

W4995 Applied Machine Learning

Preprocessing and Feature Engineering

02/08/17

Andreas Müller

Today we'll talk about preprocessing and feature-engineering. What we're talking about today mostly applies to linear models, and not to tree-based models, but it also applies to neural nets and kernel SVMs.

Notes on homework

- Next one will be shorter.
- Good job everybody!
- Tools are important!
 - Don't use your systems Python, use environments
 - Pick a decent editor / IDE!
 - If you're on windows, consider dual booting

2

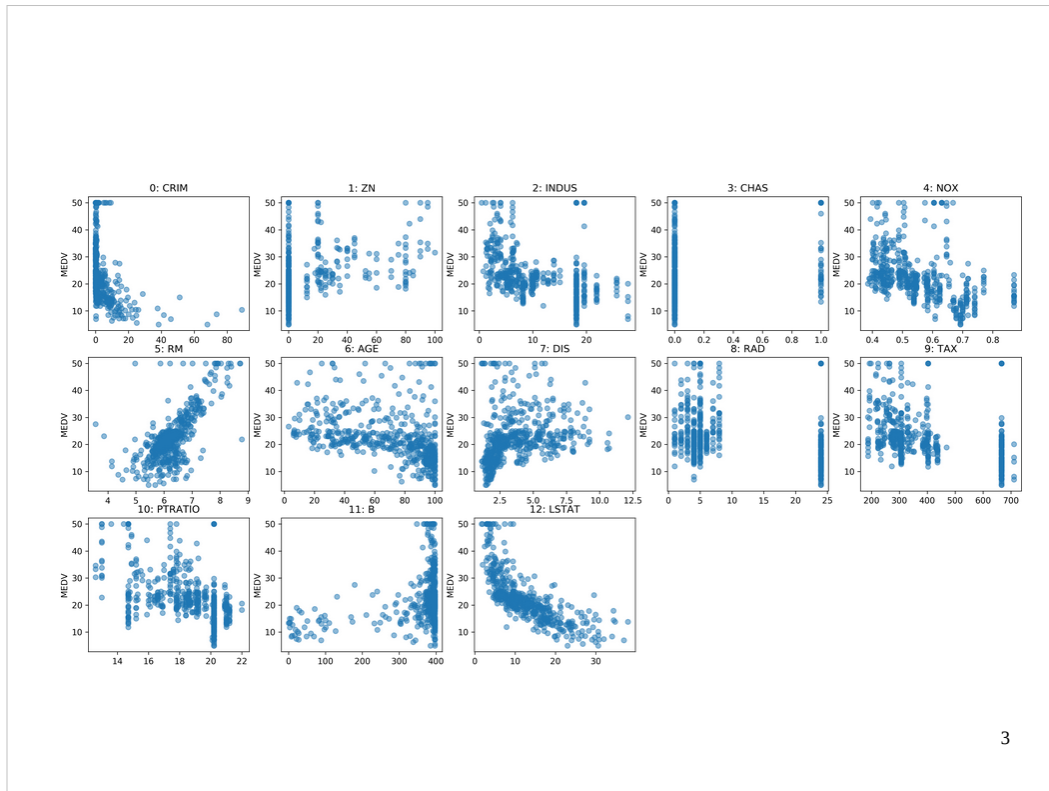
I realized the homework was a bit much for some of you who haven't been familiar with git and testing. I saw many of you made really great progress and got everything together, so good on you. I promise the next homework will be shorter.

While this might have been much all at once, all the pieces are important. Who wants to go to industry? You'll fight these fights all the time!

I gave some advice at the second lecture, and I noticed not everybody took it. If you want be a serious data scientist in industry, start now. Less applicable to researchers (but also).

Don't use system python. Know which environment you're using. Pick an editor.

If you're on windows, get comfy with cygwin or better set up dual boot. You need to know POSIX!



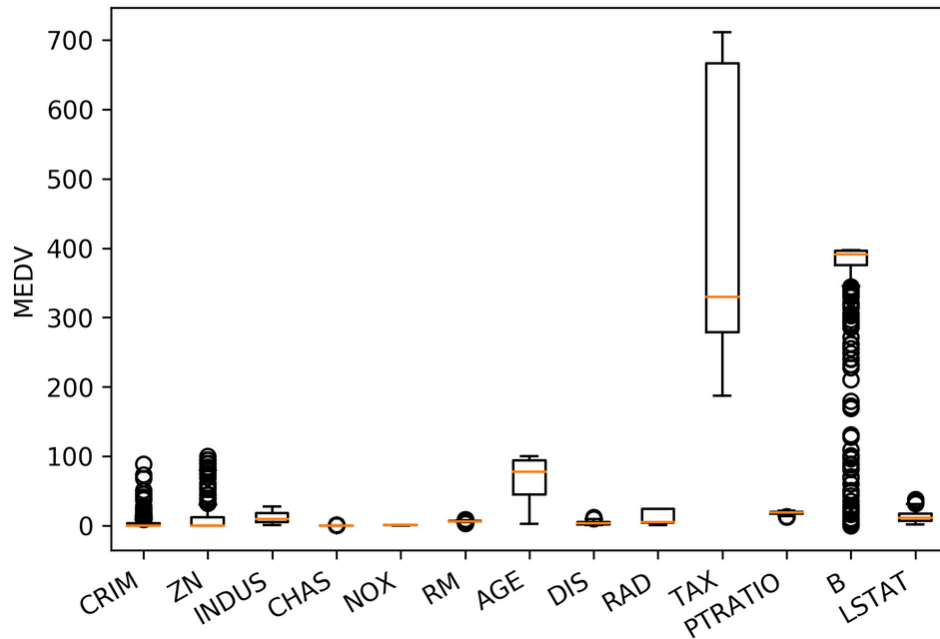
3

Let's go back to the boston housing dataset. The idea was to predict house prices. Here are the features on the x axis and the response, so price, on the y axis.

What are some thing you can notice? (concentrated distributions, skewed distributions, discrete variable, linear and non-linear effects, different scales)

Scaling

```
plt.boxplot(X)
plt.xticks(np.arange(1, X.shape[1] + 1), boston.feature_names, rotation=30, ha="right")
plt.ylabel("MEDV")
<matplotlib.text.Text at 0x7f580303eac8>
```



Let's start with the different scales.

