Intro to deep learning

Dr. Janoś Gabler, University of Bonn

Lecture 4: Classification with sklearn

Topics

- Machine learning vs. Econometrics
- Loading datasets in sklearn
- Splitting data for training and testing
- Fitting models
- Evaluating models
- Cross validation
- Hyperparameter tuning

The Fundamental difference

Econometrics

- Estimate **fundamentally unobservable** parameters and test hypotheses about them
- Cannot test how well it worked
- Focus on justifying assumptions

(Supervised) Machine learning

- Predict observable things
- Can check how well it works
- Focus on experimentation, evaluation and finding out what works

Some implications

- Even though it is tempting, you cannot interpret parameters
- Can be creative in combining simple models into complex ones
- Rapid progress and development of new models
- Programming skills matter more

- Very good paper!
- Difference of ML and Econometrics
- Overview of ML methods
- What we can(not) learn with ML
- List of potential applications in econ

Machine Learning: An Applied Econometric Approach

Sendhil Mullainathan and Jann Spiess

Machines are increasingly doing "intelligent" things: Facebook recognizes faces in photos, Siri understands voices, and Google translates websites. The fundamental insight behind these breakthroughs is as much statistical as computational. Machine intelligence became possible once researchers stopped approaching intelligence tasks procedurally and began tackling them empirically. Face recognition algorithms, for example, do not consist of hard-wired rules to scan for certain pixel combinations, based on human understanding of what constitutes a face. Instead, these algorithms use a large dataset of photos labeled as having a face or not to estimate a function $f(x)$ that predicts the presence y of a face from pixels x. This similarity to econometrics raises questions: Are these algorithms merely applying standard techniques to novel and large datasets? If there are fundamentally new empirical tools, how do they fit with what we know? As empirical economists, how can we use them?[1]

We present a way of thinking about machine learning that gives it its own place in the econometric toolbox. Central to our understanding is that machine learning

Supervised vs un-supervised learning

- **supervised learning**
  - Training data contains labeled examples of the task to solve
  - Model generalizes this to unseen data
  - Example: Regression, classification
- **unsupervised learning**
  - Training with label free data
  - Model finds patterns in data
  - Example: Clustering, dimensionality reduction
- **self-supervised learning**: Middle ground we discuss later

Terminology

- **feature, attribute**: x-variable, independent variable
- **target**: y-variable, dependent variable
- **model**: estimator
- **fitting**: running an estimation
- **classification**: regression with discrete dependent variable
- **logistic regression**: binary or multivariate logit
- **instance**: observation
- **classes**: possible values of a discrete dependent variables

Loading Datasets from Sklearn

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> iris.keys()
dict_keys(['data', 'target', 'frame', \
'target_names', 'DESCR', 'feature_names',\
'filename',  'data_module'
])
```

```
>>> iris["data"].shape
(150, 4)
```

```
>>> iris["feature_names"]
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

```
>>> iris["target_names"]
array(['setosa', 'versicolor', 'virginica'])
```

- `sklearn.datasets` contains functions to load different datasets
- The functions return a dictionary-like object
- `"data"` and `"target"` attributes store the features and target data as arrays
- Optionally, can get the `"data"` attribute as `DataFrame`

Task 1

(5 min)

Overfitting

- Estimating large models on small datasets can lead to overfitting
- Overfitting means:
  - Model can explain the concrete dataset well
  - Model would not work on any other dataset
- Reason why we need adjusted R2 in econometrics
- Example: match statistics in soccer
- Need to make sure our model evaluation detects overfitting!

Holdout samples

- Split data into training and test dataset
- Fitting and experimentation is only done on training data
- Evaluation is only done on test data
- Need to avoid leaking any information from test data into training data!
- Typical sizes:
  - 70 to 80 percent for training
  - Rest for validation

Splitting into Train and Test Sets

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...   iris["data"],
...   iris["target"],
...   random_state=1234,
...   test_size=0.3,
... )
>>> X_train.shape
```

```
(105, 4)
```

```
>>> y_train.shape
(105, )
```

```
>>> X_test.shape
(45, 4)
```

```
>>> y_test.shape
(45, )
```

- The function `train_test_split` lets you:
  - select the test set size
  - set `random_state` for reproducibility

Task 2

(5 min)

scikit-learn

- scikit-learn = sklearn
- de-facto standard python library for machine learning (not deep learning)
- methods for supervised and unsupervised machine learning
- very beginner friendly with extensive documentation

Sklearn Design Philosophy

- **Consistency**: All objects have:
  - same limited set of methods
  - consistent documentation.
- **Inspection**: All specified parameter values are exposed as public attributes.
- **Limited object hierarchy**:
  - Algorithms are custom classes
  - Datasets are standard types (Numpy arrays, DataFrames, ...)
- **Composition**: Compose complex algorithms out of simple ones
- **Sensible defaults**: Get a reasonable baseline without specifying many arguments

Basic Sklearn Steps

- Choose a class of model by importing the appropriate estimator
- Set hyperparameters by instantiating this class
- Arrange data into a features matrix and target vector
- Fit the model to your data by calling the fit() on the model instance
- Apply the Model to new data using the predict method
- Evaluate the quality of predictions

Running Logistic Regression in Sklearn

```
>>> from sklearn.linear_model import LogisticRegression
>>> model = LogisticRegression(
...   fit_intercept=True,
...   penalty=None,
... )
>>> model.fit(X_train, y_train)
>>> y_pred = model.predict(X_test)
```

```
>>> y_pred
array([1, 1, 2, 0, 1, 0, 0, 0, 1, 2, 1, 0, 2, 1, 0, 1,\
2, 0, 2, 1, 1, 1,1, 1, 2, 0, 2, 1, 2, 0, 1, 2, 0, 1, 1,\
0, 0, 0, 0, 1, 0, 1, 0, 2,2])
```

```
>>> model.score(X_test, y_test)
0.977777
```

- Use the `LogisticRegression` classifier from `sklearn` to create the `model` object
- Fit the model to the *training* set to estimate the parameters
- Use the `predict` method to generate predictions
- Use the `score` method on the *test* set to assess model quality

Task 3

(7 min)

Accuracy Score

$$Accuracy = \frac{1}{N} \sum_{i}^{N} \mathbb{I}(y_i = \hat{y}_i)$$

```
>>> from sklearn.metrics import accuracy_score
>>> accuracy_score(y_test, y_pred)
0.977777
```

- Measures the share of correctly predicted data points
- Advantage: Just one number
- Disadvantage: Might not be what you care about

Accuracy with imbalanced data

- **imbalanced data**: Some outcomes occur more frequent that others in the data
- Example: Predicting wether someone has a PhD in a classroom with 49 students and one teacher
- Models can "cheat" by predicting majority outcome
- Accurracy would be 98 % but model did not learn anything
- Will need other scores to discover such problems

The Confusion Matrix

```
>>> from sklearn.metrics import confusion_matrix
>>> import pandas as pd
>>> confusion = confusion_matrix(
...     y_test, y_pred, normalize="true"
... )
```

```
>>> confusion = pd.DataFrame(
...   confusion,
...   columns=data["target_names"],
...   index=data
...   ["target_names"]
... )
>>> print(confusion)
             Cat1        Cat2       Cat3
Cat1          1.0        0.00       0.00
Cat2          0.0        0.95       0.05
Cat3          0.0        0.20       0.80
```

- Rows are the true labels
- Columns are the predictions
- The rows always sum to 1
- Diagonal elements show the share of correctly classified examples in each category
- Element [cat2, cat3]: 5 % of observations with true label cat2 got missclassified as cat3

A note on the different scores

- Think of scores as different summaries of the confusion matrix
- Scores are first calculated for each category
- An aggregation strategy converts them into one score for the entire model
- Only some aggregation strategies work for imbalanced data

Precision Score

```
>>> from sklearn.metrics import precision_score
>>> precision_score(y_test, y_pred, average=None)
array([1., 0.94444444, 1.])
```

$$Precision_k = \frac{TP_k}{TP_k + FP_k}$$

- For each class, measures the probability of the predicted positive case actually being positive
- FP_k (*false positive*) is the total number of examples classified as label k, but actually from a different class
- Preferred metric when false positive predictions are costly

Recall Score

```
>>> from sklearn.metrics import recall_score
>>> recall_score(y_test, y_pred, average=None)
array([1., 1., 0.91666667])
```

$$Recall_k = \frac{TP_k}{TP_k + FN_k}$$

- For each class, measures the model's ability to find the positive cases
- FN_k (*false negative*) is the total number of examples actually from class k that were not predicted by the model as such

F1 Score

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_test, y_pred, average=None)
array([1., 0.97142857, 0.95652174])
```

$$F_{1k} = 2\frac{Precision_k * Recall_k}{Precision_k + Recall_k}$$

- $F1$ score is the *harmonic mean* of precision and recall
- For a given class, there is a trade-off in precision and recall
- $F1$ balances the two motives
- Good choice if you have no reason to penalize one error more than another

Summary

- Accuracy: share of correct predictions
- Precision: True positives over positive predictions
- Recall: True positives over actual positives
- F1: Harmonic mean of Precision and Recall

