COMP 204: Python programming for life sciences

Intro to machine learning with scikit-learn

Part 2

Yue Li

based on on slides from Mathieu Blanchhette and Christopher J.F. Cameron

Course evaluation and Assignment 4

Please complete the course evaluation on Minerva.

Numerical results will be available to you if there are enough evaluations (35%)

Assignment 4 is due next Friday (March 29, 23:59)

Assignment 5 will be posted on the same day Assignment 4 is due and the due date will be in two weeks

Outline

Pandas

Pandas is a Python package that allows for easy handling of 1D and 2D tabular data. Very convenient for:

- Read tabular data from file
- Basic manipulation of tabular data:
  - Row/column selection
  - Basic statistics
  - Row/column insertion/deletion
- API `http://pandas.pydata.org/pandas-docs/stable/`
- tutorials `https://pandas.pydata.org/pandas-docs/stable/tutorials.html`

Do I need to know Pandas for the exam?

- No, but you need to be able to use it if I provide you with the appropriate documentation (API).

Scikit-learn

Scikit-learn is a Python package that implements a variety of classification and regression machine learning algorithms.

- API: http://scikit-learn.org/stable/modules/classes.html
- tutorials: http://scikit-learn.org/stable/

Do I need to know Scikit-learn for the exam?
No, but you need to be able to use it if I provide you with the appropriate documentation (API).

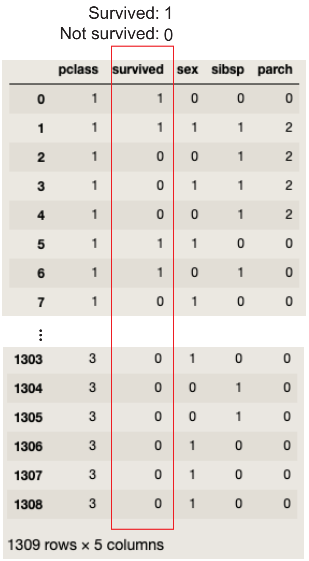
Reminder - Types of supervised learning tasks

Three general types of prediction tasks:

1. **classification**: the goal is to predict which of a predefined set of classes an example belongs to
   - digit recognition: 0 or 1 or 2 or 3 or 4... ?
   - Survivor or non-survivor from Titanic?
   - Cancer vs normal?
2. **regression**: goal is to predict a real value
   - What is the transcription factor binding affinity?
   - What's the BMI of a person based on his/her genotype?
   - What's risk for developing Alzheimer's disease for a given mutation?

Outline

Classifying survivor and non-survivor from Titanic



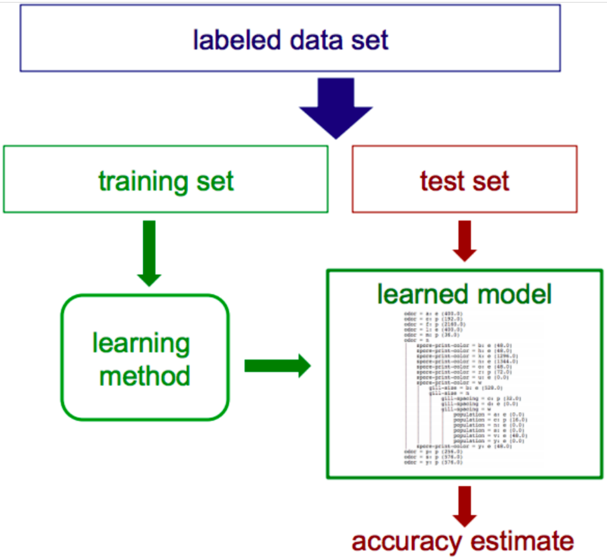
Survived: 1
Not survived: 0

	pclass	survived	sex	sibsp	parch
0	1	1	0	0	0
1	1	1	1	1	2
2	1	0	0	1	2
3	1	0	1	1	2
4	1	0	0	1	2
5	1	1	1	0	0
6	1	1	0	1	0
7	1	0	1	0	0
⋮					
1303	3	0	1	0	0
1304	3	0	0	1	0
1305	3	0	0	1	0
1306	3	0	1	0	0
1307	3	0	1	0	0
1308	3	0	1	0	0

