Machine Learning

CMPT 353

We'll be returning to material in the *Python Data Science Handbook*:

- Python Data Science Handbook: the book
- Python Data Science Handbook: SFU library
- Python Data Science Handbook: GitHub

Another good treatment of ML: Hands-On Machine Learning, Aurélien Géron. [Hands-On ML: SFU Library]

Compared to CMPT 419/726:

I'm less concerned about machine learning as a topic of study, and more concerned about using it as a tool to make sense of data.

Obviously, 2–3 weeks will cover a lot less detail than a whole course.

We will generally use the scikit-learn libraries which implements common ML algorithms for Python.

What is ML?

Usual parts we'll have (for "supervised" ML):

- A task: something we want to predict: calculate/estimate/discover/etc.
- Data/experience: sample inputs and corresponding correct outputs.
- A measure of success: if we make different predictions, how do we decide which one is "better"?

We want to take the sample inputs and build a ***model*** that captures the calculation that is needed to produce "correct" output.

If we have done it right, the model should be able to predict output for new inputs that we haven't seen before.

These "models" of the problem won't be completely designed by us.

We will create the overall design of the model, and ***train*** it on the known inputs/outputs.

Linear Regression

We just talked about linear regression as a statistical technique. It turns out that the machine learning people claim it too...

What we did before:

```
from scipy import stats
reg = stats.linregress(x, y)
print(reg.slope, reg.intercept)
```

```
0.523895506988829 -1.503281343279347
```

The same basic operation with scikit-learn to get the same results:

```
from sklearn.linear_model import LinearRegression
X = np.stack([x], axis=1)
```

```
model = LinearRegression(fit_intercept=True)
model.fit(X, y)
print(model.coef_[0], model.intercept_)
```

```
0.5238955069888295 -1.5032813432793506
```

Things to note...

The `np.stack` function is used to join 1D arrays (in this case, only one) into a 2D array.

```
arr = np.array([1,2,3])
print(arr)
print(np.stack([arr], axis=1))
```

```
[1 2 3]
[[1]
 [2]
 [3]]
```

In this example, `.reshape(-1, 1)` would also work.

If you already have 2D arrays, `np.concatenate` will put them together into a single 2D array.

The ML convention is to have 2D arrays (matrices) in capitalized variable names (`X` vs `x`), which isn't standard Python style.

```
X = np.stack([x], axis=1)
```

I hate it, but I'll go with it.

Another convention: scikit-learn uses trailing underscores to indicate values that were estimated/fitted/learned. *[*]*

```
print(model.coef_[0], model.intercept_)
```

Also not very Pythonic, but there it is.

Style aside, the operations we're doing are:

```
model = LinearRegression(fit_intercept=True)
model.fit(X, y)
```

1. Create a linear model. i.e. we will understand (model) the input → output predictions as a linear relationship.
2. Use the known input and output values to fit the model to the data.

Fit: compute the parameters that do the best job matching the model to the given data.

Also called ***training*** the model. The known correct inputs/outputs are the ***training data***.

For this model, the parameters are slope and intercept. "Best" is determined by the smallest sum of square error.

Once we have a model, we can use it to make predictions.

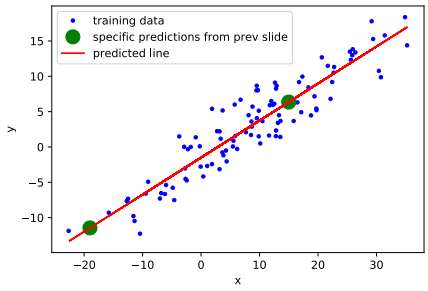
```
X_fit = [[15], [-19]]
y_fit = model.predict(X_fit)
print(y_fit)
```

```
[  6.35515126 -11.45729598]
```

i.e. when the input is `[15]`, the model predicts that the correct output is `6.35515126`.

Or we can predict the entire range and draw it:

```
plt.plot(x, y, 'b.')
plt.plot(X_fit, y_fit, 'g.', markersize=25)
plt.plot(x, model.predict(X), 'r-')
plt.legend(['training data', 'specific predictions from prev slide', 'predicted line'])
```



Summary:

- Model: the way we're understanding the problem, and making predictions.
- Parameters: values in the model that determine how it predicts.
- Fit/Train: using training data to get good values for the parameters.
- Predict: using the model to guess the correct output for new input.

The Intercept

When setting up the linear fit, we could have done the "intercept" manually like this:

```
X_with = np.concatenate([np.ones(X.shape), X], axis=1)
print(X_with[:4])
```

```
[[ 1.         29.16862815]
 [ 1.         12.8018865 ]
 [ 1.         19.74485581]
 [ 1.         34.89071839]]
```

Then we don't need the regression to worry about an intercept.

```
model = LinearRegression(fit_intercept=False)
model.fit(X_with, y)
print(model.coef_)
```

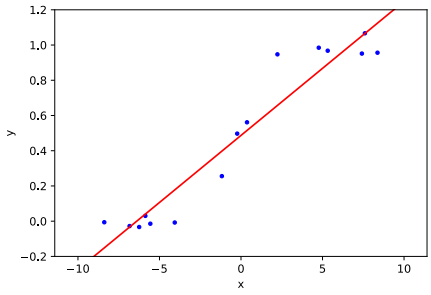
```
[-1.50328134  0.52389551]
```

Same intercept and slope. We basically just fit two a_i parameters in:

$$y = a_0 1 + a_1 x\,.$$

Polynomial Regression

What do we do when we get non-linear values, where a line won't be a good model?