Scores with imbalanced data

- Same example with 49 students and one teacher
- Models can "cheat" by predicting majority outcome
  - Accurracy: 98 %
  - Precision: 98 % for majority, 0 for minority class
  - Recall: 100 % for majority, 0 for minority class
  - F1: 99 % for majority, 0 for minority class
- If we just look at scores for majority, we don’t see problems
- Unfortunately that is what you get by default in sklearn in the binary case

Aggregation Strategies

```
>>> precision_score(
...   y_test,
...   y_pred,
...   average="macro"
... )
0.9815
```

```
>>> precision_score(
...   y_test,
...   y_pred,
...   average="weighted"
... )
0.9790
```

- $Prc^{macro} = \frac{1}{K}\sum_{1}^{K} Prc_k$
- $Prc^{weighted} = \sum_{1}^{K} w_k Prc_k$

- `"macro"` strategy takes the simple mean over scores for each class:
- `"weighted"` strategy weights the scores by the relative sizes of the classes
- Aggregate F_1 score is the harmonic mean of the aggregate precision and recall

Sklearn's Classification Report

```
>>> from sklearn.metrics import classification_report
>>> report = classification_report(
...   y_test,
...   y_pred,
...   target_names=iris["target_names"],
... )
... print(report)
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        16
  versicolor       0.94      1.00      0.97        17
   virginica       1.00      0.92      0.96        12

    accuracy                           0.98        45
   macro avg       0.98      0.97      0.98        45
weighted avg       0.98      0.98      0.98        45
```

Report with imbalanced data

```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99        49
           1       0.00      0.00      0.00         1

    accuracy                           0.98        50
   macro avg       0.49      0.50      0.49        50
weighted avg       0.96      0.98      0.97        50
```

Task 4

(15 min)

The bias variance trade-off

- Econometrics: Model is correctly specified, want consistency and unbiasedness
- ML: Model is a simplification and some amount of bias is ok
- Very simple models, e.g. just an intercept
  - Large bias, Low variance, No overfitting
- Very large models, e.g. including squares, interactions, ...
  - Small bias, High variance, Danger of overfitting
- Most machine learning models have one or more parameters that govern the bias variance trade-off

Example: Penalty in a logit model

- Logistic regression is fitted my minimizing a negative log likelihood function
- Can augment likelihood by a term that penalizes model complexity
- Typically, model complexity means many non-zero parameters
- Penalty is a function of the parameter vector

$$\theta^* = argmin_\theta - l(\theta, X, y) + \lambda p(\theta)$$

Different penalties

- l1: $(\theta) = \sum \theta_i$
  - Penalizes all deviations from zero equally
  - Induces sparsity
  - Harder numerical optimization, not compatible with all optimizers
- l2: $p(\theta) = \sum \theta_i^2$
  - Penalizes values close to zero very weakly
  - Does not induce sparsity
  - Simpler numerical optimization

Task 5

(5 min)

Task 6

(25 min)

Two splits are not enough

- Want to set tuning parameters optimally
- Naive approach:
  - Fit models with different parameters on training set
  - Evaluate performance on test set
  - Keep the best
- Problem: Hyperparameters are over-fit to the test set
- Cross validation avoids this

K-fold cross validation

- Idea: Split the training data repeatedly into:
  - Data used for actual traning
  - Data used for evaluation
- Repeat k times to get k scores
- Keep model that achieves best average score
- Use actual test set only once in the end to measure model quality

Cross - Validation

```
>>> from sklearn.model_selection import cross_val_score
>>> scores = cross_val_score(
...   LogisticRegression(max_iter=3000),
...   X_train,
...   y_train,
...   cv=5
... )
>>> scores
array([0.96667, 1., 0.93333, 0.96667, 1.])
```

```
>>> scores.mean()
0.973333
```

- Import and create instance as normal
- Do not call fit
- Provide data to `cross_val_score`
- `cross_val_score` will call fit

Task 7

(5 min)

Systematic hyperparameter tuning

- Specify a combination of hyperparameters we want to try
- Calculate cross validation score for each set of parameters
- Keep model with best performance
- Re-fit best model on entire dataset
- Implement in `GridSearchCV`

Grid Search

```
>>> from sklearn.model_selection import GridSearchCV
>>> param_grid = {
...   "penalty": ["l2", "l1"],
...   "C": [1, 10, 100],
... }
>>> grid = GridSearchCV(
...   LogisticRegression(),
...   param_grid,
...   cv=7,
... )
>>> grid.fit(X_train, y_train)
```

```
>>> grid.best_params_
{'C': 100, 'penalty': 'l2'}
```

```
>>> grid.best_estimator_.score(
...   X_test,
...   y_test
... )
0.930763
```

- `param_grid` keys are names of arguments of `LogisticRegression`
- `param_grid` values are lists of possible values for the arguments
- Setting up the grid does not fit models yet
- `grid.fit()` takes some time and often produces warnings

Task 8

(15 min)