1309 rows × 5 columns

- Goal: For a given passenger, we want to predict whether he or she survive using the following input variables:
  - pclass (passenger class),
  - sex (sex of passenger: male or female)
  - sibsp (number of siblings/spouses aboard)
  - parch (number of parents or children aboard)
- Objective function (cross-entropy error): $E = \sum_i -y_i \log p_i - (1-y_i)\log(1-p_i)$
- Algorithm: logistic regression $p_{survived} = \sigma(w_{pclass}x_{pclass} + w_{sex}x_{sex} + w_{sibsp}x_{sibsp} + w_{parch}x_{parch})$
- where $\sigma(z) = 1/(1+\exp(-z))$
- $w \leftarrow \arg\min_w E$

Model evaluation



```
odor = a: e (400.0)
odor = c: p (192.0)
odor = f: p (2160.0)
odor = l: e (400.0)
odor = m: p (36.0)
odor = n
|   spore-print-color = b: e (48.0)
|   spore-print-color = h: e (48.0)
|   spore-print-color = k: e (1296.0)
|   spore-print-color = n: e (1344.0)
|   spore-print-color = o: e (48.0)
|   spore-print-color = r: p (72.0)
|   spore-print-color = u: e (0.0)
|   spore-print-color = w
|   |   gill-size = b: e (528.0)
|   |   gill-size = n
|   |   |   gill-spacing = c: p (32.0)
|   |   |   gill-spacing = d: e (0.0)
|   |   |   gill-spacing = w
|   |   |   |   population = a: e (0.0)
|   |   |   |   population = c: p (16.0)
|   |   |   |   population = n: e (0.0)
|   |   |   |   population = s: e (0.0)
|   |   |   |   population = v: e (48.0)
|   |   |   |   population = y: e (0.0)
|   spore-print-color = y: e (48.0)
odor = p: p (256.0)
odor = s: p (576.0)
odor = y: p (576.0)
```

Split the dataset into training and testing datasets

We split the data into 80% training and 20% testing

```
from sklearn import model_selection

X = data.drop(["survived"], axis=1).values
y = data["survived"].values
results = model_selection.train_test_split(X, y,
   test_size = 0.2, shuffle = True)
X_train, X_test, y_train, y_test = results
```

Logistic regression prediction

```
logitreg = linear_model.LogisticRegression()
fit = logitreg.fit(X_train, y_train)
effect_size = pd.DataFrame(fit.coef_)
effect_size.columns = data.drop(["survived"],
↪   axis=1).columns
print(effect_size)
# pclass        sex       sibsp     parch
# -0.78978 -2.514111 -0.205692  0.059857
y_train_pred = fit.predict(X_train)
y_test_pred = fit.predict(X_test)
```

- We train logistic regression on the training data: `logitreg.fit(X_train, y_train)`
- We then apply the trained model `fit` to predict survivor: `y_train_pred=fit.predict(X_train)`, `y_test_pred=fit.predict(X_test)`
- Our prediction is binary 0 (not survived) or 1 (survived) based on whether the predicted probabilities are greater than 0.5

Classification Accuracy

```python
# accuracy = correctly classified / total classified
acc_train = sum(y_train_pred==y_train)/len(y_train)
acc_test = sum(y_test_pred==y_test)/len(y_test)
print(f"train accuracy: {acc_train:.3f}; \
test accuracy: {acc_test:.3f}")
# train accuracy: 0.793; test accuracy: 0.779
```

- We then evaluate the prediction accuracy:

$$Accuracy = \frac{\text{Correctly classified passengers}}{\text{Total number of classified passengers}}$$

- As what we expect, the accuracy for predicting survivors in the training dataset (79.3%) is slightly higher than the accuracy in predicting survivors in the testing dataset (77.9%)
- But no obvious overfitting in our model

Outline

Classification threshold: how to decide who is a survivor?

pclass	sex	sibsp	parch	**pred_prob**	true_label
3.0	1.0	1.0	0.0	0.095376	0.0
1.0	1.0	0.0	0.0	0.414317	1.0
2.0	0.0	0.0	0.0	0.771990	1.0
3.0	1.0	1.0	0.0	0.095376	1.0
3.0	1.0	1.0	0.0	0.095376	0.0
1.0	0.0	0.0	1.0	0.892909	1.0

- pred_prob: predicted probabilities for passenger to survive
- By default, `fit.predict(X_test)` use 0.5 as threshold, i.e.,

```
if pred_prob > 0.5:
    survived = 1
else:
    survived = 0
```

- What accuracy do we get with a different threshold say 0.6?
- Can we evaluate the model without setting arbitrary threshold?

True and false positive rates

At a specific threshold, we can calculate TPR and FPR:

True positive rate (TPR) (aka sensitivity)

The proportion of positive examples that are predicted positive

- Fraction of survivors who are predicted to survive

$$TPR = \frac{TP}{TP + FN}$$

False positive rate (FPR)

The proportion of negative examples that are predicted to be positive

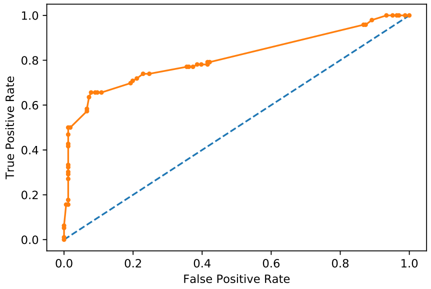
- Fraction of non-survivors who are predicted to survive

$$FPR = \frac{FP}{FP + TN}$$

Receiver Operating Characteristic (ROC) curve

- We can create a table for TPR and FPR at each Threshold.
- ROC curve plots TPR (y-axis) versus FPR (x-axis)
- *Area under the curve (AUC)* is a metric common used to evaluate the model. In our case, the AUC is equal to 80.9%

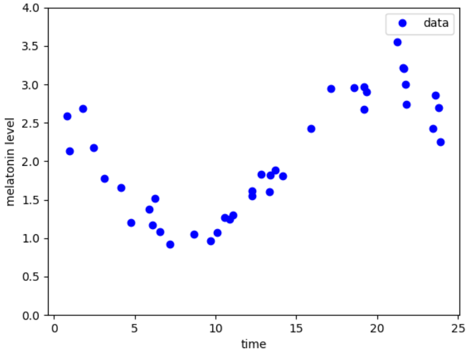
TPR	FPR	Threshold
0.000000	0.000000	1.898057
0.010417	0.000000	0.898057
0.052083	0.000000	0.892909
0.062500	0.000000	0.888421
0.156250	0.006024	0.887533
0.156250	0.012048	0.877039
...	...	...
0.656250	0.090361	0.448929
0.656250	0.096386	0.441242
...	...	...
0.770833	0.361446	0.230475
0.770833	0.373494	0.215268
0.781250	0.385542	0.206121
0.781250	0.397590	0.197264
...	...	...
1.000000	1.000000	0.026146



Outline

A regression problem

Background: Melatonin (sleep hormone) levels vary over time in a cyclical manner.
Data: We have measured the patient's melatonin levels at different times.
Goal: Learn to predict a patient's melatonin level as a function of time, e.g. to choose when to deliver a drug

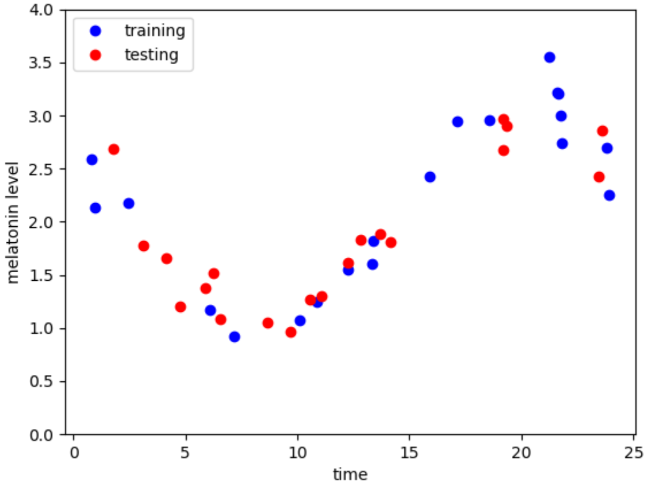


Splitting training and test sets

Assuming X is a numpy array containing times of measurements, and y is a numpy array containing melatonin levels.
In ML, we always want to split the data into two non-overlapping sets: training set and test set.
Here, we use 50% of the examples for the training, and 50% for the testing.
Note: Often a larger fraction of the data is used for training (e.g. 80% training, 20% testing).

```
from sklearn import model_selection
# split data into training and test datasets
X_train, X_test, y_train, y_test = \
    model_selection.train_test_split(X, y, test_size =
     ↪  0.5, shuffle = True)
```

Splitting training and test sets



Regression problem

- Problem: Let the melatonin level be y and time be x
- Goal: Learn a function $f(x)$ to predict y values from x values
- Objective function: sum of square errors

$$E = \sum_{i \in train} (f(x_i) - y_i)^2$$

- Algorithm: We will treat $f(x)$ as a *polynomial*:
- Let's start with a polynomial of degree 1:

$$f(x) = ax + b$$

- The goal of learning is to choose the value of coefficients a and b based on training data.
- We want to choose a and b so as to best fit the training data that *minimizes* the sum of square error

$$a, b \leftarrow \arg\min_{a,b} E$$

Regression with scikit-learn

To learn a regression using scikit-learn:

```python
# transform data into matrices for regression
reg_X_train = X_train[:,np.newaxis]
reg_X_test = X_test[:,np.newaxis]
```

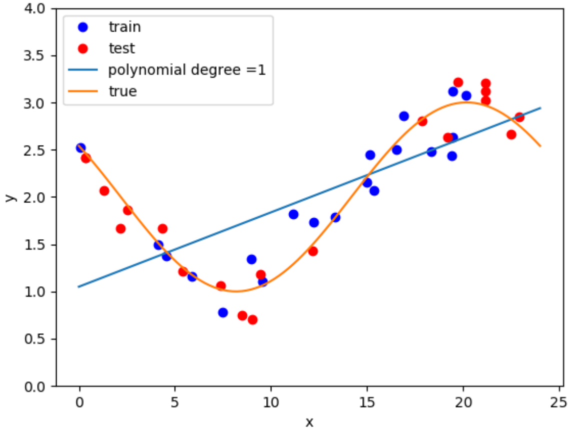
```python
# Create a polynomial regression model
model = make_pipeline(PolynomialFeatures(degree),
↪  Ridge(0))

# Fit the model to the training data
model.fit(reg_X_train, y_train)

# Apply the model to make predictions on the training data
pred_train =  model.predict(reg_X_train)

# Apply the model to make predictions on the test data
pred_test =  model.predict(reg_X_test)

# Calculate mean squared errors
train_err = mean_squared_error(y_train,pred_train)
test_err = mean_squared_error(y_test,pred_test)
```

1-degree polynomial regression

For our data, the best choice is $a = 1.4$, $b = 0.9$ i.e.

$$y = 1.4x + 0.9$$

Just by eyeballing, we can tell that the fit is not good.



Mean Squared error and Underfitting

Problem: Just by looking at the plot we can tell that the fit to the training data is very bad:
The fitted line is far from the observed values at most training examples.

Measuring prediction errors:
Mean-squared-error = Sum of the squares of the difference between the predicted and observed values divided by the total number of the training examples:

$$MSE(train) = \frac{\sum_{i \in train} (f(x_i) - y_i)^2}{N_{train}}$$

Here: MSE(train) = 0.31 and MSE(test) = 0.50

When the training error is too large, we call this *underfitting*: The predictor cannot fit the training data well because it is too limited in the type of functions it can represent.