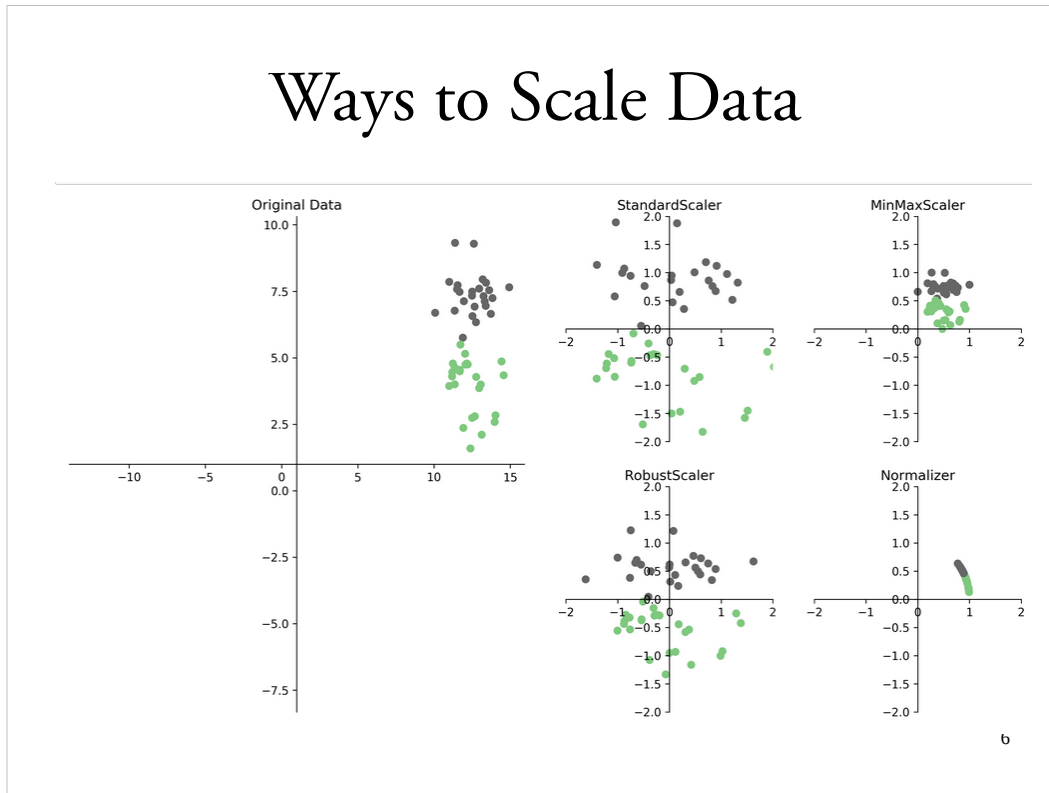
Many model want data that is on the same scale.

KNearestNeighbors: If the distance in TAX is between 300 and 400 then the distance difference in CHArS doesn't matter!

Linear models: the different scales mean different penalty. L2 is the same for all!

We can also see non-gaussian distributions here btw!

Ways to Scale Data



StandardScaler: subtract mean, divide by standard deviation.

MinMaxScaler: subtract minimum, divide by range. Afterwards between 0 and 1.

Robust Scaler: uses median and quantiles, therefore robust to outliers. Similar to StandardScaler.

Normalizer: only considers angle, not length. Helpful for histograms, not that often used.

StandardScaler is usually good, but doesn't guarantee particular min and max values

Sparse Data

- Data with many zeros – only store non-zero entries.
- Subtracting anything will make the data “dense” (no more zeros) and blow the RAM.
- Only scale, don’t center (use MaxAbsScaler)

7

You have to be careful if you have sparse data. Sparse data is data where most entries of the data-matrix X are zero – often only 1% or less are not zero.

You can store this efficiently by only storing the non-zero elements.

Subtracting the mean results in all features becoming non-zero!

So don’t subtract anything, but you can still scale.

MaxAbsScaler scales between -1 and 1 by dividing with the maximum absolute value for each feature.

```
from sklearn.linear_model import Ridge
X, y = boston.data, boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
ridge = Ridge().fit(X_train_scaled, y_train)

X_test_scaled = scaler.transform(X_test)
ridge.score(X_test_scaled, y_test)

0.63448846877867426
```

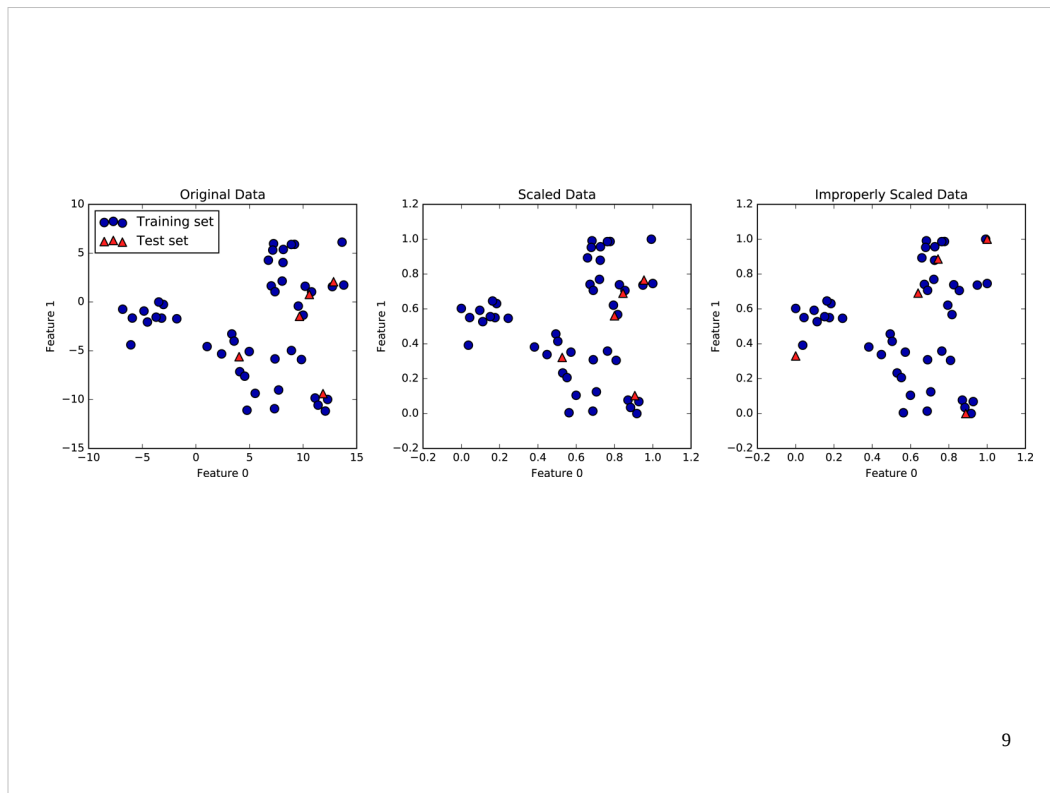
8

Here's how you do the scaling with StandardScaler in scikit-learn. Similar interface to models, but “transform” instead of “predict”. “transform” is always used when you want a new representation of the data.

Fit on training set, transform training set, fit ridge on scaled data, transform test data, score scaled test data.

The fit computes mean and standard deviation on the training set, transform subtracts the mean and the standard deviation.

We fit on the training set and apply transform on both the training and the test set. That means the training set mean gets subtracted from the test set, not the test-set mean. That's quite important.



Here's an illustration why this is important using the min-max scaler. Left is the original data. Center is what happens when we fit on the training set and then transform the training and test set using this transformer. The data looks exactly the same, but the ticks changed. Now the data has a minimum of zero and a maximum of one on the training set. That's not true for the test set, though. No particular range is ensured for the test-set. It could even be outside of 0 and 1. But the transformation is consistent with the transformation on the training set, so the data looks the same.

On the right you see what happens when you use the test-set minimum and maximum for scaling the test set. That's what would happen if you'd fit again on the test set. Now the test set also has minimum at 0 and maximum at 1, but the data is totally distorted from what it was before. So don't do that.

Scikit-Learn API summary

estimator.fit(X_train, [y_train])	
estimator.predict(X_test)	estimator.transform(X_test)
Classification	Preprocessing
Regression	Dimensionality Reduction
Clustering	Feature Extraction
	Feature selection

Efficient short cuts:

```
est.fit_transform(X) == est.fit(X).transform(X)
```

```
est.fit_predict(X) == est.fit(X).predict(X)
```

10

Here's a summary of the scikit-learn methods.

All models have a fit method which takes the training data `X_train`. If the model is supervised, such as our classification and regression models, they also take a `y_train` parameter. The scalers don't use a `y_train` because they don't use the labels at all – you could say they are unsupervised methods, but arguably they are not really learning methods at all.

Models (also known as estimators in scikit-learn) to make a prediction of a target variable, you use the predict method, as in classification and regression.

If you want to create a new representation of the data, a new kind of `X`, then you use the transform method, as we did with scaling. The transform method is also used for preprocessing, feature extraction and feature selection, which we'll see later. All of these change `X` into some new form.

There's two important shortcuts. To fit an estimator and immediately transform the training data, you can use `fit_transform`. That's often more efficient than using first fit and then transform. The same goes for `fit_predict`.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
scores = cross_val_score(RidgeCV(), X_train, y_train, cv=10)
np.mean(scores), np.std(scores)

(0.71718655233314066, 0.12521148650633437)
```

```
scores = cross_val_score(RidgeCV(), X_train_scaled, y_train, cv=10)
np.mean(scores), np.std(scores)

(0.71789046947346136, 0.12695447250917097)
```

```
from sklearn.neighbors import KNeighborsRegressor
scores = cross_val_score(KNeighborsRegressor(), X_train, y_train, cv=10)
np.mean(scores), np.std(scores)

(0.498718658066668803, 0.14628381664585244)
```

```
from sklearn.neighbors import KNeighborsRegressor
scores = cross_val_score(KNeighborsRegressor(), X_train_scaled, y_train, cv=10)
np.mean(scores), np.std(scores)

(0.74979219238995831, 0.10573244928159024)
```

11

Let's apply the scaler to the Boston housing data. First I used the StandardScaler to scale the training data. Then I applied ten-fold cross-validation to evaluate the Ridge model on the data with and without scaling. I used RidgeCV which automatically picks alpha for me. With and without scaling we get an R^2 of about .72, so no difference. Often there is a difference for Ridge, but not in this case.

If we use KNeighborsRegressor instead, we see a big difference. Without scaling R^2 is about .5, and with scaling it's .75. That makes sense since we saw that for distance calculations basically all features are dominated by the TAX feature.

However, there is a bit of a problem with the analysis we did here. Can you see it?

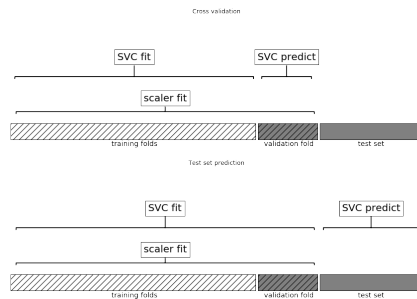
A note on preprocessing (and pipelines)

12

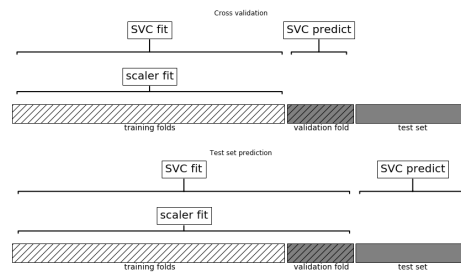
I want to talk a bit more about preprocessing and cross-validation here, and introduce pipelines.

Leaking information

Information leak



No information leak



Need to include preprocessing in cross-validation!

13

What we did was we trained the scaler on the training data, and then applied cross-validation to the scaled data. That's what's shown on the left. The problem is that we use the information of all of the training data for scaling, so in particular the information in the test fold. This is also known as information leakage. If we apply our model to new data, this data will not have been used to do the scaling, so our cross-validation will give us a biased result that might be too optimistic.

On the right you can see how we should do it: we should only use the training part of the data to find the mean and standard deviation, even in cross-validation. That means that for each split in the cross-validation, we need to scale the data a bit differently. This basically means the scaling should happen inside the cross-validation loop, not outside.

In practice, estimating mean and standard deviation is quite robust and you will not see a big difference between the two methods. But for other preprocessing steps that we'll see later, this might make a huge difference. So we should get it right from the start.

```
from sklearn.linear_model import Ridge
X, y = boston.data, boston.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
ridge = Ridge().fit(X_train_scaled, y_train)

X_test_scaled = scaler.transform(X_test)
ridge.score(X_test_scaled, y_test)

0.63448846877867426
```

```
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(StandardScaler(), Ridge())
pipe.fit(X_train, y_train)
pipe.score(X_test, y_test)

0.63448846877867426
```

14

Now I want to show you how to do preprocessing and cross-validation right with scikit-learn.

At the top here you see the workflow for scaling the data and then applying ridge again. Fit the scaler on the training set, transform on the training set, fit ridge on the training set, transform the test set, and evaluate the model.

Because this is such a common pattern, scikit-learn has a tool to make this easier, the pipeline. The pipeline is an estimator that allows you to chain multiple transformations of the data before you apply a final model.

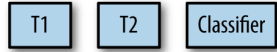
You can build a pipeline using the `make_pipeline` function.

Just provide as parameters all the estimators. All but the last one need to have a transform method. Here we only have two steps: the standard scaler and ridge.

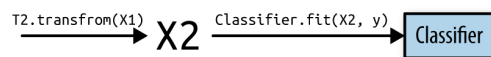
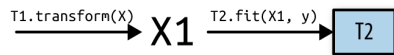
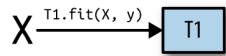
`make_pipeline` returns an estimator that does both steps at once. We can call `fit` on it to fit first the scaler and then ridge on the scaled data, and when we call `score`, it transforms the data and then evaluates the model.

The code below is exactly equivalent to the code above, only shorter.

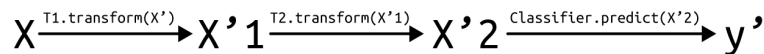
```
pipe = make_pipeline(T1(), T2(), Classifier())
```



```
pipe.fit(X, y)
```



```
pipe.predict(X')
```



15

Let's dive a bit more into the pipeline. Here is an illustration of what happens with three steps, T1, T2 and Classifier. Imagine T1 to be a scaler and T2 to be any other transformation of the data.

If we call fit on this pipeline, it will first call fit on the first step with the input X. Then it will transform the input X to X1, and use X1 to fit the second step, T2. Then it will use T2 to transform the data from X1 to X2. Then it will fit the classifier on X2.

If we call predict on some data X', say the test set, it will call transform on T1, creating X'1. Then it will use T2 to transform X'1 into X'2, and call the predict method of the classifier on X'2. This sounds a bit complicated, but it's really just doing "the right thing" to apply multiple transformation steps.

```
from sklearn.neighbors import KNeighborsRegressor
knn_pipe = make_pipeline(StandardScaler(), KNeighborsRegressor())
scores = cross_val_score(knn_pipe, X_train, y_train, cv=10)
np.mean(scores), np.std(scores)

(0.74531180733825564, 0.10614366233973388)
```

16

How does that help with the cross-validation problem?

Because now all steps are contained in pipeline, we can simply pass the whole pipeline to cross-validation, and all processing will happen inside the cross-validation loop. That solves the data leakage problem.

Here you can see how we can build a pipeline using a standard scaler and kneighborsregressor and pass it to cross-validation.


```
print(knn_pipe.steps)

[('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('kneighborsregressor', KNeighborsRegressor(algorithm='auto', leaf_size=30,
metric='minkowski', metric_params=None, n_jobs=1, n_neighbors=5, p=2,
weights='uniform'))]
```

```
from sklearn.pipeline import Pipeline
pipe = Pipeline((("scaler", StandardScaler()),
                 ("regressor", KNeighborsRegressor)))
```

17

But let's talk a bit more about pipelines, because they are great. The pipeline has an attribute called `steps`, which --- contains its steps. `Steps` is a list of tuples, where the first entry is a string and the second is an estimator (model). The string is the "name" that is assigned to this step in the pipeline. You can see here that our first step is called "standardscaler" in all lower case letters, and the second is called `kneighborsregressor`, also all lower case letters. By default, step names are just lowercased class-names. You can also name the steps yourself using the `Pipeline` class directly. Then you can specify the steps as tuples of name and estimator. `make_pipeline` is just a shortcut to generate the names automatically.

Pipeline and GridSearchCV

```
from sklearn.model_selection import GridSearchCV

knn_pipe = make_pipeline(StandardScaler(), KNeighborsRegressor())
param_grid = {'kneighborsregressor__n_neighbors': range(1, 10)}
grid = GridSearchCV(knn_pipe, param_grid, cv=10)
grid.fit(X_train, y_train)
print(grid.best_params_)
print(grid.score(X_test, y_test))

{'kneighborsregressor__n_neighbors': 7}
0.600015753391
```

18

These names are important for using pipelines with grid-search. Recall that for using GridSearchCV you need to specify a parameter grid as a dictionary, where the keys are the parameter names. If you are using a pipeline inside GridSearchCV, you need to specify not only the parameter name, but also the step name – because multiple steps could have a parameter with the same name.

The way to do this is to use the stepname, then two underscores, and then the parameter name, as the key for the param_grid dictionary.

You can see that the best_params_ will have this same format.

This way you can tune the parameters of all steps in a pipeline at once!

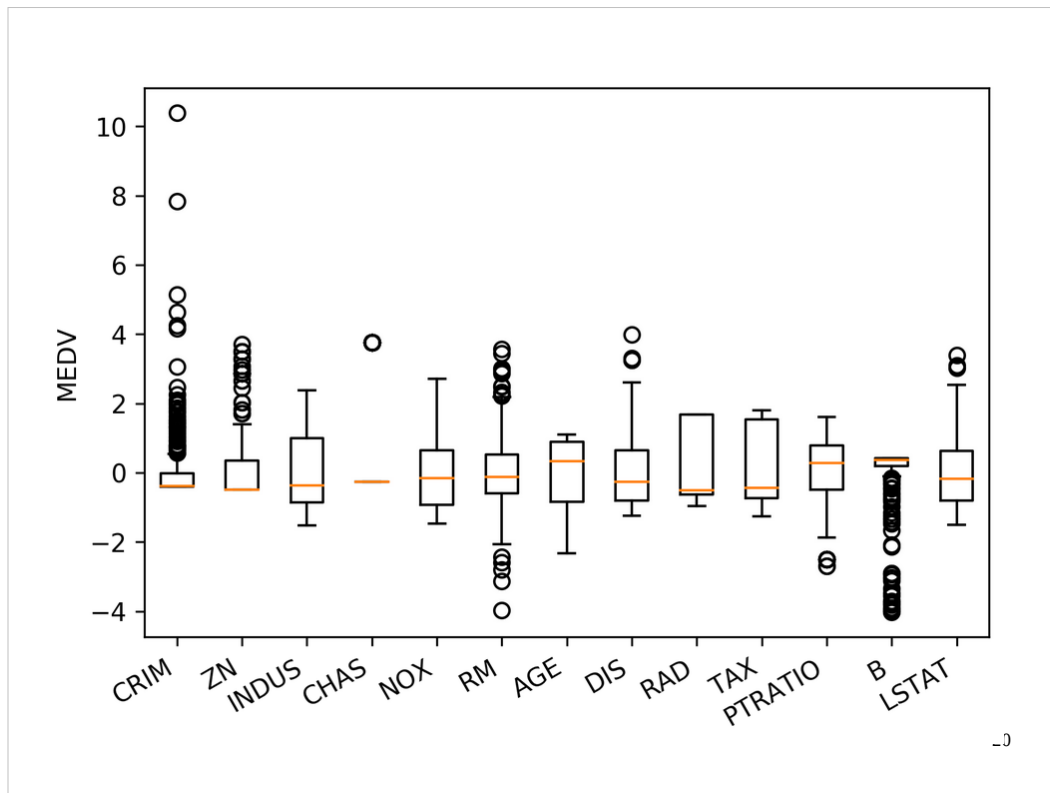
And you don't have to worry about leaking information, since all transformations are contained in the pipeline.

You should always use pipelines for preprocessing. Not only does it make your code shorter, it also makes it less likely that you have bugs.

Feature Distributions

19

Now that we discussed scaling and pipelines, let's talk about some more preprocessing methods. One important aspect is dealing with different input distributions.



Here is a box plot of the boston housing data after transforming it with the standard scaler. Even though the mean and standard deviation are the same for all features, the distributions are quite different. You can see very concentrated distributions like Crim and B, and very skewed distributions like RAD and Tax (and also crim and B).

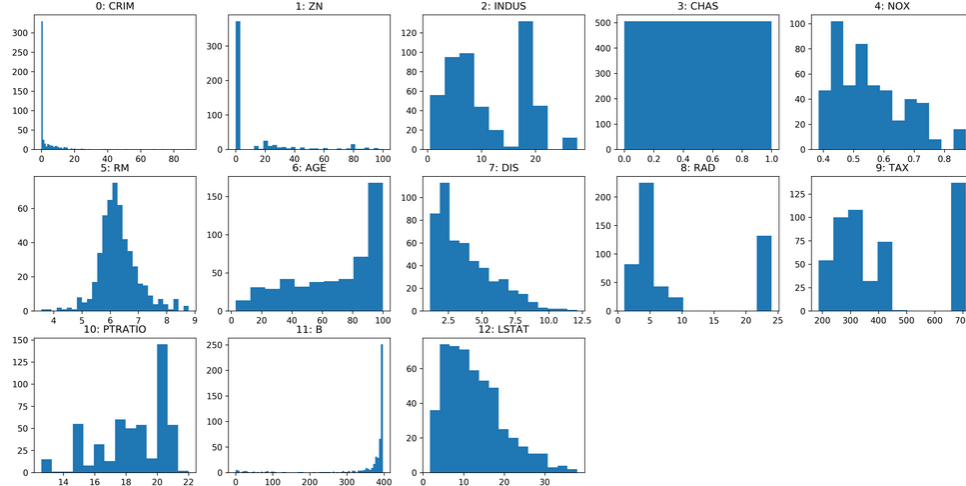
Many models, in particular linear models and neural networks, work better if the features are approximately normal distributed.

Let's also check out the histograms of the data to see a bit better what's going on.

```

fig, axes = plt.subplots(3, 5, figsize=(20, 10))
for i, ax in enumerate(axes.ravel()):
    if i > 12:
        ax.set_visible(False)
        continue
    ax.hist(X[:, i], bins="auto")
    ax.set_title("{}: {}".format(i, boston.feature_names[i]))

```



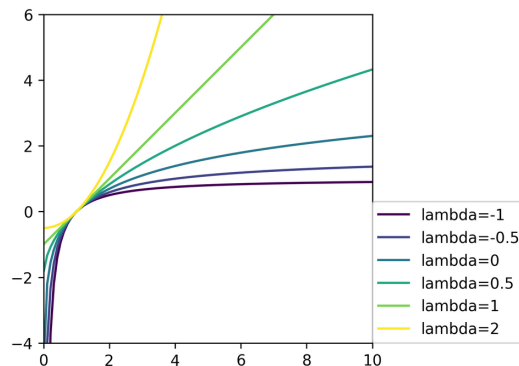
21

Clearly CRIM and ZN and B are very peaked, and LSTAT and DIS and Age are very asymmetric. Sometimes you can use a hack like applying a logarithm to the data to get better behaved values. There is slightly more rigorous technique though.

Box-Cox Transform

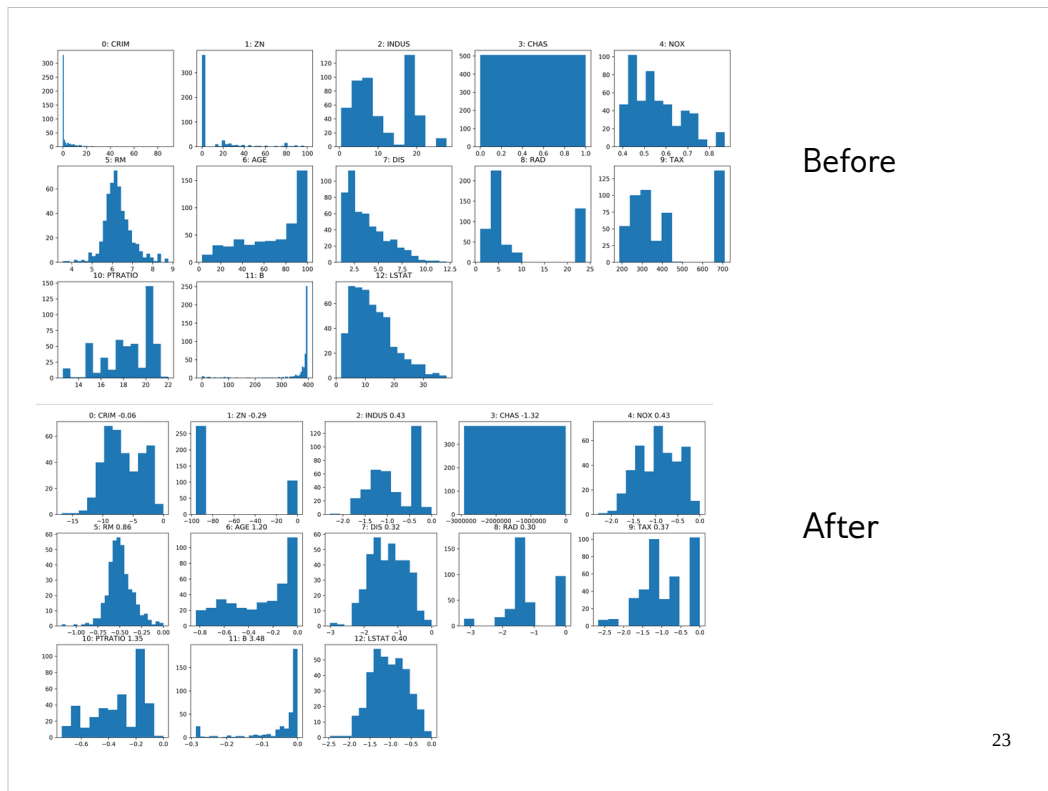
$$bc_{\lambda}(x) = \begin{cases} \frac{x^{\lambda}-1}{\lambda} & \text{if } \lambda \neq 0 \\ \log(x) & \text{if } \lambda = 0 \end{cases}$$

Only applicable for positive x!

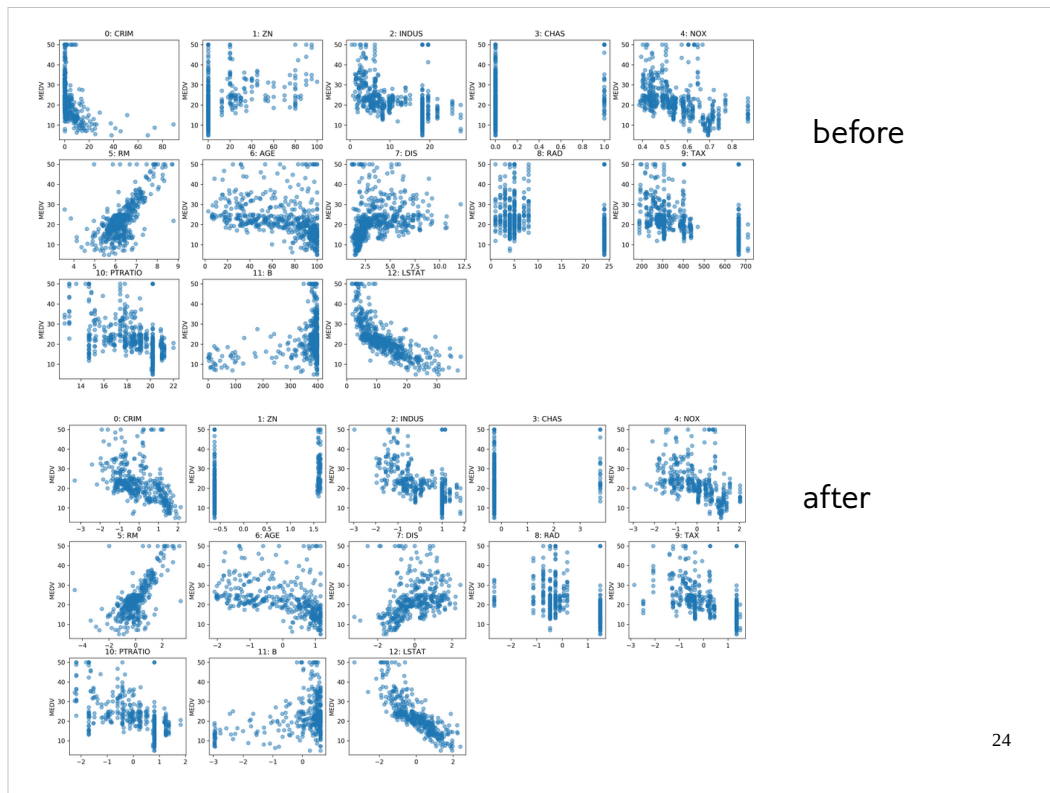


22

The Box-Cox transformation is a family of univariate functions to transform your data, parametrized by a parameter lambda. For lambda=1 the function is the identity, for lambda = 2 it is square, lambda =0 is log and there is many other functions in between. For a given dataset, a separate parameter lambda is determined for each feature, by minimizing the skewdness of the data (making skewdness close to zero, not close to -inf), so it is more “gaussian”. The skewdness of a function is a measure of the asymmetry of a function and is 0 for functions that are symmetric around their mean. Unfortunately the Box-Cox transformation is only applicable to positive features.



Here are the histograms of the original data and the transformed data. The title of each subplot shows the estimated lambda. If the lambda is close to 1, the transformation didn't change much. If it is away from 1, there was a significant transformation. You can clearly see the effect on “CRIM” which was approximately log-transformed, and lstat and nox which were approximately transformed by sqrt. For the binary CHAS the transformation doesn't make a lot of sense, though.



Here is a comparison of the feature vs response plot before and after the box-cox transformation. The dis, lstat and crim relationships now look a bit more obvious and linear.

Discrete Features

Categorical Variables

$$\{\text{"red"}, \text{"green"}, \text{"blue"}\} \subset \mathbb{R}^p \quad ?$$

Before we can apply a machine learning algorithm, we first need to think about how we represent our data. Earlier, I said $x \in \mathbb{R}^n$. That's not how you usually get data. Often data has units, possibly different units for different sensors, it has a mixture of continuous values and discrete values, and different measurements might be on totally different scales.

First, let me explain how to deal with discrete input variables, also known as categorical features. They come up in nearly all applications.

Let's say you have three possible values for a given measurement, whether you used setup1 setup2 or setup3. You could try to encode these into a single real number, say 0, 1 and 2, or e , π , τ .

However, that would be a bad idea for algorithms like linear regression.

Categorical Variables

	"red"	"green"	"blue"
	1	0	0
	0	1	0
	0	0	1

If you encode all three values using the same feature, then you are imposing a linear relation between them, and in particular you define an order between the categories. Usually, there is no semantic ordering of the categories, and so we shouldn't introduce one in our representation of the data.

Instead, we add one new feature for each category, And that feature encodes whether a sample belongs to this category or not.

That's called a one-hot encoding, because only one of the three features in this example is active at a time. You could actually get away with $n-1$ features, but in machine learning that usually doesn't matter.

