides the dataset without shuffling the data;
- You can change the strategy by setting the *cv* parameter cross-validation generator (i.e. *LeaveOneOut*) or an iterable that yields train/test splits.
- The *score* method of the estimator is used by default, but it is possible to use another metric by properly setting the *scoring* parameter.

```
cross_val_score(clf, X, y, cv=LeaveOneOut(len(X)),
    scoring='roc_auc')
```

Grid Search

- Most estimators have hyper-parameters that must be set by the user (i.e. kernel of *SVM*'s);
- Grid Search generates candidates from a grid of parameter values specified by the user;
- The model is trained at each "point", and the best one is saved.



Grid Search

1. Define a grid:

```
param_grid = [
    {'C': [1, 1e1], 'gamma': [1e-3, 1e-4]},
    {'C': [1e1, 1e2, 1e3], 'kernel': ['poly', 'rbf']},
]
```

2. Search the optimal parameters:

```
grid = GridSearchCV(clf, param_grid=param_grid, cv=5)
grid.fit(X_train, y_train)
```

3. Evaluate the results:

*grid_scores_* *best_estimator_* *best_score_* *best_params_*

4. Or use the *GridSearchCV* object as the estimator:

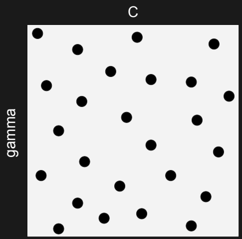
```
grid.predict(X_test)
grid.score(X_test, y_test)
grid.transform(X_test)
```

Grid Search

- Other parameters:
  - *scoring*: use another score function, instead of the model's default;
  - *fit_params*: a dictionary with additional params to be passed to the *fit* (i.e. *sample_weight*);
  - *n_jobs*: jobs to run in parallel;
  - *error_score*: value to give to a score when the fit raises an error, or 'raise' to throw an exception.

Random Search [1]

- Search the parameter space randomly;
- Some parameters do not influence model's performance;
- The number of iterations is independent of the number of parameters to try and their possible values.



[1] Bergstra, J. and Bengio, Y., Random search for hyper-parameter optimization, The Journal of Machine Learning Research (2012)

Random Search

1. Define the parameter distributions:

```
from scipy.stats import randint, uniform
param_distributions = {
    'C': randint(1, 11), 'kernel': ['poly', 'rbf'],
    'gamma': uniform(loc=1e-4, scale=9e-4),
}
```

2. Search the optimal parameters:

```
rsearch = RandomizedSearchCV(clf, param_distributions=param_distributions, cv=5, scoring='f1')
rsearch.fit(X_train, y_train)
```

3. Use the *RandomizedSearchCV* as you used *GridSearchCV*:

```
rsearch.predict(X_test)
rsearch.score(X_test, y_test)
rsearch.transform(X_test)
```

Metrics

- There are 3 approaches to evaluate your estimator:
  - **Score method**: the one provided by the estimator;
  - **Scoring parameter**: the one used on *GridSearchCV* and *RandomizedSearchCV*;
  - **Metric functions**: implemented on scikit-learn's *metrics* module.

Metrics

- There are 3 approaches to evaluate your estimator:
    - Score method;
    - Scoring parameter;
    - **Metric functions**.
- Examples:

```
y_pred = clf.predict(X_test)

accuracy = accuracy_score(y_true, y_pred)
cross_entropy = log_loss(y_true, y_pred)
mse = mean_squared_error(y_true, y_pred)
```

Metrics Interface

- Metrics ending in *_score* return a value where higher is better;
- Metrics ending in *_loss* or *_error* return a value where lower is better;
- It is possible to use these metrics on the *GridSearchCV* and *RandomizedSearchCV* *scoring* parameter with *make_scorer*:

```
grid = GridSearchCV(clf, param_grid, scoring=make_scorer(
    accuracy_score)
```

- Create your own scorer with the *make_scorer* function:

```
def dummy_score_function(truth, prediction):
  return 1
score = make_scorer(dummy_score_function, greater_is_better=True)
score(clf, X, y_true)
# 1
score = make_scorer(dummy_score_function,greater_is_better=False)
score(clf, X, y_true)
# -1
```

Metrics Interface

- It is possible to pass arguments to the *dummy_score_function*:

```python
def dummy_score_function(truth, prediction, dummy=1):
  return dummy

score = make_scorer(dummy_score_function, dummy=2)
score(clf, X, y_true)
# 2
```

