COMP 204: Python programming for life sciences

Intro to machine learning with scikit-learn

Part 2

Yue Li

based on on slides from Mathieu Blanchette and Christopher J.F. Cameron
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/

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

- **tutorials:** [http://scikit-learn.org/stable/](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
   - Cat vs Bird?
   - Cancer vs normal?
   - digit recognition: 0 or 1 or 2 or 3 or 4... ?

2. **regression**: goal is to predict a real value
   - What will the price of oil be tomorrow?
   - How fast will this tumour grow?

3. **probability estimation**: goal is to estimate a probability
   - will it rain tomorrow?
   - will this drug be effective on this patient?
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).

```python
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

![Graph showing melatonin level over time for training and testing sets.](image)
Regression problem

Goal: Learn a function $f(x)$ to predict $y$ values from $x$ values

Here, we will assume that $f(x)$ is 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.
For our data, the best choice is $a = 0.9$, $b = 1.4$: 
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]

# 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)
```
Mean Squared error and Underfitting

**Problem:** The fit to the training data is very bad: The polynomial 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:

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

Here: $MSE(train) = 0.442$ and $MSE(test) = 0.545$

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

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

```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)
```

The fit is a bit better:

![Plot of quadratic regression fit]
Higher-degree polynomial

We can further improve the fit to the training data by considering higher degree polynomial, e.g. degree = 5

\[ f(x) = ax^5 + bx^4 + cx^3 + dx^2 + ex + f \]

All we need to do is: degree = 5

```python
model = make_pipeline(PolynomialFeatures(degree), Ridge(0))

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

MSE(train) = 0.028, MSE(test) = 0.066
And even higher-degree polynomial

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

\[
\text{MSE}(\text{train}) = 0.01, \quad \text{MSE}(\text{test}) = 8.56
\]
### Training vs Testing Errors

<table>
<thead>
<tr>
<th>Polynomial degree</th>
<th>Training error</th>
<th>Testing error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.442</td>
<td>0.545</td>
</tr>
<tr>
<td>2</td>
<td>0.09</td>
<td>0.218</td>
</tr>
<tr>
<td>5</td>
<td>0.028</td>
<td>0.516</td>
</tr>
<tr>
<td>10</td>
<td>0.010</td>
<td>484.566</td>
</tr>
</tbody>
</table>
Overfitting

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: $\text{MSE}(\text{train})$ is very small. But the corresponding testing error $\text{MSE}(\text{test})$ 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:

$$\text{MSE}(\text{train}) \ll \text{MSE}(\text{test}).$$

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).
Until now, our goal was to learn $f(x)$, where $x$ was a single variable. In general, we may want to generalize this to multiple variables $x_1, x_2, \ldots, x_n$: For example, with $n = 2$ variables, we could choose the polynomial:

$$f(x_1, x_2) = ax_1^2 + bx_2^2 + cx_1x_2 + dx_1 + ex_2 + f$$

Here, each training example is a pair of numbers $x_1, x_2$. The goal remains the same: choose the coefficients to minimize the training/testing error.
A two-dimensional classification problem

Suppose you want to learn to predict if a person has a prostate cancer based on two easily-measured variables obtained from blood sample: Complete Blood Count (CBC) and Prostate-specific antigen (PSA). We have collected data from patients known to have or not have prostate cancer:

<table>
<thead>
<tr>
<th>CBC</th>
<th>PSA</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>142</td>
<td>67</td>
<td>Normal</td>
</tr>
<tr>
<td>132</td>
<td>58</td>
<td>Normal</td>
</tr>
<tr>
<td>178</td>
<td>69</td>
<td>Cancer</td>
</tr>
<tr>
<td>188</td>
<td>46</td>
<td>Normal</td>
</tr>
<tr>
<td>183</td>
<td>68</td>
<td>Cancer</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A two-dimensional classification problem
Training objective

Goal: Learn classification function \( f(x_1, x_2) \), where

- \( x_1 = \text{CBC} \)
- \( x_2 = \text{PSA} \)
- \( f(x_1, x_2) < 0 \) if normal
- \( f(x_1, x_2) > 0 \) if cancer
A perfect classifier

\[ f(\text{CBC}, \text{PSA}) = 0.8 \times \text{CBC} - \text{PSA} - 20 \]

\[ f(\text{CBC}, \text{PSA}) > 0 \implies \text{Cancer} \]

\[ f(\text{CBC}, \text{PSA}) < 0 \implies \text{Normal} \]
Support vector machines (SVM)

Support vector machines are popular classifiers for high-dimensional data. In their simplest form, they try to find a straight line that separates the positive (cancer) and negative (normal) examples.

SVMs try to maximize the margin between the predictors’ decision boundary and the nearest positive/negative examples.
A two-dimensional classification problem

\[ f(CBC, PSA) > 0 \implies \text{Cancer} \]
\[ f(CBC, PSA) < 0 \implies \text{Normal} \]

\[ f(CBC, PSA) = 0.8 \times CBC - PSA - 20 \]
SVMs in scikit-learn

scikit-learn SVM classifier API

```python
from sklearn import model_selection, svm

# Create a SVM classifier
clf = svm.SVC()

# fit it to the training data
clf.fit(X_train, y_train)

# apply the trained predictor to the training and testing data
pred_train = clf.predict(X_train)
pred_test = clf.predict(X_test)
print(pred_test)

# calculate training and testing error
train_err = mean_squared_error(y_train, pred_train)
test_err = mean_squared_error(y_test, pred_test)
print("train_error:", train_err, "\ntest_error:", test_err)
```
A harder classification problem

Here, no linear decision boundary can perfectly separate the positive and negative example. What to do? Find the boundary that minimizes the number of classification errors.