Obvious answer: fit something with more freedom than just a line. We're going to need a different function with more parameters.

If we can create an array with $1 = x^0$ and $x = x^1$ values for a linear fit, we can do more of the same.

And, we don't even have to do it manually.

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=3, include_bias=True)
X = np.array([[2], [3], [4]])
print(poly.fit_transform(X))
```

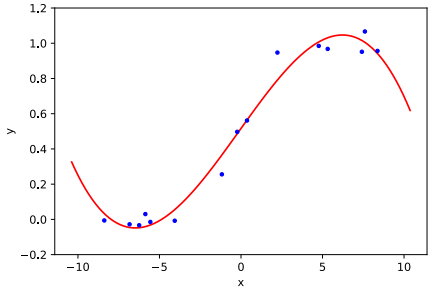
```
[[ 1.  2.  4.  8.]
 [ 1.  3.  9. 27.]
 [ 1.  4. 16. 64.]]
```

If we can do a linear fit to that input, we're finding a_i parameters to fit this to the data:

$$y = a_0 x^0 + a_1 x^1 + a_2 x^2 + a_3 x^3 \, .$$

We can use this to predict with a polynomial:

```
poly = PolynomialFeatures(degree=3, include_bias=True)
X_poly = poly.fit_transform(X)
model = LinearRegression(fit_intercept=False)
model.fit(X_poly, y)
```



What did the fit really decide?

```
print(model.coef_)
```

```
[ 5.16190050e-01  1.29319129e-01 -4.21668370e-04 -1.06965776e-03]
```

In other words, our predictions are,

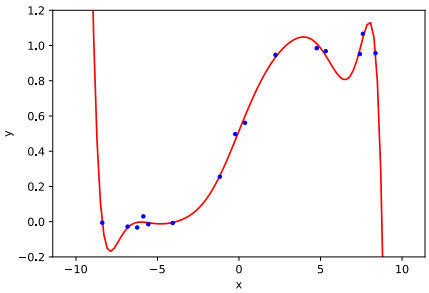
$$y_{\text{pred}} = 0.516 + 0.129x - 0.000422x^2 - 0.00107x^3 \, .$$

That seems useful: we can use higher-degree polynomial inputs and simple linear fitting to allow more flexibility.

A higher-degree polynomial allows the `.fit()` process more freedom to fit the data better.

If some is good, more must be better.

```
poly = PolynomialFeatures(degree=11, include_bias=True)
X_poly = poly.fit_transform(X)
model = LinearRegression(fit_intercept=False)
model.fit(X_poly, y)
```



The fit line does an excellent job fitting the points, but a bad job fitting what I think is going on.

What we did: take an x value and transform it to several $f_1(x), f_2(x), \ldots, f_k(x)$ values, then do a linear fit to find a_i that give the best prediction,

$$y_{\text{predicted}} = a_1 f_1(x) + a_2 f_2(x) + \cdots + a_k f_k(x) \, .$$

The f_i functions are *basis functions*.

There's nothing special about the powers (except that they're often useful).

These could have been any functions: Gaussian curves, sine/cosine curves, logarithms/exponentials, …. You can choose whatever makes sense for your data.

ML Pipelines

It's common to have a situation like this: transform the data (in one, two, or more ways). Then fit some model to the result.

That becomes cumbersome: constantly must transform the `X` values when trying to use the model.

```
poly = PolynomialFeatures(degree=3, include_bias=True)
X_poly = poly.fit_transform(X)
model = LinearRegression(fit_intercept=False)
model.fit(X_poly, y)
```

And later,

```
plt.plot(X_range[:, 0], model.predict(poly.transform(X_range)), 'r-')
```

Scikit-learn can combine several steps into a *pipeline model*:

```
from sklearn.pipeline import make_pipeline
model = make_pipeline(
    PolynomialFeatures(degree=3, include_bias=True),
    LinearRegression(fit_intercept=False)
)
```

This model takes our `X` values, creates polynomial features, and does a linear fit on them.

Then we can treat this like a single model and not explicitly worry about the steps.

```
model = make_pipeline(
    PolynomialFeatures(degree=3, include_bias=True),
    LinearRegression(fit_intercept=False)
)
model.fit(X, y)
```

```
plt.plot(X_range[:, 0], model.predict(X_range), 'r-')
```

Pipeline steps are all scikit-learn *estimator* objects.

Steps 1 to $n - 1$ will be *transformers* that can `.fit()` and `.transform()` to modify the values (or `.fit_transform()` which combines the two).

The last step is an *estimator* which can `.fit()` and `.predict()`.

It's easy to end up with several steps, and nice to not have to deal with each one every time you work with `X` values.

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.impute import SimpleImputer
model = make_pipeline(
    SimpleImputer(strategy='mean'), # impute missing values
    MinMaxScaler(),                 # scale each feature to 0-1
    PolynomialFeatures(degree=3, include_bias=True),
    LinearRegression(fit_intercept=False)
)
```

Course rule: if it can be a pipeline, it should be.

