A(E)3M33UI — Exercise D: Basis expansion for linear and logistic regression

Petr Pošík

March 13, 2017

Goals of this exercise:

- make linear models non-linear using basis expansion
- build a simple custom operator in `scikit-learn`
- construct pipelines in `scikit-learn`
- demonstrate that more flexible models achieve lower error rates on the training data
- show that different predictive models (in this case linear regression, logistic regression, and support vector machine) have the same interface in `scikit-learn`.

The program code for this exercise is organized in several Python modules:

- `exD-1.py` – main script for the **first** part (regression models)
- `exD-2.py` – main script for the **second** part (classification models)
- `mpg.py` – helper functions for loading data
- `plotting.py` – helper functions for graphical display of the data and models,
- `mapping.py` – functions for basis expansion,
- `model_evaluation.py` – functions for computing model errors.

After completion, zip all the above files and hand in the archive via the Upload system.
If you will not manage to complete the exercise in the lab, finish it as a homework!

1 Non-linear regression models using linear regression and basis expansion

As in the previous exercise, we work with the `auto-mpg.csv` dataset, and study the relation of horse power and displacement, i.e. we will build the model $\widehat{hp} = h(disp)$.

Run the `exD-1.py` script. It should plot the data and end up in an error.

Task 1: In module `plotting.py`, fill in the missing code in `plot_1D_regr_model()`, so that the example script will show the predictions of the model given as a parameter.

Hints: Assume that the `model` argument is a `sklearn` predictive model, i.e. that it has a `predict()` method. There is nothing new in this task, you have done this already in the last exercise.

Now, we can see the predictions of the linear model in a figure. We compute the mean squared error of this model on the training data.

Task 2: In `model_evaluation.py`, fill in the function `compute_model_error(model,X,y,err_func)`, which takes a trained `model`, the training data `X`, `y`, and the error function `err_func`, and produces the error of the model.

Hints:

- Look at the function `compute_err_MSE()` in `model_evaluation.py`. You should be familiar with it, we implemented it last week.
- Make the function `compute_model_error` universal in such a way that it can compute the error of any predictive model, as long as a suitable `err_func` is provided by the user.
- In `exD-1.py`, we supply the `compute_err_MSE` function to the `compute_model_error` as the error function (`err_func`) argument. Later in this exercise, we will provide a different error function for classification models.

1.1 Basis expansion: polynomials

In this part of the exercise, we implement the basis expansion as a transformation usable in the `scikit-learn` pipeline, i.e. it must implement the relevant APIs (functions: `fit()` and `transform()`.

In this section, we implement a simple polynomial mapping that transforms the data points from the feature space X to an image space containing $X, X^2, \ldots, X^{\texttt{max\_deg}}$, where the `max_deg` is the maximal degree of the polynom.

Task 3: In `mapping.py`, implement the class `PolynomialMapping`:

- In case of transformations, the `fit()` method is used to train the transformation in an unsupervised way (i.e. it may be useful e.g. for PCA, ICA, k-means clustering, etc.). In our case, the transformation is fixed and does not depend on the data; the `fit` method will be empty, but it must return `self`, i.e. the current instance of the `PolynomialMapping` class.
- The `transform()` method performs the actual transformation (or mapping). In our case, it takes a $[m \times D]$ matrix X, and returns a matrix of size $[m \times (\texttt{max\_deg} \cdot D)]$, where the first block of D columns will be just a copy of X, the second block is X^2 (meaning a matrix of squares of items), etc.

Hints:

- You should be already able to concatenate Numpy arrays using `numpy.hstack()`.

- You can test your solution in Python shell. By issuing the commands:

  ```
  >>> pm = PolynomialMapping(2)
  >>> pm.transform(X)
  ```

  you should get a matrix with 2 columns, where the values in the second one are squares of the values in the first one.

Now, we should learn about `scikit-learn` pipelines. They allow to chain a series of preprocessing steps (transformations) with a final predictive model, making the whole sequence effectively a single larger model, which can be used as a whole. In our case, we build a pipeline of `PolynomialMapping` and `LinearRegression`.

Task 4: In `exD-1.py`, fill in the missing code to

1. create an instance of `PolynomialMapping` class with degree 2,
2. create an instance of a pipe containing the polynomial mapping and the linear regresion model (which already exists in the workspace),
3. fit the pipe to the training data,
4. plot the pipe predictions in the graph, and
5. compute the error of the pipe (quadratic model).

Once the above part is finished, it shouldn't be hard to embed it in a for-loop, so that we will see a nice comparison of the linear, quadratic, cubic and quartic polynomial models.

1.2 Comparison of polynomial models with increasing degree

Task 5: In `exD-1.py`, fill in the missing code inside the for-loop, which shall for degrees from 1 to 4 create a pipeline containing the `PolynomialMapping` instance with the respective degree and the linear regression model, fit the model, plot the model predictions in the graph and compute the error for each of the model.

If successfully finished, you should see that with increasing flexibility of the model (increasing degree), the error of the model *measured on the training data* gets smaller. This is a general phenomenon and we shall study this in the next lectures.

2 Non-linear classification models using logistic regression and basis expansion

We shall now turn our attention to the script `exD-2.py`. Run it, it will end up with an error, but you should at least see the data points with color indicating their class.

Task 6: In `exD-2.py`, fill in the code that would fit a logistic regression model to the data.