```
import pandas as pd
df = pd.DataFrame({'salary': [103, 89, 142, 54, 63, 219],
                  'boro': ['Manhattan', 'Queens', 'Manhattan', 'Brooklyn', 'Brooklyn', 'Bronx']})
df
```

	boro	salary
0	Manhattan	103
1	Queens	89
2	Manhattan	142
3	Brooklyn	54
4	Brooklyn	63
5	Bronx	219

```
pd.get_dummies(df)
```

	salary	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens
0	103	0.0	0.0	1.0	0.0
1	89	0.0	0.0	0.0	1.0
2	142	0.0	0.0	1.0	0.0
3	54	0.0	1.0	0.0	0.0
4	63	0.0	1.0	0.0	0.0
5	219	1.0	0.0	0.0	0.0

```
df = pd.DataFrame({'salary': [103, 89, 142, 54, 63, 219],
                  'boro': [0, 1, 0, 2, 2, 3]})
df
```

	boro	salary
0	0	103
1	1	89
2	0	142
3	2	54
4	2	63
5	3	219

```
pd.get_dummies(df)
```

	boro	salary
0	0	103
1	1	89
2	0	142
3	2	54
4	2	63
5	3	219

```
pd.get_dummies(df, columns=['boro'])
```

	salary	boro_0	boro_1	boro_2	boro_3
0	103	1.0	0.0	0.0	0.0
1	89	0.0	1.0	0.0	0.0
2	142	1.0	0.0	0.0	0.0
3	54	0.0	0.0	1.0	0.0
4	63	0.0	0.0	1.0	0.0
5	219	0.0	0.0	0.0	1.0

	boro	salary
0	Manhattan	103
1	Queens	89
2	Manhattan	142
3	Brooklyn	54
4	Brooklyn	63
5	Bronx	219

	salary	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Queens
0	103	0.0	0.0	1.0	0.0
1	89	0.0	0.0	0.0	1.0
2	142	0.0	0.0	1.0	0.0
3	54	0.0	1.0	0.0	0.0
4	63	0.0	1.0	0.0	0.0
5	219	1.0	0.0	0.0	0.0

	boro	salary
0	Staten Island	73
1	Manhattan	98
2	Brooklyn	204
3	Bronx	54

	salary	boro_Bronx	boro_Brooklyn	boro_Manhattan	boro_Staten Island
0	73	0.0	0.0	0.0	1.0
1	98	0.0	0.0	1.0	0.0
2	204	0.0	1.0	0.0	0.0
3	54	1.0	0.0	0.0	0.0

Pandas Categorical Columns

```
import pandas as pd
df = pd.DataFrame({'salary': [103, 89, 142, 54, 63, 219],
                  'boro': ['Manhattan', 'Queens', 'Manhattan', 'Brooklyn', 'Brooklyn', 'Bronx']})
df.boro = df.boro.astype("category", categories=['Manhattan', 'Queens', 'Brooklyn', 'Bronx', 'Staten Island'])
pd.get_dummies(df)
```

	salary	boro_Manhattan	boro_Queens	boro_Brooklyn	boro_Bronx	boro_Staten Island
0	103	1.0	0.0	0.0	0.0	0.0
1	89	0.0	1.0	0.0	0.0	0.0
2	142	1.0	0.0	0.0	0.0	0.0
3	54	0.0	0.0	1.0	0.0	0.0
4	63	0.0	0.0	1.0	0.0	0.0
5	219	0.0	0.0	0.0	1.0	0.0

OneHotEncoder

