Introduction to Supervised Learning Linear Regression

AcadView

May 30, 2018

---

1 Overview

The well-known concept of garbage ingarbage out applies 100% to any task in machine learning. Any experienced professional can recall numerous times when a simple model trained on high-quality data was proven to be better than a complicated multi-model ensemble built on data that wasn't clean.

To start, we review three similar but different tasks:

- **feature extraction and feature engineering:** transformation of raw data into features suitable for modeling;
- **feature transformation:** transformation of data to improve the accuracy of the algorithm;
- **feature selection:** removing unnecessary features.

This article will contain almost no math, but there will be a fair amount of code.

2 Feature Extraction

In practice, data rarely comes in the form of ready-to-use matrices. That's why every task begins with feature extraction. Sometimes, it can be enough to read the csv file and convert it into numpy.array, but this is a rare exception. Let's look at some of the popular types of data from which features can be extracted.

2.1 Texts

Text is a type of data that can come in different formats; there are so many text processing methods that cannot fit in a single article. Nevertheless, we will review the most popular ones.

Before working with text, one must tokenize it. Tokenization implies splitting the text into units (hence, tokens). Most simply, tokens are just the words. But splitting by word can lose some of the meaning – "Santa Barbara" is one token, not two, but "rock'n'roll" should not be split into two tokens. There are ready-to-use tokenizers that take into account peculiarities of the language, but they make mistakes as well, especially when you work with specific sources of text (newspapers, slang, misspellings, typos).

After tokenization, you will normalize the data. For text, this is about stemming and/or lemmatization; these are similar processes used to process different forms of a word. One can read about the difference between them here.

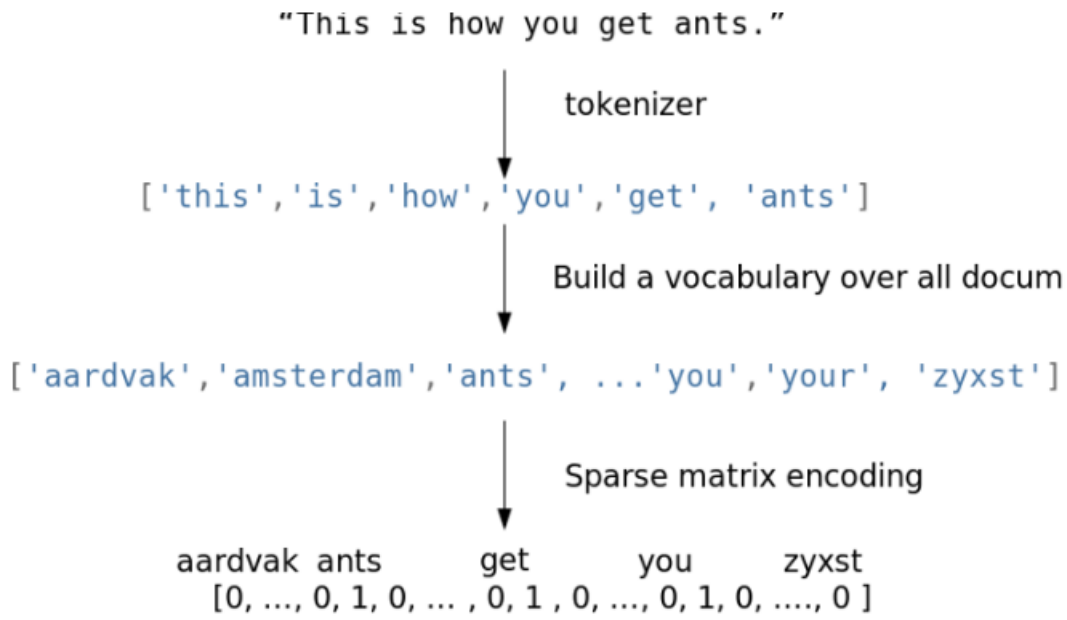
So, now that we have turned the document into a sequence of words, we can represent it with vectors. The easiest approach is called Bag of Words: we create a vector with the length of the dictionary, compute the number of occurrences of each word in the text, and place that number of occurrences in the appropriate position in the vector. The process described looks simpler in code:

```
from functools import reduce
import numpy as np

texts = [['i', 'have', 'a', 'cat'],
        ['he', 'have', 'a', 'dog'],
        ['he', 'and', 'i', 'have', 'a', 'cat', 'and', 'a', 'dog']]

dictionary = list(enumerate(set(list(reduce(lambda x, y: x + y,
texts)))))

def vectorize(text):
    vector = np.zeros(len(dictionary))
    for i, word in dictionary:
        num = 0
        for w in text:
            if w == word:
                num += 1
        if num:
            vector[i] = num
    return vector

for t in texts:
    print(vectorize(t))
```

Expected result:

```
[0. 1. 0. 1. 1. 0. 1.]
[0. 1. 1. 0. 1. 1. 0.]
[2. 1. 1. 1. 2. 1. 1.]
```

Here is an illustration of the process:



This is an extremely naive implementation. In practice, you need to consider stop words, the maximum length of the dictionary, more efficient data structures (usually text data is converted to a sparse vector), etc.

When using algorithms like Bag of Words, we lose the order of the words in the text, which means that the texts "i have no cows" and "no, i have cows" will appear identical after vectorization when, in fact, they have the opposite meaning. To avoid this problem, we can revisit our tokenization step and use N-grams (the sequence of N consecutive tokens) instead.

```
In : from sklearn.feature_extraction.text import CountVectorizer

In : vect = CountVectorizer(ngram_range=(1,1))

In : vect.fit_transform(['no i have cows', 'i have no
cows']).toarray()
Out:
array([[1, 1, 1],
       [1, 1, 1]], dtype=int64)

In : vect.vocabulary_
Out: {'cows': 0, 'have': 1, 'no': 2}

In : vect = CountVectorizer(ngram_range=(1,2))

In : vect.fit_transform(['no i have cows', 'i have no
cows']).toarray()
Out:
array([[1, 1, 1, 0, 1, 0, 1],
       [1, 1, 0, 1, 1, 1, 0]], dtype=int64)

In : vect.vocabulary_
Out: {'cows': 0,
      'have': 1,
      'have cows': 2,
      'have no': 3,
      'no': 4,
      'no cows': 5,
      'no have': 6}
```

Also note that one does not have to use only words. In some cases, it is possible to generate N-grams of characters. This approach would be able to account for similarity of related words or handle typos.

```
from scipy.spatial.distance import euclidean
from sklearn.feature_extraction.text import CountVectorizer

vect = CountVectorizer(ngram_range=(3,3), analyzer='char_wb')

n1, n2, n3, n4 = vect.fit_transform(['andersen', 'petersen',
'petrov', 'smith']).toarray()

euclidean(n1, n2), euclidean(n2, n3), euclidean(n3, n4)
# (2.8284271247461903, 3.1622776601683795, 3.3166247903554)
```

Adding onto the Bag of Words idea: words that are rarely found in the corpus (in all the documents of this dataset) but are present in this particular document might be more important. Then it makes sense to increase the weight of more domain-specific words to separate them out from common words. This approach is called TF-IDF (term frequency-inverse document frequency). The default option is as follows:

$$idf(t, D) = \log \frac{|D|}{df(d, t) + 1}$$

$$tfidf(t, d, D) = tf(t, d) \times idf(t, D)$$

3 Feature transformations

3.1 Normalization and changing distribution

Monotonic feature transformation is critical for some algorithms and has no effect on others. This is one of the reasons for the increased popularity of decision trees and all its derivative algorithms (random forest, gradient boosting). Not everyone can or want to tinker with transformations, and these algorithms are robust to unusual distributions.