Hints:

- Take inspiration in `exD-1.py` from the case with linear regresion.
- You should just create an instance of `linear_model.LogisticRegression` and fit it to the data.

The script shall then continue by graphically showing the predictions and decision boundary of the trained model.

Task 7: In `model_evaluation.py`, implement the function `compute_err_01`, which shall return the average number of incorrect predictions.

After completion of the above task, the script shall be able to compute the error rate of the logistic regression model, and shall display it in the title of the figure.

2.1 Basis expansion: Pure polynomials

Task 8: In `exD-2.py`, fill in the code that shall build a pipe containing an instance of `PolynomialMapping` with certain `degree` and an instance of `linear_model.LogisticRegression`, fit the pipe to the data, plot the classification of the model using `plot_2D_class_model`, and compute the error rate of the model.

Hints:

- Again, take inspiration in `exD-1.py` from the case with linear regresion.
- All the building blocks are already present in the `exD-2.py` in the case for logistic regression.
- Experiment a bit with the degree of the model.

Now, you shall see another figure showing the predictions of logistic regression with non-linear decision boundary. The error rate printed in the figure title shall be a bit lower than in case of the linear prediction, although the decision boundaries do not differ much (compared by human eyes).

2.2 Basis expansion: Full polynomial mapping

Task 9: In `mapping.py`, implement the class `FullPolynomialMapping`. The `PolynomialMapping` class produces only pure polynomials like X_1^2, X_2^2, X_1^3, X_2^3, etc. It does not produce the crossproduct terms X_1X_2, $X_1^2X_2$, $X_1X_2^2$, etc. which shall be part of the `FullPolynomialMapping`. The class shall be general in the respect that it can handle matrix `X` with any number of columns and it shall work for any given maximal degree `max_deg`.

Hints:

- This task is a bit more involved. You may start by expecting that the matrix X contains 2 columns only (x and y coordinates), and you want to construct a full quadratic mapping only, i.e. include terms X_1, X_2, X_1^2, X_1X_2, and X_2^2.
- If you want to implement the class in a general way (as required), you may use the function `itertools.combinations_with_replacement`. An example of its usage:

```
>>> from itertools import combinations_with_replacement
>>> list(combinations_with_replacement([0,1,2], 1))
[(0,), (1,), (2,)]
>>> list(combinations_with_replacement([0,1,2], 2))
[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]
>>> list(combinations_with_replacement([0,1,2], 3))
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 1), (0, 1, 2),
 (0, 2, 2), (1, 1, 1), (1, 1, 2), (1, 2, 2), (2, 2, 2)]
```

Do you see how the results can be used to generate

- the linear terms X_1, X_2, X_3,
- the quadratic terms X_1^2, X_1X_2, X_1X_3, X_2^2, X_2X_3, X_3^2,
- the cubic terms X_1^3, $X_1^2X_2$, $X_1^2X_3$, $X_1X_2^2$, $X_1X_2X_3$, $X_1X_3^2$, X_2^3, $X_2^2X_3$, $X_2X_3^2$, X_3^3?

Task 10: In `exD-2.py`, fill in the code that would build a pipe containing an instance of `FullPolynomialMapping` with certain `degree` and an instance of `LogisticRegression`, fit the pipe to the data, plot the classification of the model using `plot_2D_class_model`, and compute the error rate of the model.

Hints:

- Experiment a bit with the degree of the model and observe the results.

2.3 SVM: Linear kernel

We shall now try to solve the same task by a support vector machine. For this task, we shall use the `sklearn.svm.SVC` class. This can be used to create a support vector classifier with various kinds of kernels.

Task 11: In `exD-2.py`, fill in the code that shall instantiate a `sklearn.svm.SVC` model with linear kernel, fit it to the data, plot the classification of the model using `plot_2D_class_model`, and compute the error rate of the model.

Hints:

- The default kernel used by the `SVC` model is the RBF kernel. If we want to use a linear kernel (to train the optimal separating hyperplane with soft margin), we have to specify `kernel='linear'` when constructing the model instance like this:

  ```
  model = sklearn.svm.SVC(kernel='linear')
  ```

- Observe the difference between the linear model trained by the logistic regression and the linear model trained by the SVM algorithm.

2.4 SVM: RBF kernel

Task 12: In `exD-2.py`, fill in the code that shall instantiate a `sklearn.svm.SVC` model with RBF kernel, fit it to the data, plot the classification of the model using `plot_2D_class_model`, and compute the error rate of the model.

Hints:

- The RBF kernel is the default one when constructing the instance of `SVC` class, you do not need to specify it explicitly (but you can).
- If you do not see anything interesting in the figure, try to zoom in to see the neighborhood of the data points in more detail.
- Try to set the `gamma` parameter of the SVC model to a value different from the default one, e.g.

  ```
  >>> model = sklearn.svm.SVC(gamma=0.005)
  ```

  It is the parameter of the RBF kernel and this setting changes the widths of the RBF functions.
- Experiment a bit with the `gamma` value.

3 Summary

We have seen several easy ways how to use algorithms for fitting linear models to create non-linear ones, both in the regresssion and classification setting. We have also build a custom transformation operator able to work in a pipeline with a predictive model. It was demonstrated that the error of a model measured on the training data usually gets smaller with increasing flexibility of the model. And we have also seen the support vector machine in action. *But, we still have no idea which of the models seen today shall be better for prediction!*

Finish the exercise as a homework, ask questions on the forum, and upload the solution via Upload system!