```
from sklearn.preprocessing import OneHotEncoder

df = pd.DataFrame({'salary': [103, 89, 142, 54, 63, 219],
                   'boro': [0, 1, 0, 2, 2, 3]})
X = df.values
ohe = OneHotEncoder(categorical_features=[0]).fit(X)
ohe.transform(X).toarray()

array([[ 1.,  0.,  0.,  0., 103.],
       [ 0.,  1.,  0.,  0.,  89.],
       [ 1.,  0.,  0.,  0., 142.],
       [ 0.,  0.,  1.,  0.,  54.],
       [ 0.,  0.,  1.,  0.,  63.],
       [ 0.,  0.,  0.,  1., 219.]])
```

- Fit-transform paradigm ensures train and test-set categories correspond.
- Only works for integers right now, not strings (we're fixing this).

One-Hot vs statisticians

- One-hot is redundant (last one is $1 - \text{sum of others}$)
- Can introduce co-linearity
- Can drop one
- Choice which one matters for penalized models
- Keeping all can make the model more interpretable

Models Supporting Discrete Features

- In principle:
 - All tree-based models
- In scikit-learn:
 - None
- In scikit-learn soon:
 - Decision trees, random forests

Count-Based Encoding

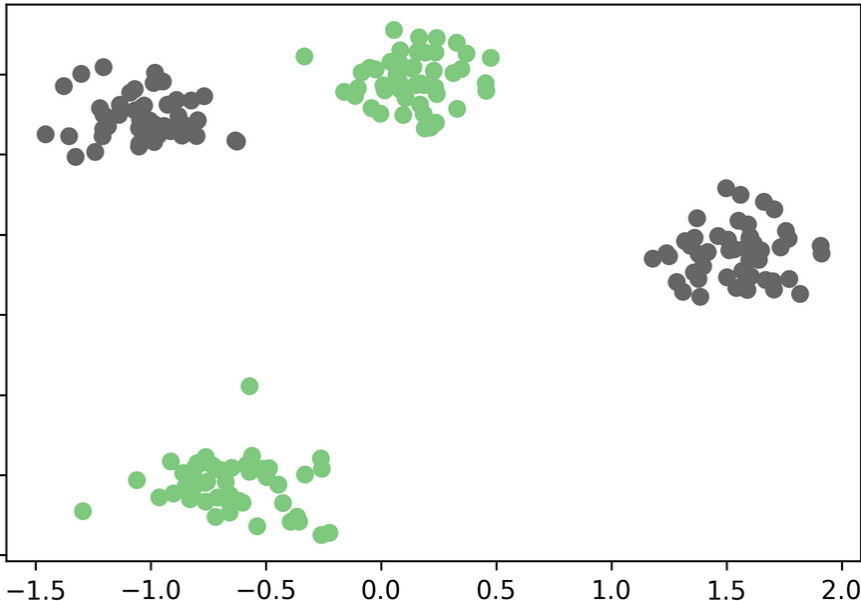
- For high cardinality categorical features
- Example: US states, given low samples
- Instead of 50 one-hot variables, one “response encoded” variable.
- For regression:
 - “people in this state have an average response of y ”
- Binary classification:
 - “people in this state have likelihood p for class 1”
- Multiclass:
 - One feature per class: probability distribution

Example

- FIXME

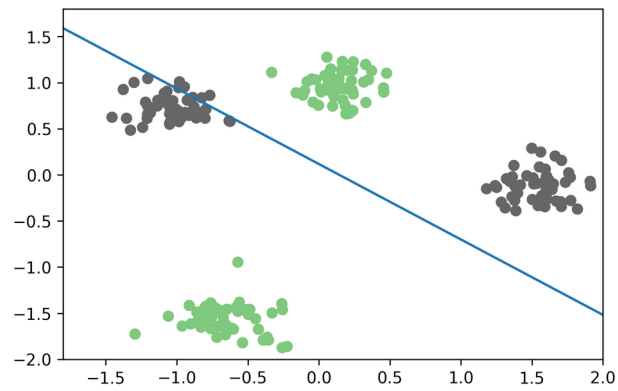
Feature Engineering

Interaction Features



A scatter plot titled "Interaction Features" showing two distinct clusters of data points. The x-axis ranges from -1.5 to 2.0, and the y-axis ranges from -2.0 to 1.0. The dark grey cluster is located on the left side of the plot, centered around x = -1.0 and y = 0.7. The light green cluster is located on the right side of the plot, centered around x = 0.2 and y = 1.0. There are also a few outliers for both clusters.

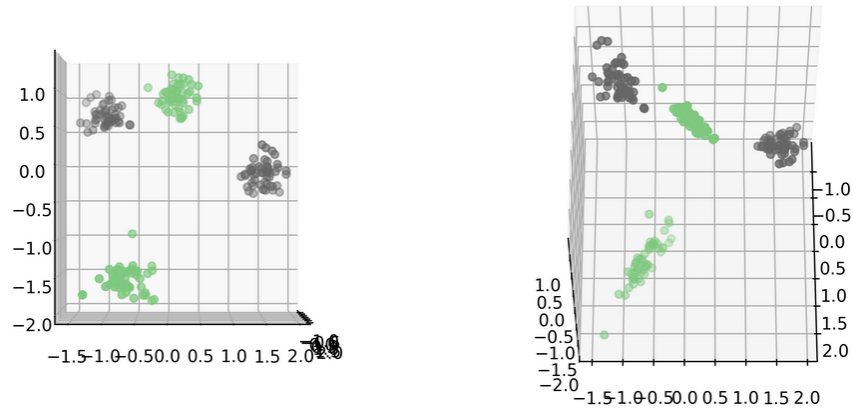
Interaction Features

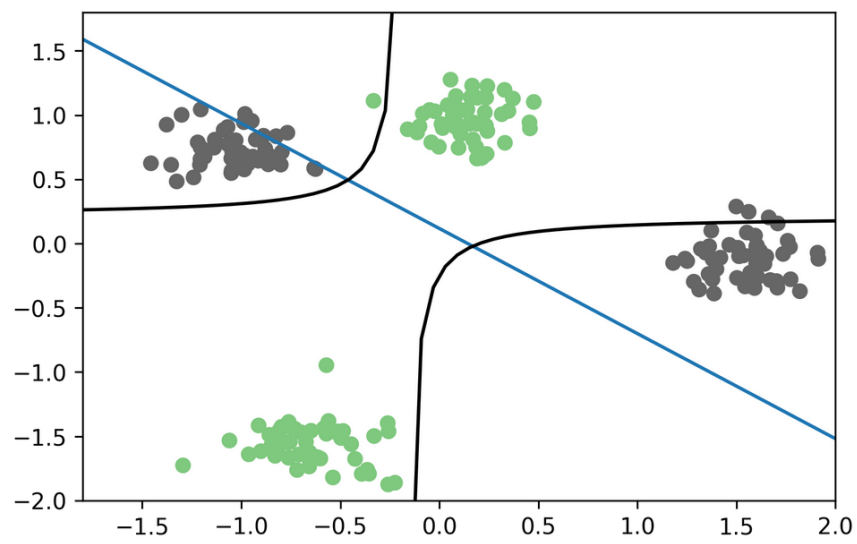