There are also purely engineering reasons: **np.log** is a way of dealing with large numbers that do not fit in **np.float64**. This is an exception rather than a rule; often it's driven by the desire to adapt the dataset to the requirements of the algorithm. Parametric methods usually require a minimum of symmetric and unimodal distribution of data, which is not always given in real data. There may be more stringent requirements; recall our earlier article about linear models.

However, data requirements are imposed not only by parametric methods; K nearest neighbors will predict complete nonsense if features are not normalized e.g. when one distribution is located in the vicinity of zero and does not go beyond (-1, 1) while the other's range is on the order of hundreds of thousands.

A simple example: suppose that the task is to predict the cost of an apartment from two variablesthe distance from city center and the number of rooms. The number of rooms rarely exceeds 5 whereas the distance from city center can easily be in the thousands of meters.

The simplest transformation is Standard Scaling (or Z-score normalization):

$$z = \frac{x - \mu}{\sigma}$$

Note that Standard Scaling does not make the distribution normal in the strict sense.

```
In : from sklearn.preprocessing import StandardScaler

In : from scipy.stats import beta

In : from scipy.stats import shapiro

In : data = beta(1, 10).rvs(1000).reshape(-1, 1)

In : shapiro(data)
Out: (0.8783774375915527, 3.0409122263582326e-27)

# Value of the statistic, p-value

In : shapiro(StandardScaler().fit_transform(data))
Out: (0.8783774375915527, 3.0409122263582326e-27)

# With such p-value we'd have to reject the null hypothesis of
normality of the data
```

But, to some extent, it protects against outliers:

```
In : data = np.array([1, 1, 0, -1, 2, 1, 2, 3, -2, 4,
100]).reshape(-1, 1).astype(np.float64)

In : StandardScaler().fit_transform(data)
Out:
array([[-0.31922662],
       [-0.31922662],
       [-0.35434155],
       [-0.38945648],
       [-0.28411169],
       [-0.31922662],
       [-0.28411169],
       [-0.24899676],
       [-0.42457141],
       [-0.21388184],
       [ 3.15715128]])

In : (data - data.mean()) / data.std()
Out:
array([[-0.31922662],
       [-0.31922662],
       [-0.35434155],
       [-0.38945648],
       [-0.28411169],
       [-0.31922662],
       [-0.28411169],
       [-0.24899676],
       [-0.42457141],
       [-0.21388184],
       [ 3.15715128]])
```

Another fairly popular option is**MinMax Scaling**, which brings all the points within a predetermined interval (typically (0, 1)).

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

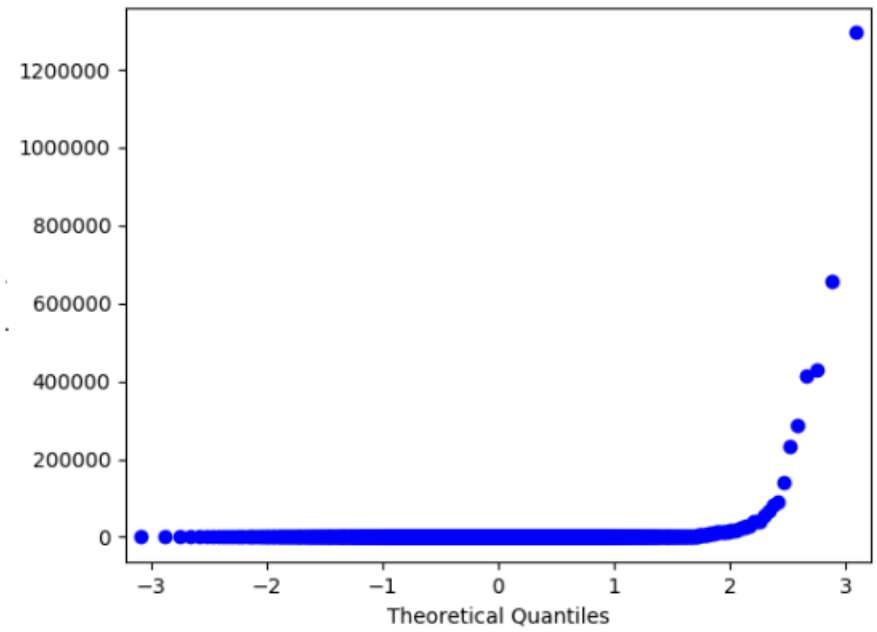
```
In : from sklearn.preprocessing import MinMaxScaler

In : MinMaxScaler().fit_transform(data)
Out:
array([[ 0.02941176],
       [ 0.02941176],
       [ 0.01960784],
       [ 0.00980392],
       [ 0.03921569],
       [ 0.02941176],
       [ 0.03921569],
       [ 0.04901961],
       [ 0. ],
       [ 0.05882353],
       [ 1. ]])

In : (data - data.min()) / (data.max() - data.min())
Out:
array([[ 0.02941176],
       [ 0.02941176],
       [ 0.01960784],
       [ 0.00980392],
       [ 0.03921569],
       [ 0.02941176],
       [ 0.03921569],
       [ 0.04901961],
       [ 0. ],
       [ 0.05882353],
       [ 1. ]])
```

StandardScaling and MinMax Scaling have similar applications and are often more or less interchangeable. However, if the algorithm involves the calculation of distances between points or vectors, the default choice is StandardScaling. But MinMax Scaling is useful for visualization by bringing features within the interval (0, 255). If we assume that some data is not normally distributed but is described by the log-normal distribution, it can easily be transformed to a normal distribution:

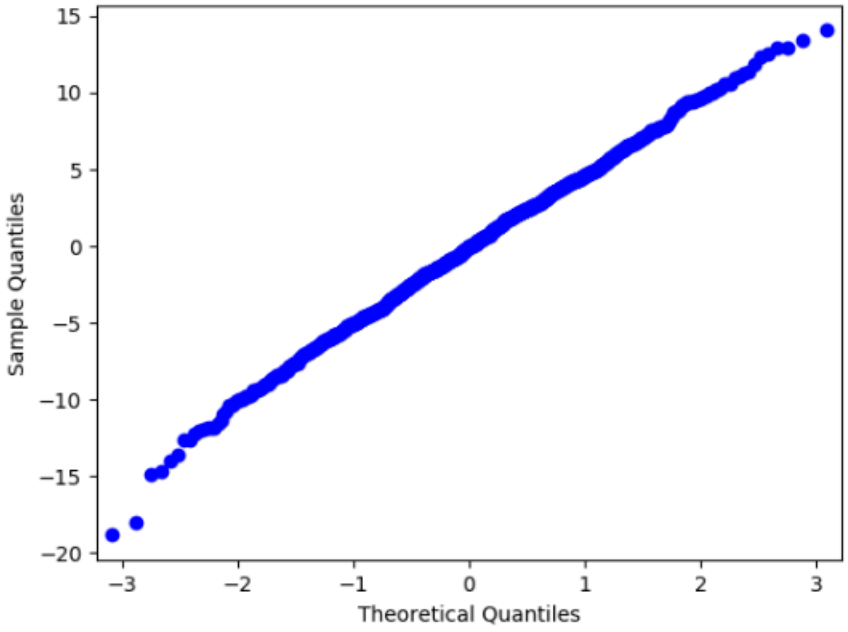
```
In : from scipy.stats import lognorm

In : data = lognorm(s=1).rvs(1000)

In : shapiro(data)
Out: (0.05714237689971924, 0.0)

In : shapiro(np.log(data))
Out: (0.9980740547180176, 0.3150389492511749)
```

The lognormal distribution is suitable for describing salaries, price of securities, urban population, number of comments on articles on the internet, etc. However, to apply this procedure, the underlying distribution does not necessarily have to be lognormal; you can try to apply this transformation to any distribution with a heavy right tail. Furthermore, one can try to use other similar transformations, formulating their own hypotheses on how to approximate the available distribution to a normal. In addition, you can also try adding a constant to the feature**np.log (x + const).**

In the examples above, we have worked with synthetic data and strictly tested normality using the Shapiro-Wilk test. Let's try to look at some real data and test for normality using a less formal methodQ-Q plot. For a normal distribution, it will look like a smooth diagonal line, and visual anomalies should be intuitively understandable.



Q-Q plot for lognormal distribution



Q-Q plot for the same distribution after taking the logarithm

3.2 Filling in the missing values

Not many algorithms can work with missing values, and the real world often provides data with gaps. Fortunately, this is one of the tasks for which one doesn't need any creativity. Both key python libraries for data analysis provide easy-to-use solutions: **pandas.DataFrame.fillna** and **sklearn.preprocessing.Imputer**.

These solutions do not have any magic happening behind the scenes. Approaches to handling missing values are pretty straightforward:

- encode missing values with a separate blank value like "n/a" (for categorical variables);
- use the most probable value of the feature (mean or median for the numerical variables, the most common value for categorical variables);
- or, conversely, encode with some extreme value (good for decision-tree models since it allows the model to make a partition between the missing and non-missing values);
- for ordered data (e.g. time series), take the adjacent valuenext or previous.

Easy-to-use library solutions sometimes suggest sticking to something like $\mathbf{df} = \mathbf{df}.\mathbf{fillna}(\mathbf{0})$ and not sweat the gaps. But this is not the best solution: data preparation takes more time than building models, so thoughtless gap-filling may hide a bug in processing and damage the model.

4 Feature Selection

Why would it even be necessary to select features? To some, this idea may seem counter-intuitive, but there are at least two important reasons to get rid of unimportant features. The first is clear to every engineer: the more data, the higher the computational complexity. As long as we work with toy datasets, the size of the data is not a problem, but, for real loaded production systems, hundreds of extra features will be quite tangible. The second reason is that some algorithms take noise (non-informative features) as a signal and overfit.

4.1 Statistical approaches

The most obvious candidate for removal is a feature whose value remains unchanged, i.e., it contains no information at all. If we build on this thought, it is reasonable to say that features with low variance are worse than those with high variance. So, one can consider cutting features with variance below a certain threshold.

```
In : from sklearn.feature_selection import VarianceThreshold

In : from sklearn.datasets import make_classification

In : x_data_generated, y_data_generated = make_classification()

In : x_data_generated.shape Out: (100, 20)

In : VarianceThreshold(.7).fit_transform(x_data_generated).shape
Out: (100, 19)

In : VarianceThreshold(.8).fit_transform(x_data_generated).shape
Out: (100, 18)

In : VarianceThreshold(.9).fit_transform(x_data_generated).shape
Out: (100, 15)
```

There are other ways that are also based on classical statistics.

```
In : from sklearn.feature_selection import SelectKBest, f_classif

In : x_data_kbest = SelectKBest(f_classif,
k=5).fit_transform(x_data_generated, y_data_generated)

In : x_data_varth =
VarianceThreshold(.9).fit_transform(x_data_generated)

In : from sklearn.linear_model import LogisticRegression

In : from sklearn.model_selection import cross_val_score

In : cross_val_score(LogisticRegression(), x_data_generated,
y_data_generated, scoring='neg_log_loss').mean() Out:
-0.45367136377981693

In : cross_val_score(LogisticRegression(), x_data_kbest,
y_data_generated, scoring='neg_log_loss').mean() Out:
-0.35775228616521798

In : cross_val_score(LogisticRegression(), x_data_varth,
y_data_generated, scoring='neg_log_loss').mean() Out:
-0.44033042718359772
```

We can see that our selected features have improved the quality of the classifier. Of course, this example is purely artificial; however, it is worth using for real problems.

4.2 Grid search

Finally, we get to the most reliable method, which is also the most computationally complex: trivial grid search. Train a model on a subset of features, store results, repeat for different subsets, and compare the quality of models to identify the best feature set. This approach is called Exhaustive Feature Selection.

Searching all combinations usually takes too long, so you can try to reduce the search space. Fix a small number N, iterate through all combinations of N features, choose the best combination, and then iterate through the combinations of (N + 1) features so that the previous best combination of features is fixed and only a single new feature is considered. It is possible to iterate until we hit a maximum number of characteristics or until the quality of the model ceases to increase significantly. This algorithm is called Sequential Feature Selection.

This algorithm can be reversed: start with the complete feature space and remove fea-

tures one by one until it does not impair the quality of the model or until the desired number of features is reached.

```
In : selector = SequentialFeatureSelector(LogisticRegression(),
scoring='neg_log_loss', verbose=2, k_features=3, forward=False,
n_jobs=-1)

In : selector.fit(x_data_scaled, y_data)

In : selector.fit(x_data_scaled, y_data)

[2017-03-30 01:42:24] Features: 45/3 -- score: -0.682830838803
[2017-03-30 01:44:40] Features: 44/3 -- score: -0.682779463265
[2017-03-30 01:46:47] Features: 43/3 -- score: -0.682727480522
[2017-03-30 01:48:54] Features: 42/3 -- score: -0.682680521828
[2017-03-30 01:50:52] Features: 41/3 -- score: -0.68264297879
[2017-03-30 01:52:46] Features: 40/3 -- score: -0.682607753617
[2017-03-30 01:54:37] Features: 39/3 -- score: -0.682570678346
[2017-03-30 01:56:21] Features: 38/3 -- score: -0.682536314625
[2017-03-30 01:58:02] Features: 37/3 -- score: -0.682520258804
[2017-03-30 01:59:39] Features: 36/3 -- score: -0.68250862986
[2017-03-30 02:01:17] Features: 35/3 -- score: -0.682498213174
...
[2017-03-30 02:21:09] Features: 10/3 -- score: -0.68657335969
[2017-03-30 02:21:18] Features: 9/3 -- score: -0.688405548594
[2017-03-30 02:21:26] Features: 8/3 -- score: -0.690213724719
[2017-03-30 02:21:32] Features: 7/3 -- score: -0.692383588303
[2017-03-30 02:21:36] Features: 6/3 -- score: -0.695321584506
[2017-03-30 02:21:40] Features: 5/3 -- score: -0.698519960477
[2017-03-30 02:21:42] Features: 4/3 -- score: -0.704095390444
[2017-03-30 02:21:44] Features: 3/3 -- score: -0.713788301404

But improvement couldn't last forever
```

5 Pearson correaltion and p-value

from scipy.stats.stats import pearsonr
pearsonr$(\mathbf{x}, \mathbf{y})$
Calculates a Pearson correlation coefficient and the p-value for testing non-correlation.

The **Pearson correlation coefficient** measures the linear relationship between two datasets. Strictly speaking, Pearson's correlation requires that each dataset be normally distributed. Like other correlation coefficients, this one varies between -1 and +1 with 0

implying no correlation. Correlations of -1 or +1 imply an exact linear relationship. Positive correlations imply that as x increases, so does y. Negative correlations imply that as x increases, y decreases.

The **p-value** roughly indicates the probability of an uncorrelated system producing datasets that have a Pearson correlation at least as extreme as the one computed from these datasets. The p-values are not entirely reliable but are probably reasonable for datasets larger than 500 or so.

```
import numpy as np
from scipy.stats import pearsonr
x = np.array([ 58295.62187335,  45420.95483714,
    3398.64920064,    977.22166306, 5515.32801851,
    14184.57621022,  16027.2803392 ,  15313.01865824,
    6443.2448182 ])
y = np.array([ 143547.79123381,   22996.69597427,
    2591.56411049, 661.93115277,    8826.96549102,
    17735.13549851, 11629.13003263,   14438.33177173,
      6997.89334741])

r,p = pearsonr(x,y)
print( "r", r)
print ("p-value", p)
```

The output is as follows:

Returns: **r** : 0.831087956392

Pearson's correlation coefficient

p-value : 0.00550539621039

2-tailed p-value