Quadratic regression (degree $= 2$)

We can improve the fit to the training data by considering a polynomial of degree 2:

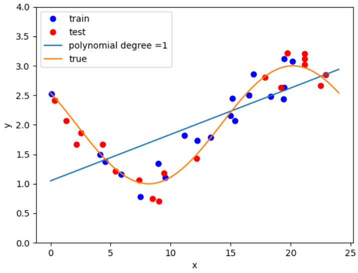
$$f(x) = ax^2 + bx + c$$

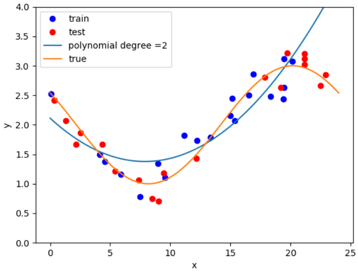
All we need to do is: degree $= 2$

```
# Create a polynomial regression model
model = make_pipeline(PolynomialFeatures(degree),
↪  Ridge(0))

# Fit the model to the training data
model.fit(reg_X_train, y_train)
```

Some improvement with 2-degree polynomial

The fit using degree = 2 is a bit better:

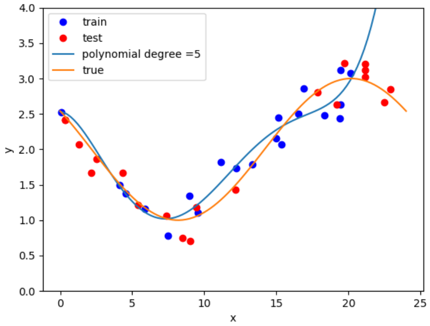




- 1-degree: MSE(train) = 0.31 and MSE(test) = 0.50
- 2-degree: MSE(train) = **0.099**, MSE(test) =**0.23** (somehow testing error is slightly higher than 1-degree)

Higher-degree polynomial

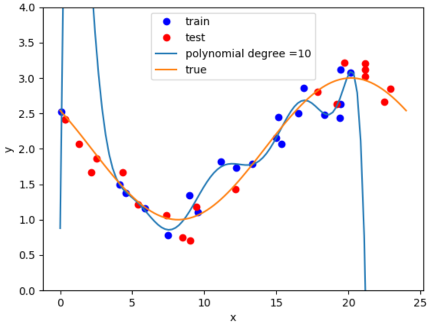
We can further improve the fit to the training data by considering higher degree polynomial, e.g. degree = 5
$f(x) = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$



- 1-degree: MSE(train) = 0.31 and MSE(test) = 0.50
- 2-degree: MSE(train) = 0.099, MSE(test) =0.23
- 5-degree: MSE(train) = **0.022**, MSE(test) =**0.057**

And even higher-degree polynomial

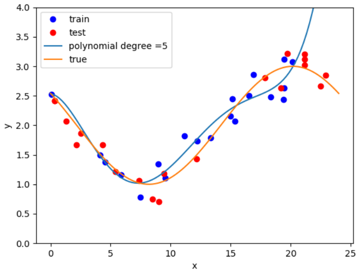
Let's see if we keep going to higher degrees: degree = 10

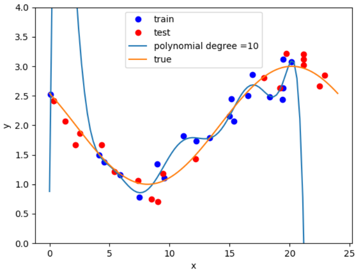


- 1-degree: MSE(train) = 0.442 and MSE(test) = 0.545
- 2-degree: MSE(train) = 0.42, MSE(test) =0.598
- 5-degree: MSE(train) = 0.028, MSE(test) =**0.066**
- 10-degree: MSE(train) = **0.01**, MSE(test) =0.40

Overfitting (recap)

If the number of parameters to learn is large (e.g. for a polynomial of degree 10, there are 11 parameters), the predictor is able to fit the training data very well: MSE(train) is very small.
But the corresponding testing error MSE(test) is very large!
This is *bad*, because our goal is for our predictor to do well on the test data (i.e. data it hasn't seen during its training).





Overfitting (recap)

If the number of parameters to learn is large (e.g. for a polynomial of degree 10, there are 11 parameters), the predictor is able to fit the training data very well: MSE(train) is very small.
But the corresponding testing error MSE(test) is very large!
This is *bad*, because our goal is for our predictor to do well on the test data (i.e. data it hasn't seen during its training).

This is called *overfitting*:Predictor is able to fit the training data very well, but fits testing data very poorly:

$$MSE(train) << MSE(test).$$

Overfitting happens when the predictor has too much flexibility in choose the values of too many parameters.
To limit overfitting, we have to limit the number of parameters the predictor has to estimate (or use other means such as regularization).