```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
logreg = LogisticRegressionCV().fit(X_train, y_train)
logreg.score(X_test, y_test)
```

0.5

```
# Same as PolynomialFeatures(order=2, interactions_only=True)
X_interaction = np.hstack([X, X[:, 0:1] * X[:, 1:]])
```





```
: X_i_train, X_i_test, y_train, y_test = train_test_split(X_interaction, y, random_state=0)
logreg3 = LogisticRegressionCV().fit(X_i_train, y_train)
logreg3.score(X_i_test, y_test)
: 0.95999999999999996
```

	age	articles_bought	gender	spend\$	time_online
0	14	5	M	70	269
1	16	10	F	12	1522
2	12	2	M	42	235
3	25	1	F	64	63
4	22	1	F	93	21

	age	articles_bought	spend\$	time_online	gender_F	gender_M
0	14	5	70	269	0.0	1.0
1	16	10	12	1522	1.0	0.0
2	12	2	42	235	0.0	1.0
3	25	1	64	63	1.0	0.0
4	22	1	93	21	1.0	0.0

	age_M	articles_bought_M	spend\$_M	time_online_M	gender_M_M	age_F	articles_bought_F	spend\$_F	time_online_F	gender_F_F
0	14.0	5.0	70.0	269.0	1.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	16.0	10.0	12.0	1522.0	1.0
2	12.0	2.0	42.0	235.0	1.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	25.0	1.0	64.0	63.0	1.0
4	0.0	0.0	0.0	0.0	0.0	22.0	1.0	93.0	21.0	1.0

One model per gender!

Keep original: common model + model for each gender to adjust.

Product of multiple categoricals: common model + multiple models to adjust for combinations

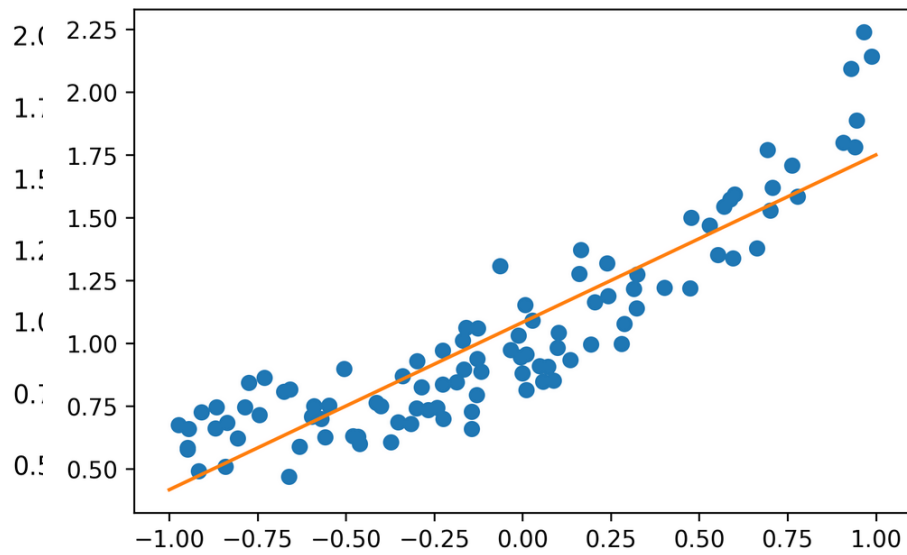
```
age articles_bought gender spend$ time_online  + Male * (age articles_bought spend$ time_online )  
  
+ Female * (age articles_bought spend$ time_online )  
  
+ (age > 20) * (age articles_bought gender spend$ time_online)  
  
+ (age <= 20) * (age articles_bought gender spend$ time_online)  
  
+ (age <= 20) * Male * (age articles_bought gender spend$ time_online)  
  
....
```

```

from sklearn.linear_model import LinearRegression
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
line = np.linspace(-1, 1, 100).reshape(-1, 1)
plt.plot(x, y, 'o')
plt.plot(line, lr.predict(line))
lr.score(X_test, y_test)

```

2.4 0.76332391526170273

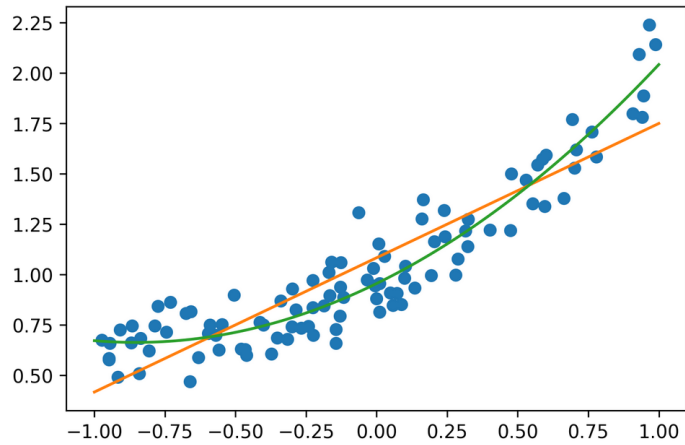


Polynomial Features

```
poly_lr = make_pipeline(PolynomialFeatures(include_bias=False), LinearRegression())
poly_lr.fit(X_train, y_train)

plt.plot(x, y, 'o')
plt.plot(line, lr.predict(line))
plt.plot(line, poly_lr.predict(line))
poly_lr.score(X_test, y_test)
```

0.83367862697542183



Polynomial Features

- `PolynomialFeatures()` adds polynomials and interactions.
- Transformer interface like scalers etc.
- Create polynomial algorithms with `make_pipeline!`

Polynomial Features

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures()
X_bc_poly = poly.fit_transform(X_bc_scaled)
print(X_bc_scaled.shape)
print(X_bc_poly.shape)
```

```
(379, 13)
(379, 105)
```

```
scores = cross_val_score(RidgeCV(), X_bc_scaled, y_train, cv=10)
np.mean(scores), np.std(scores)
```

```
(0.75938668786977215, 0.081102768502434197)
```

```
scores = cross_val_score(RidgeCV(), X_bc_poly, y_train, cv=10)
np.mean(scores), np.std(scores)
```

```
(0.86528575688853915, 0.080389451676537382)
```

Other features?

- Plot the data, see if there are periodic patterns!

Discretization and Binning

- Loses data.
- Target-independent might be bad
- Powerful combined with interactions to create new features!