- Makes it useful to create a wrapper over other score functions with parameters different from the default ones:

```python
count_correct = make_scorer(accuracy_score, normalize=False)
```

- Makes it possible to use any function on the *GridSearchCV* and *RandomizedSearchCV scoring* parameter:

```python
grid = GridSearchCV(clf, param_grid, scoring=count_correct)
```

Preprocessing

- Methods that transform the input data so they are more suitable for the estimator;
- Standardize input data to have 0 mean and unit variance;

```
X_normalized = StandardScaler().fit_transform(X_train)
```

- Perform *One Hot Encoding*;

```
encoded = OneHotEncoder(categorical_features=[0, 1]).
    fit_transform(X_train)
```

- Add *Polynomial Features*;

```
X_transformed = PolynomialFeatures(degree=3).fit_transform(
    X_train)
```

Pipeline

- Allows to chain multiple estimators into one;

```
Pipeline([('scaler', scaler), ('pca', pca), ('clf', clf)])
```

- Merge preprocessing, feature selection and classification into one single estimator: only one *fit* and *predict*;

```
pipeline.fit(X, [y])
pipeline.predict(X)
pipeline.transform(X)
```

- Search the hyper-parameters jointly;

```
GridSearchCV(pipeline, param_grid)
```

- *make_pipeline* is a convenient method to create a pipeline without naming the estimators.

```
make_pipeline(scaler, pca, clf)
```

Feature Union

- Combines multiple transformers into one;

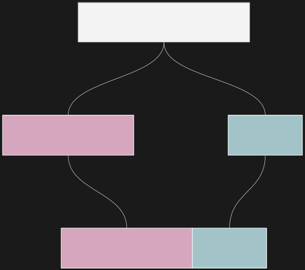
```
union = FeatureUnion([('rbm', rbm), ('pca', pca)])
```

- You can use it on a *Pipeline*;

```
Pipeline([('union', union), ('clf', clf)])
```

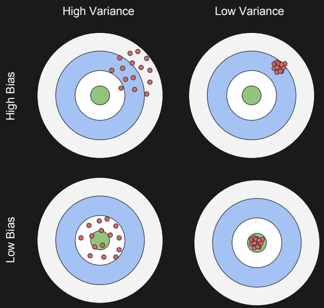
- Or simply use it as a transformer;

```
union.transform(X)
```



Debug your model

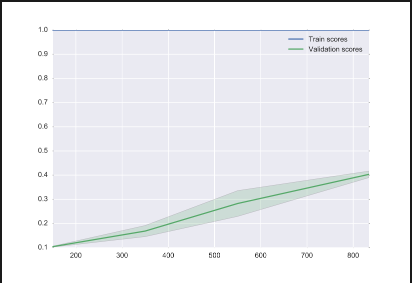
- There are two sources of error: **bias** and **variance** [2];
- **Bias** is the learner's tendency of consistently learning the wrong thing;
- **Variance** is the learner's tendency of learning the noise of the signal.



[2] Domingos, P. (2012). A few useful things to know about machine learning. Communications of the ACM, 55(10), 78–87

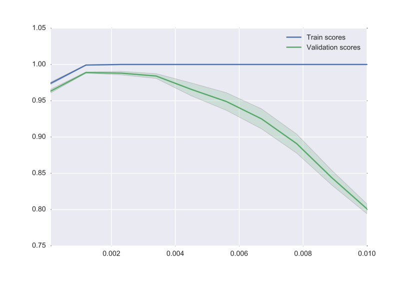
Learning curve

- Evaluate how well your model fits the data as you train it with more data;
- When training score decreases while the validation score increases before stabilizing => *High Bias*;
- When the training score is high while the validation score is low => *High Variance*.



Validation curve

- See how the score changes with the variation of a hyper-parameter;
- This is not a substitute of *Grid Search* / *Random Search*.



Model persistence

- It is possible to use pickle to save the model;

```
pickle.dump(clf, 'model.pkl')
clf2 = pickle.load('model.pkl')
```

- It is better to use the *joblib* module since it is more efficient saving large numpy arrays;

```
from sklearn.externals import joblib
joblib.dump(clf, 'model.pkl')
clf2 = joblib.load('model.pkl')
```

- The *joblib* interface is the same as *pickle*'s.

Final notes

- It is possible to clone an estimator. The clone will have the same hyper-parameters without being fit on any data;

```
from sklearn.base import clone
clf2 = clone(clf)
```

- There is a lot more to Machine Learning than this;
- Get your hands dirty;
- Have fun :) Good luck with your projects!