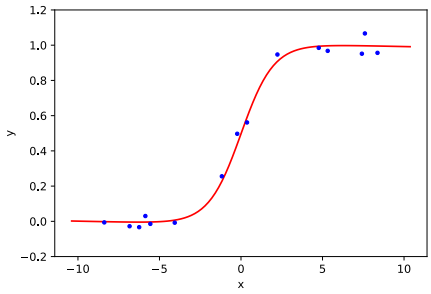
We can create a pipeline that uses a custom basis function for linear regression. Our transformation function takes the `X` array and returns a new one.

```
def sigmoid_basis(X):  # [[x]] -> [[1, x, 1/(1 + e^x)]]
    one = np.ones(X.shape)
    sigmoid = 1 / (1 + np.exp(X))
    return np.concatenate((one, X, sigmoid), axis=1)

from sklearn.preprocessing import FunctionTransformer
model = make_pipeline(
    FunctionTransformer(sigmoid_basis, validate=True),
    LinearRegression(fit_intercept=False)
)
model.fit(X, y)
print(model.named_steps['linearregression'].coef_)
```

Results: excellent (particularly since I knew the data source when designing the basis functions)

```
[ 1.01205597 -0.00198257 -1.03079678]
```



Training and Validation

The high-degree polynomial had a lot of freedom to fit the points, but somehow followed them too closely. How can we decide that is the wrong thing (for a real data set)?

The problem: we're relying too much on this exact data set. We train on it, and then claim it fits the data perfectly. Of course it does.

We are ***overfitting*** the training data.

We need to look at the model on *some other data* where we can get a more honest evaluation.

Common solution: break the available data up.

The usual thing to do is to split into ***training data*** which is used to fit the model, and ***validation data*** which is used to evaluate how well that went.

If the model does well on the validation data (which it hasn't seen before), that's a good sign.

As usual, we can do that in one line:

```
from sklearn.model_selection import train_test_split
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
print(X_train.shape, X_valid.shape)
print(y_train.shape, y_valid.shape)
```

```
(45, 1) (15, 1)
(45,) (15,)
```

[Note: different data set from the previous examples.]

We can use the validation data to produce a "score": some value where larger = better fit.

```
model.fit(X_train, y_train)
print(model.score(X_valid, y_valid))
```

```
0.9742339805871324
```

We have some confidence that the score is related to how well the model will predict for new never-before-seen values.

It also makes some sense to compare the scores on the training and validation data. It will usually be a little lower on the validation data, but being *much* smaller is a sign of overfitting.

```
print(model.score(X_train, y_train))
print(model.score(X_valid, y_valid))
```

```
0.9837583175794484
0.9742339805871324
```

The `.score()` method is vaguely equivalent to:

```
y_predicted = model.predict(X_valid)
score = somehow_score_predictions(y_valid, y_predicted)
```

But was the value we found good? We could explore...

```
def score_polyfit(n):
    model = make_pipeline(
        PolynomialFeatures(degree=n, include_bias=True),
        LinearRegression(fit_intercept=False)
    )
    model.fit(X_train, y_train)
    print('n=%i: score=%.5g' % (n, model.score(X_valid, y_valid)))

score_polyfit(1)
score_polyfit(5)
score_polyfit(9)
score_polyfit(13)
score_polyfit(17)
```

```
n=1: score=0.8202
n=5: score=0.97423
n=9: score=0.98873
n=13: score=0.98741
n=17: score=0.42944
```

Apparently degree 9 is pretty good.

But (as is often the case) there are several values that are all quite good: it's hard to pick exactly one “correct” value. Probably degree 5–13 would be pretty good for this data.

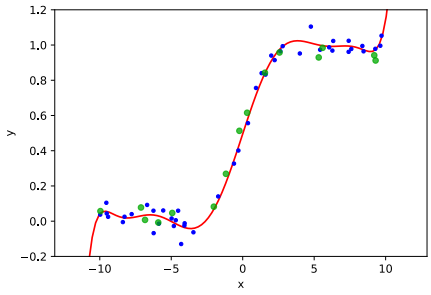
It can also be useful to look at the scores on both the training and validation data.

```
trainscore = model.score(X_train, y_train)
validscore = model.score(X_valid, y_valid)
```

```
n=1: trainscore=0.8479 validscore=0.8202
n=5: trainscore=0.98376 validscore=0.97423
n=9: trainscore=0.99286 validscore=0.98873
n=13: trainscore=0.99321 validscore=0.98741
n=17: trainscore=0.67952 validscore=0.42944
```

If there's a big drop training to validation, it's usually a sign of overfitting.

This is the prediction of the degree 9 model, with training (blue) and validation points (green).



In case you're interested, the degree 17 model: