



## mlcourse.ai – Open Machine Learning Course

Author: [Arseny Kravchenko](#). Translated and edited by [Christina Butsko](#), [Egor Polusmak](#), [Anastasia Manokhina](#), [Anna Larionova](#), [Evgeny Sushko](#) and [Yuanyuan Pao](#). This material is subject to the terms and conditions of the [Creative Commons CC BY-NC-SA 4.0](#) license. Free use is permitted for any non-commercial purpose.

## Topic 6. Feature Engineering and Feature Selection

In this course, we have already seen several key machine learning algorithms. However, before moving on to the more fancy ones, we'd like to take a small detour and talk about data preparation. The well-known concept of “garbage in—garbage 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, I wanted to 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. Some examples will use the dataset from Renthop company, which is used in the [Two Sigma Connect: Rental Listing Inquiries Kaggle competition](#). The file `train.json` is also kept [here](#) as `renthop_train.json`. In this task, you need to predict the popularity of a new rental listing, i.e. classify the listing into three classes: `['low', 'medium', 'high']`. To evaluate the solutions, we will use the log loss metric (the smaller, the better). Those who do not have a Kaggle account, will have to register; you will also need to accept the rules of the competition in order to download the data.

In [1]:

```
import json
import pandas as pd

# Let's load the dataset from Renthop right away
with open('../data/renthop_train.json', 'r') as raw_data:
    data = json.load(raw_data)
    df = pd.DataFrame(data)
```

## Article outline

1. Feature Extraction
  1. Texts
  2. Images
  3. Geospatial data
  4. Date and time
  5. Time series, web, etc.
2. Feature transformations
  1. Normalization and changing distribution
  2. Interactions
  3. Filling in the missing values
3. Feature selection
  1. Statistical approaches
  2. Selection by modeling
  3. Grid search

## 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.

### 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:

In [2]:

```
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']]

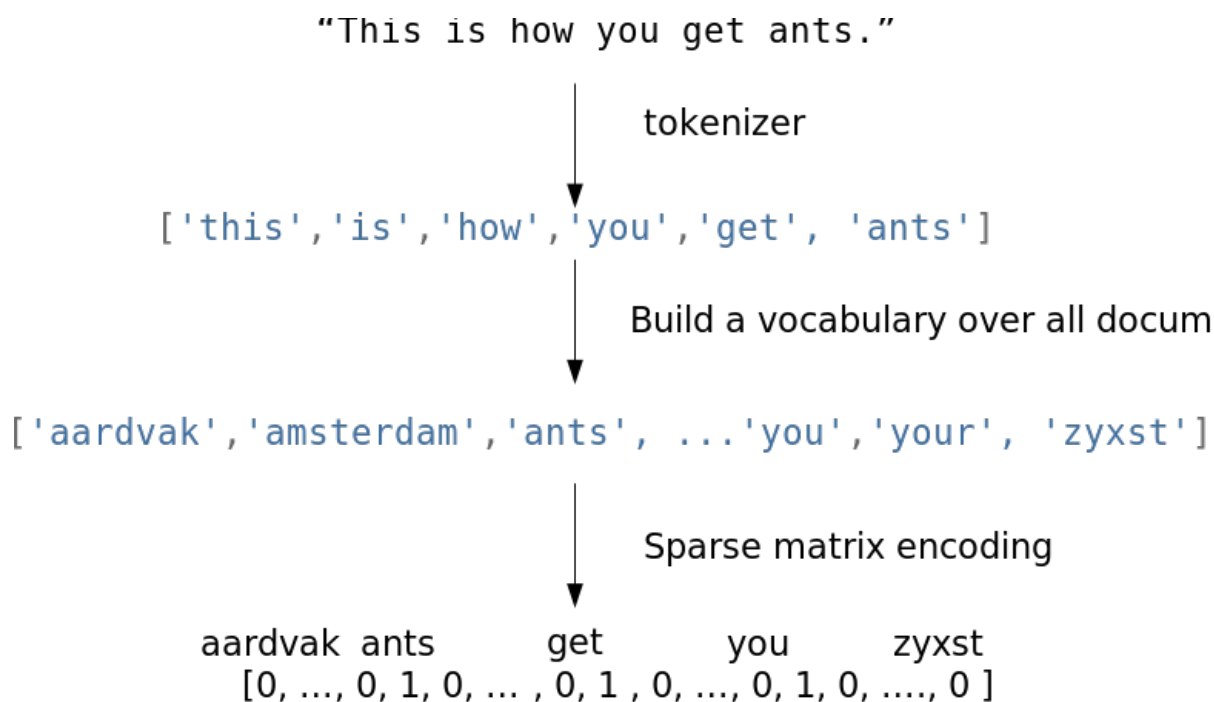
dictionary = list(enumerate(set(list(reduce(lambda x, y: x + y, texts)))))

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))
```

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

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 [3]:

```
from sklearn.feature_extraction.text import CountVectorizer

vect = CountVectorizer(ngram_range=(1,1))
vect.fit_transform(['no i have cows', 'i have no cows']).toarray()
```

Out[3]:

```
array([[1, 1, 1],
       [1, 1, 1]], dtype=int64)
```

In [4]:

```
vect.vocabulary_
```

Out[4]:

```
{'no': 2, 'have': 1, 'cows': 0}
```

In [5]:

```
vect = CountVectorizer(ngram_range=(1,2))
vect.fit_transform(['no i have cows', 'i have no cows']).toarray()
```

Out[5]:

```
array([[1, 1, 1, 0, 1, 0, 1],
       [1, 1, 0, 1, 1, 1, 0]], dtype=int64)
```

In [6]:

```
vect.vocabulary_
```

Out[6]:

```
{'no': 4,
 'have': 1,
 'cows': 0,
 'no have': 6,
 'have cows': 2,
 'have no': 3,
 'no cows': 5}
```

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.

In [7]:

```
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)
```

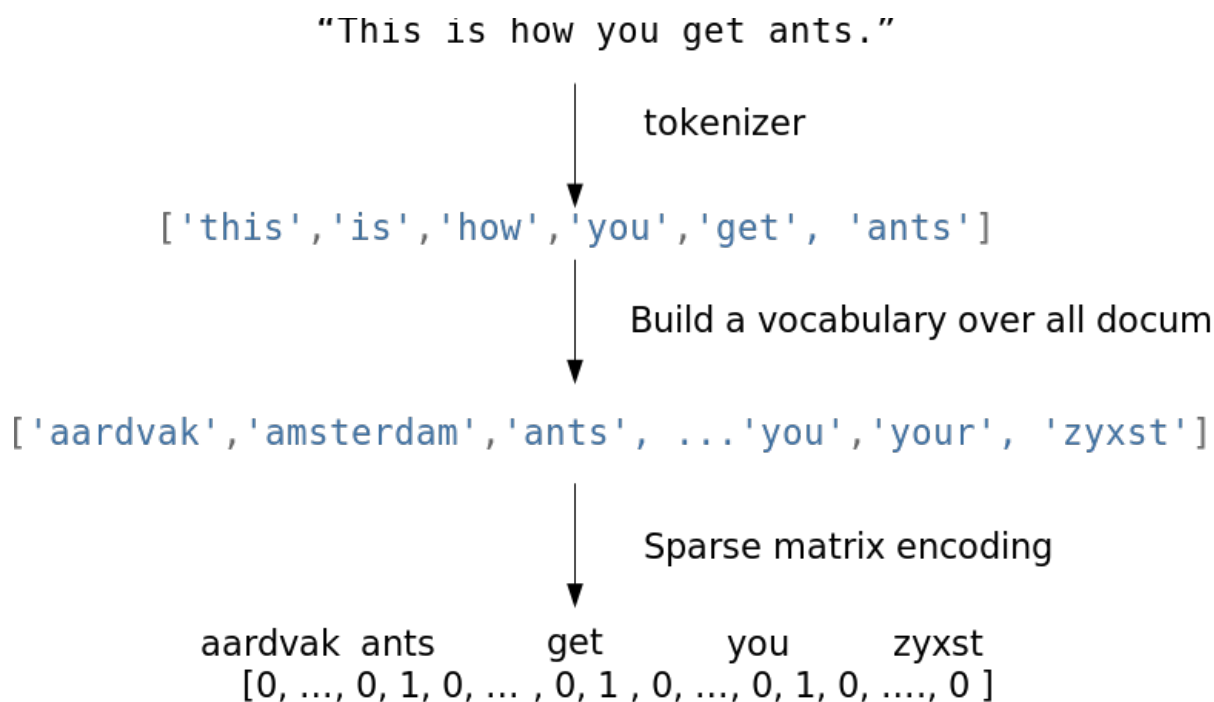
Out[7]:

```
(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), which cannot be written in a few lines, so you should look into the details in references such as [this wiki](#). The default option is as follows:

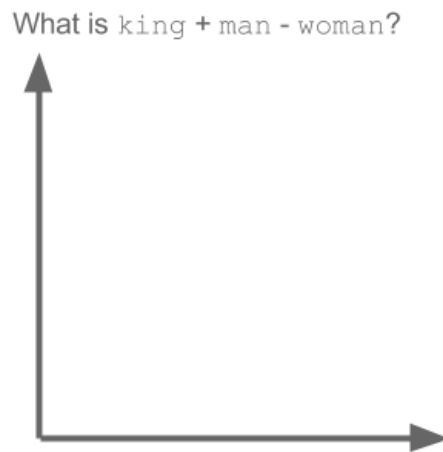
$$idf(t, D) = \log \frac{|D|}{df(d, t) + 1}$$
$$tfidf(t, d, D) = tf(t, d) \times idf(t, D)$$

Analogous of Bag of Words can be found outside of text problems e.g. bag of sites in the [Catch Me If You Can competition](#), [bag of apps](#), [bag of events](#), etc.



Using these algorithms, it is possible to obtain a working solution for a simple problem, which can serve as a baseline. However, for those who do not like the classics, there are new approaches. The most popular method in the new wave is Word2Vec, but there are a few alternatives as well (GloVe, Fasttext, etc.).

Word2Vec is a special case of the word embedding algorithms. Using Word2Vec and similar models, we can not only vectorize words in a high-dimensional space (typically a few hundred dimensions) but also compare their semantic similarity. This is a classic example of operations that can be performed on vectorized concepts: king - man + woman = queen.



It is worth noting that this model does not comprehend the meaning of the words but simply tries to position the vectors such that words used in common context are close to each other. If this is not taken into account, a lot of fun examples will come up.

Such models need to be trained on very large datasets in order for the vector coordinates to capture the semantics. A pretrained model for your own tasks can be downloaded [here](#).

Similar methods are applied in other areas such as bioinformatics. An unexpected application is [food2vec](#). You can probably think of a few other fresh ideas; the concept is universal enough.

## Images

Working with images is easier and harder at the same time. It is easier because it is possible to just use one of the popular pretrained networks without much thinking but harder because, if you need to dig into the details, you may end up going really deep. Let's start from the beginning.

In a time when GPUs were weaker and the "renaissance of neural networks" had not happened yet, feature generation from images was its own complex field. One had to work at a low level, determining corners, borders of regions, color distributions statistics, and so on. Experienced specialists in computer vision could draw a lot of parallels between older approaches and neural networks; in particular, convolutional layers in today's networks are similar to [Haar cascades](#). If you are interested in reading more, here are a couple of links to some interesting libraries: [skimage](#) and [SimpleCV](#).

Often for problems associated with images, a convolutional neural network is used. You do not have to come up with the architecture and train a network from scratch. Instead, download a pretrained state-of-the-art network with the weights from public sources. Data scientists often do so-called fine-tuning to adapt these networks to their needs by "detaching" the last fully connected layers of the network, adding new layers chosen for a specific task, and then training the network on new data. If your task is to just vectorize the image (for example, to use some non-network classifier), you only need to remove the last layers and use the output from the previous layers:

In [8]:

```
# doesn't work with Python 3.7
## Install Keras and tensorflow (https://keras.io/)
# from keras.applications.resnet50 import ResNet50, preprocess_input
# from keras.preprocessing import image
# from scipy.misc import face
# import numpy as np

# resnet_settings = {'include_top': False, 'weights': 'imagenet'}
# resnet = ResNet50(**resnet_settings)

## What a cute raccoon!
# img = image.array_to_img(face())
# img
```

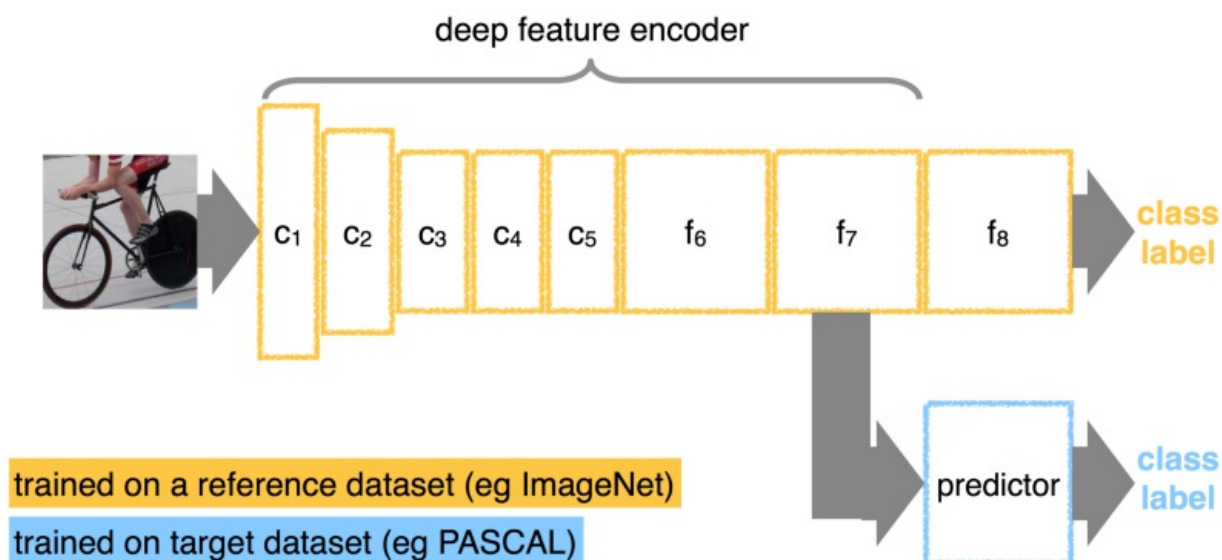
In [9]:

```
## In real life, you may need to pay more attention to resizing
# img = img.resize((224, 224))

# x = image.img_to_array(img)
# x = np.expand_dims(x, axis=0)
# x = preprocess_input(x)

## Need an extra dimension because model is designed to work with an array
## of images - i.e. tensor shaped (batch_size, width, height, n_channels)

# features = resnet.predict(x)
```



Here's a classifier trained on one dataset and adapted for a different one by "detaching" the last layer and adding a new one instead.

Nevertheless, we should not focus too much on neural network techniques. Features generated by hand are still very useful: for example, for predicting the popularity of a rental listing, we can assume that bright apartments attract more attention and create a feature such as "the average value of the pixel". You can find some inspiring examples in the documentation of [relevant libraries](#).

If there is text on the image, you can read it without unraveling a complicated neural network. For example, check out [pytesseract](#).

```
import pytesseract
from PIL import Image
import requests
from io import BytesIO

##### Just a random picture from search
img = 'http://ohscurrent.org/wp-content/uploads/2015/09/domus-01-google.jpg'

img = requests.get(img)
img = Image.open(BytesIO(img.content))
text = pytesseract.image_to_string(img)

text

Out: 'Google'
```

One must understand that `pytesseract` is not a solution for everything.

```
##### This time we take a picture from Renthop
```

```
img = requests.get('https://photos.renthop.com/2/8393298_6acaf11f030217d05f3a5604b9a2f70f.jpg')  
img = Image.open(BytesIO(img.content))  
pytesseract.image_to_string(img)
```

```
Out: 'Cunveztible to 4}»'
```

Another case where neural networks cannot help is extracting features from meta-information. For images, EXIF stores many useful meta-information: manufacturer and camera model, resolution, use of the flash, geographic coordinates of shooting, software used to process image and more.

## Geospatial data

Geographic data is not so often found in problems, but it is still useful to master the basic techniques for working with it, especially since there are quite a number of ready-to-use solutions in this field.

Geospatial data is often presented in the form of addresses or coordinates of (Latitude, Longitude). Depending on the task, you may need two mutually-inverse operations: geocoding (recovering a point from an address) and reverse geocoding (recovering an address from a point). Both operations are accessible in practice via external APIs from Google Maps or OpenStreetMap. Different geocoders have their own characteristics, and the quality varies from region to region. Fortunately, there are universal libraries like [geopy](#) that act as wrappers for these external services.

If you have a lot of data, you will quickly reach the limits of external API. Besides, it is not always the fastest to receive information via HTTP. Therefore, it is necessary to consider using a local version of OpenStreetMap.

If you have a small amount of data, enough time, and no desire to extract fancy features, you can use `reverse_geocoder` in lieu of OpenStreetMap:

```
import reverse_geocoder as revgc
```

```
revgc.search((df.latitude, df.longitude))  
Loading formatted geocoded file...
```

```
Out: [OrderedDict([('lat', '40.74482'),  
                  ('lon', '-73.94875'),  
                  ('name', 'Long Island City'),  
                  ('admin1', 'New York'),  
                  ('admin2', 'Queens County'),  
                  ('cc', 'US')])]
```

When working with geocoding, we must not forget that addresses may contain typos, which makes the data cleaning step necessary. Coordinates contain fewer misprints, but its position can be incorrect due to GPS noise or bad accuracy in places like tunnels, downtown areas, etc. If the data source is a mobile device, the geolocation may not be determined by GPS but by WiFi networks in the area, which leads to holes in space and teleportation. While traveling along in Manhattan, there can suddenly be a WiFi location from Chicago.

WiFi location tracking is based on the combination of SSID and MAC-addresses, which may correspond to different points e.g. federal provider standardizes the firmware of routers up to MAC-address and places them in different cities. Even a company's move to another office with its routers can cause issues.

The point is usually located among infrastructure. Here, you can really unleash your imagination and invent features based on your life experience and domain knowledge: the proximity of a point to the subway, the number of stories in the building, the distance to the nearest store, the number of ATMs around, etc. For any task, you can easily come up with dozens of features and extract them from various external sources. For problems outside an urban environment, you may consider features from more specific sources e.g. the height above sea level.

If two or more points are interconnected, it may be worthwhile to extract features from the route between them. In that case, distances (great circle distance and road distance calculated by the routing graph), number of turns with the ratio of left to right turns, number of traffic lights, junctions, and bridges will be useful. In one of my own tasks, I generated a feature called "the complexity of the road", which computed the graph-calculated distance divided by the GCD.

## Date and time

You would think that date and time are standardized because of their prevalence, but, nevertheless, some pitfalls remain.

Let's start with the day of the week, which are easy to turn into 7 dummy variables using one-hot encoding. In addition, we will also create a separate binary feature for the weekend called `is_weekend`.

```
df['dow'] = df['created'].apply(lambda x: x.date().weekday())
df['is_weekend'] = df['created'].apply(lambda x: 1 if x.date().weekday() in (5, 6) else 0)
```

Some tasks may require additional calendar features. For example, cash withdrawals can be linked to a pay day; the purchase of a metro card, to the beginning of the month. In general, when working with time series data, it is a good idea to have a calendar with public holidays, abnormal weather conditions, and other important events.

Q: What do Chinese New Year, the New York marathon, and the Trump inauguration have in common?

A: They all need to be put on the calendar of potential anomalies.

Dealing with hour (minute, day of the month ...) is not as simple as it seems. If you use the hour as a real variable, we slightly contradict the nature of data: `0<23` while `0:00:00 02.01> 01.01 23:00:00`. For some problems, this can be critical. At the same time, if you encode them as categorical variables, you'll breed a large numbers of features and lose information about proximity -- the difference between 22 and 23 will be the same as the difference between 22 and 7.

There also exist some more esoteric approaches to such data like projecting the time onto a circle and using the two coordinates.

In [10]:

```
def make_harmonic_features(value, period=24):
    value *= 2 * np.pi / period
    return np.cos(value), np.sin(value)
```

This transformation preserves the distance between points, which is important for algorithms that estimate distance (kNN, SVM, k-means ...)

In [11]:

```
from scipy.spatial import distance
euclidean(make_harmonic_features(23), make_harmonic_features(1))
```

Out[11]:

0.5176380902050424

In [12]:

```
euclidean(make_harmonic_features(9), make_harmonic_features(11))
```

Out[12]:

0.5176380902050414

In [13]:

```
euclidean(make_harmonic_features(9), make_harmonic_features(21))
```

Out[13]:

2.0

However, the difference between such coding methods is down to the third decimal place in the metric.

## Time series, web, etc.

Regarding time series—we will not go into too much detail here (mostly due to my personal lack of experience), but I will point you to a [useful library that automatically generates features for time series](#).

If you are working with web data, then you usually have information about the user's User Agent. It is a wealth of information. First, one needs to extract the operating system from it. Secondly, make a feature `is_mobile`. Third, look at the browser.



In [14]:

```
# Install pyyaml ua-parser user-agents
```

```
import user_agents
```

```
ua = 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Ubuntu Chromium/56.0.2924.76 Chrome/56.0.2924.76 Safari/537.36'
```

```
ua = user_agents.parse(ua)
```

```
print('Is a bot? ', ua.is_bot)
```

```
print('Is mobile? ', ua.is_mobile)
```

```
print('Is PC? ', ua.is_pc)
```

```
print('OS Family: ', ua.os.family)
```

```
print('OS Version: ', ua.os.version)
```

```
print('Browser Family: ', ua.browser.family)
```

```
print('Browser Version: ', ua.browser.version)
```

```
Is a bot? False
```

```
Is mobile? False
```

```
Is PC? True
```

```
OS Family: Ubuntu
```

```
OS Version: ()
```

```
Browser Family: Chromium
```

```
Browser Version: (56, 0, 2924)
```

As in other domains, you can come up with your own features based on intuition about the nature of the data. At the time of this writing, Chromium 56 was new, but, after some time, only users who haven't rebooted their browser for a long time will have this version. In this case, why not introduce a feature called "lag behind the latest version of the browser"?

In addition to the operating system and browser, you can look at the referrer (not always available), [http\\_accept\\_language](#), and other meta information.

The next useful piece of information is the IP-address, from which you can extract the country and possibly the city, provider, and connection type (mobile/stationary). You need to understand that there is a variety of proxy and outdated databases, so this feature can contain noise. Network administration gurus may try to extract even fancier features like suggestions for [using VPN](#). By the way, the data from the IP-address is well combined with [http\\_accept\\_language](#) : if the user is sitting at the Chilean proxies and browser locale is `ru_RU` , something is unclear and worth a look in the corresponding column in the table ( `is_traveler_or_proxy_user` ).

Any given area has so many specifics that it is too much for an individual to absorb completely. Therefore, I invite everyone to share their experiences and discuss feature extraction and generation in the comments section.

## Feature transformations

### 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 variables—the 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 [15]:

```
from sklearn.preprocessing import StandardScaler
from scipy.stats import beta
from scipy.stats import shapiro
import numpy as np
```

```
data = beta(1, 10).rvs(1000).reshape(-1, 1)
shapiro(data)
```

Out[15]:

```
(0.8499014377593994, 9.379595554363724e-30)
```

In [16]:

```
# Value of the statistic, p-value
shapiro(StandardScaler().fit_transform(data))

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

Out[16]:

```
(0.8499014377593994, 9.379595554363724e-30)
```

But, to some extent, it protects against outliers:

In [17]:

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

Out[17]:

```
array([[ -0.31922662],
       [-0.31922662],
       [-0.35434155],
       [-0.38945648],
       [-0.28411169],
       [-0.31922662],
       [-0.28411169],
       [-0.24899676],
       [-0.42457141],
       [-0.21388184],
       [ 3.15715128]])
```

In [18]:

```
(data - data.mean()) / data.std()
```

Out[18]:

```
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 [19]:

```
from sklearn.preprocessing import MinMaxScaler

MinMaxScaler().fit_transform(data)
```

Out[19]:

```
array([[0.02941176],
       [0.02941176],
       [0.01960784],
       [0.00980392],
       [0.03921569],
       [0.02941176],
       [0.03921569],
       [0.04901961],
       [0.      ],
       [0.05882353],
       [1.      ]])
```

In [20]:

```
(data - data.min()) / (data.max() - data.min())
```

Out[20]:

```
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 [21]:

```
from scipy.stats import lognorm

data = lognorm(s=1).rvs(1000)
shapiro(data)
```

Out[21]:

```
(0.584032416343689, 1.401298464324817e-43)
```

In [22]:

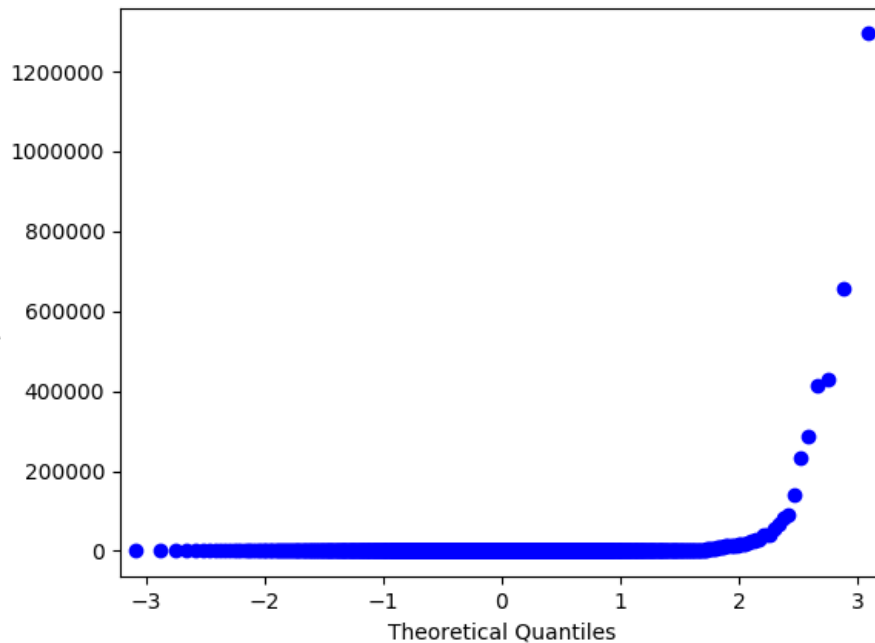
```
shapiro(np.log(data))
```

Out[22]:

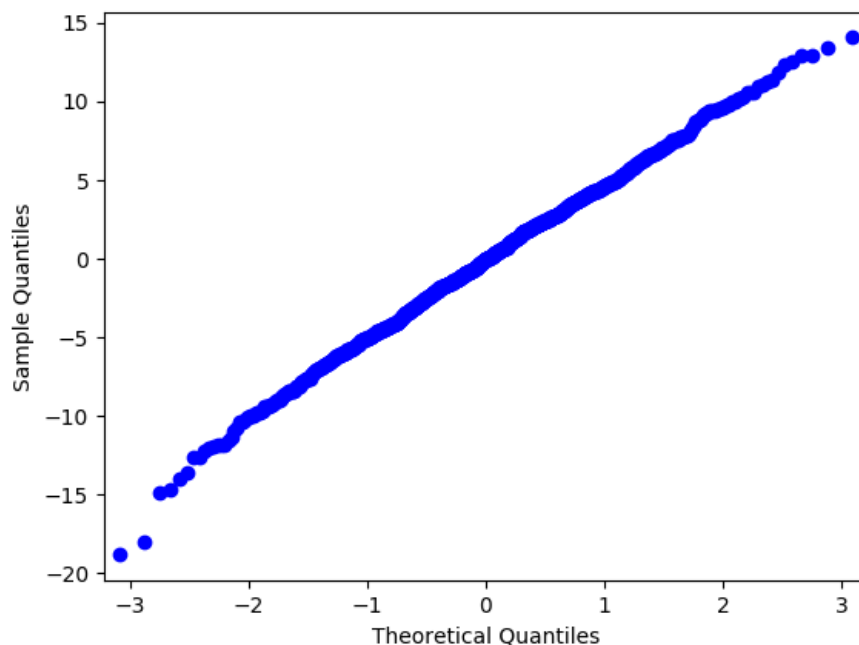
```
(0.9983807802200317, 0.4792686402797699)
```

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. Examples of such transformations are [Box-Cox transformation](#) (logarithm is a special case of the Box-Cox transformation) or [Yeo-Johnson transformation](#) (extends the range of applicability to negative numbers). 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 method—[Q-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

In [23]:

```
# Let's draw plots!
import statsmodels.api as sm

# Let's take the price feature from Renthop dataset and filter by hands the most extreme values for clarity

price = df.price[(df.price <= 20000) & (df.price > 500)]
price_log = np.log(price)

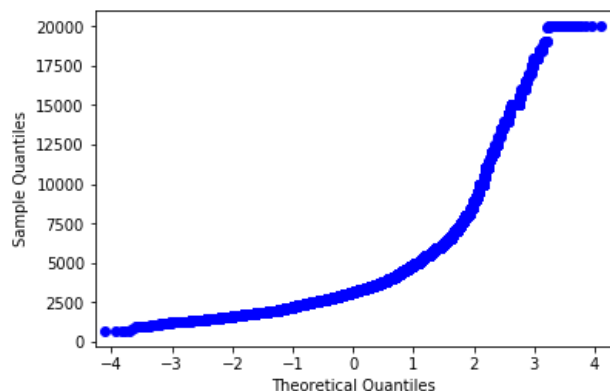
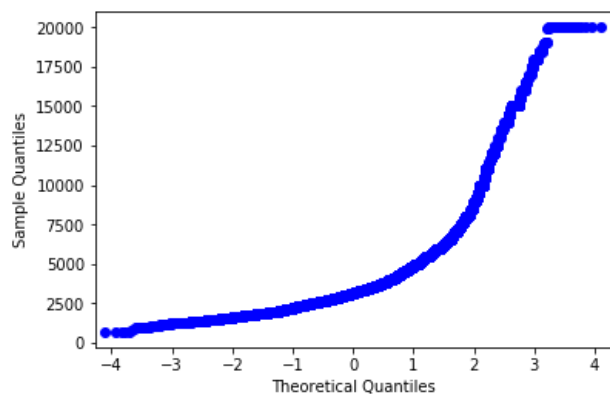
# A lot of gestures so that sklearn didn't shower us with warnings
price_mm = MinMaxScaler().fit_transform(price.values.reshape(-1, 1).astype(np.float64)).flatten()
price_z = StandardScaler().fit_transform(price.values.reshape(-1, 1).astype(np.float64)).flatten()
```

Q-Q plot of the initial feature

In [24]:

```
sm.qqplot(price, loc=price.mean(), scale=price.std())
```

Out[24]:

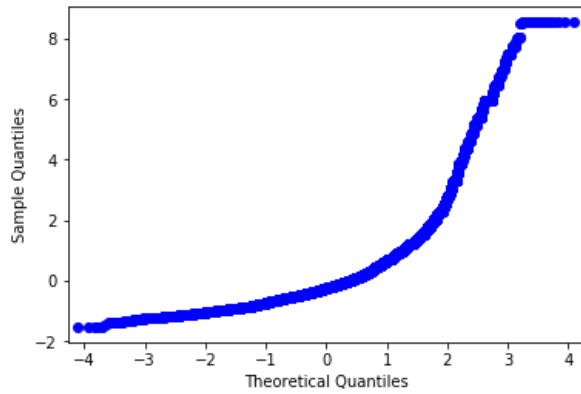
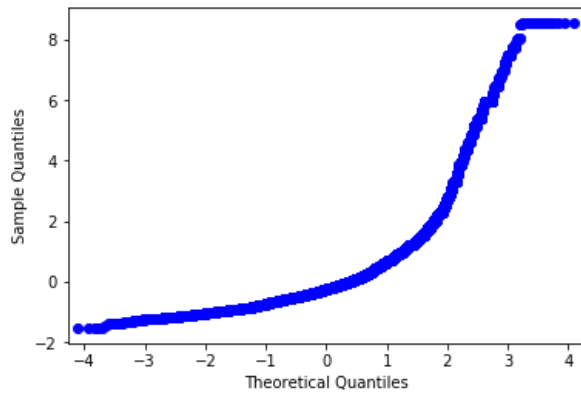


Q-Q plot after StandardScaler. Shape doesn't change

In [25]:

```
sm.qqplot(price_z, loc=price_z.mean(), scale=price_z.std())
```

Out[25]:

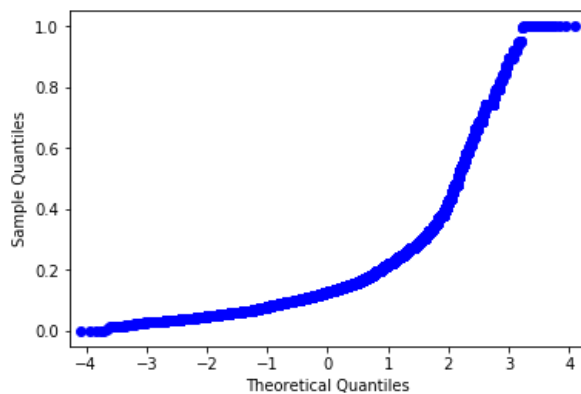
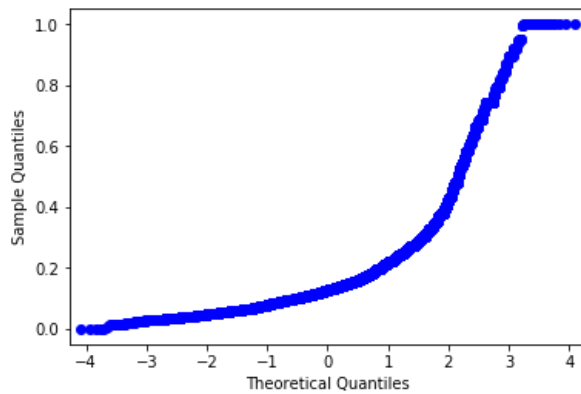


Q-Q plot after MinMaxScaler. Shape doesn't change

In [26]:

```
sm.qqplot(price_mm, loc=price_mm.mean(), scale=price_mm.std())
```

Out[26]:

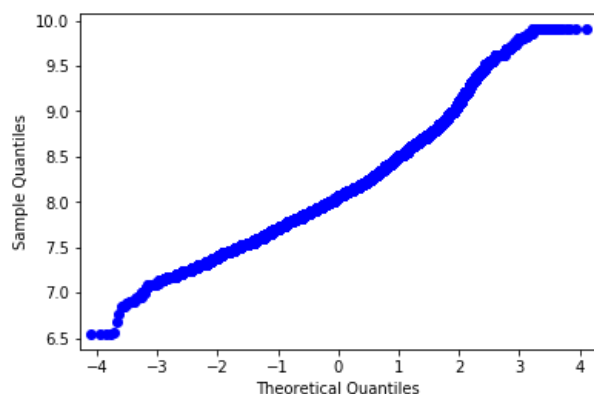
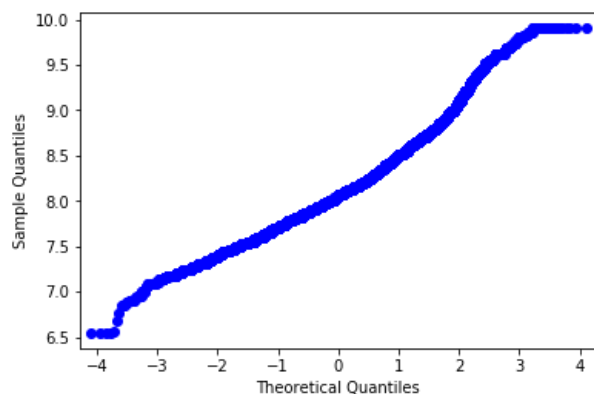


Q-Q plot after taking the logarithm. Things are getting better!

In [27]:

```
sm.qqplot(price_log, loc=price_log.mean(), scale=price_log.std())
```

Out[27]:



Let's see whether transformations can somehow help the real model. There is no silver bullet here.

## Interactions

If previous transformations seemed rather math-driven, this part is more about the nature of the data; it can be attributed to both feature transformations and feature creation.

Let's come back again to the Two Sigma Connect: Rental Listing Inquiries problem. Among the features in this problem are the number of rooms and the price. Logic suggests that the cost per single room is more indicative than the total cost, so we can generate such a feature.

In [28]:

```
rooms = df["bedrooms"].apply(lambda x: max(x, .5))  
# Avoid division by zero; .5 is chosen more or less arbitrarily  
df["price_per_bedroom"] = df["price"] / rooms
```

You should limit yourself in this process. If there are a limited number of features, it is possible to generate all the possible interactions and then weed out the unnecessary ones using the techniques described in the next section. In addition, not all interactions between features must have a physical meaning; for example, polynomial features (see [sklearn.preprocessing.PolynomialFeatures](#)) are often used in linear models and are almost impossible to interpret.

## 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 value—next or previous.

The difference between learning theory and practice is that, in theory, they converge but, in practice, "ValueError: invalid literal 'NaN'".



Easy-to-use library solutions sometimes suggest sticking to something like `df = df.fillna(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.

## Feature selection

Why would it even be necessary to select features? To some, this idea may seem counterintuitive, 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.

## 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 [29]:

```
from sklearn.feature_selection import VarianceThreshold
from sklearn.datasets import make_classification

x_data_generated, y_data_generated = make_classification()
x_data_generated.shape
```

Out[29]:

(100, 20)

In [30]:

```
VarianceThreshold(.7).fit_transform(x_data_generated).shape
```

Out[30]:

(100, 20)

In [31]:

```
VarianceThreshold(.8).fit_transform(x_data_generated).shape
```

Out[31]:

(100, 20)



In [32]:

```
VarianceThreshold(.9).fit_transform(x_data_generated).shape
```

Out[32]:

```
(100, 15)
```

There are other ways that are also [based on classical statistics](#).

In [33]:

```
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

x_data_kbest = SelectKBest(f_classif, k=5).fit_transform(x_data_generated, y_data_generated)
x_data_varth = VarianceThreshold(.9).fit_transform(x_data_generated)
```

In [34]:

```
logit = LogisticRegression(solver='lbfgs', random_state=17)
```

In [35]:

```
cross_val_score(logit, x_data_generated, y_data_generated,
                 scoring='neg_log_loss', cv=5).mean()
```

Out[35]:

```
-0.6601651812409639
```

In [36]:

```
cross_val_score(logit, x_data_kbest, y_data_generated,
                 scoring='neg_log_loss', cv=5).mean()
```

Out[36]:

```
-0.4490370233466178
```

In [37]:

```
cross_val_score(logit, x_data_varth, y_data_generated,
                 scoring='neg_log_loss', cv=5).mean()
```

Out[37]:

```
-0.5604474179491241
```

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.

## Selection by modeling

Another approach is to use some baseline model for feature evaluation because the model will clearly show the importance of the features. Two types of models are usually used: some “wooden” composition such as [Random Forest](#) or a linear model with Lasso regularization so that it is prone to nullify weights of weak features. The logic is intuitive: if features are clearly useless in a simple model, there is no need to drag them to a more complex one.

In [38]:

```
# Synthetic example

from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.model_selection import cross_val_score
from sklearn.pipeline import make_pipeline

x_data_generated, y_data_generated = make_classification()

rf = RandomForestClassifier(n_estimators=10, random_state=17)
pipe = make_pipeline(SelectFromModel(estimator=rf), logit)

print(cross_val_score(logit, x_data_generated, y_data_generated,
                      scoring='neg_log_loss', cv=5).mean())
print(cross_val_score(rf, x_data_generated, y_data_generated,
                      scoring='neg_log_loss', cv=5).mean())
print(cross_val_score(pipe, x_data_generated, y_data_generated,
                      scoring='neg_log_loss', cv=5).mean())
```

```
-0.2009514079653809
-0.6138115414016354
-0.1512133279261138
```

We must not forget that this is not a silver bullet again-it can make the performance worse.

In [39]:

```
#x_data, y_data = get_data()
x_data = x_data_generated
y_data = y_data_generated

pipe1 = make_pipeline(StandardScaler(),
                      SelectFromModel(estimator=rf), logit)

pipe2 = make_pipeline(StandardScaler(), logit)

print('LR + selection: ', cross_val_score(pipe1, x_data, y_data,
                      scoring='neg_log_loss', cv=5).mean())
print('LR: ', cross_val_score(pipe2, x_data, y_data,
                      scoring='neg_log_loss', cv=5).mean())
print('RF: ', cross_val_score(rf, x_data, y_data,
                      scoring='neg_log_loss', cv=5).mean())
```

```
LR + selection: -0.1651673258448792
LR: -0.21646446021272087
RF: -0.6138115414016354
```

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

In [40]:

```
# Install mlxtend
from mlxtend.feature_selection import SequentialFeatureSelector

selector = SequentialFeatureSelector(logit, scoring='neg_log_loss',
                                   verbose=2, k_features=3, forward=False, n_jobs=-1)

selector.fit(x_data, y_data)

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done 8 out of 20 | elapsed: 1.2s remaining: 1.7s
```

[Parallel(n\_jobs=-1)]: Done 20 out of 20 | elapsed: 1.3s finished

[2018-11-09 17:22:35] Features: 19/3 -- score: -0.1873509505302199[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 6 out of 19 | elapsed: 0.0s remaining: 0.1s  
[Parallel(n\_jobs=-1)]: Done 16 out of 19 | elapsed: 0.1s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 19 out of 19 | elapsed: 0.1s finished

[2018-11-09 17:22:36] Features: 18/3 -- score: -0.17774491860817282[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 5 out of 18 | elapsed: 0.0s remaining: 0.1s  
[Parallel(n\_jobs=-1)]: Done 15 out of 18 | elapsed: 0.1s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 18 out of 18 | elapsed: 0.1s finished

[2018-11-09 17:22:36] Features: 17/3 -- score: -0.17138833166292974[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 3 out of 17 | elapsed: 0.0s remaining: 0.2s  
[Parallel(n\_jobs=-1)]: Done 12 out of 17 | elapsed: 0.1s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 17 out of 17 | elapsed: 0.1s finished

[2018-11-09 17:22:36] Features: 16/3 -- score: -0.16439505331749654[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 2 out of 16 | elapsed: 0.0s remaining: 0.3s  
[Parallel(n\_jobs=-1)]: Done 11 out of 16 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 16 out of 16 | elapsed: 0.1s finished

[2018-11-09 17:22:36] Features: 15/3 -- score: -0.15902316605600558[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 8 out of 15 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 15 out of 15 | elapsed: 0.0s finished

[2018-11-09 17:22:36] Features: 14/3 -- score: -0.1563135277001207[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 7 out of 14 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 14 out of 14 | elapsed: 0.0s finished

[2018-11-09 17:22:36] Features: 13/3 -- score: -0.1533863692367438[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 4 out of 13 | elapsed: 0.0s remaining: 0.1s  
[Parallel(n\_jobs=-1)]: Done 11 out of 13 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 13 out of 13 | elapsed: 0.0s finished

[2018-11-09 17:22:36] Features: 12/3 -- score: -0.15049956265937406[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 3 out of 12 | elapsed: 0.1s remaining: 0.2s  
[Parallel(n\_jobs=-1)]: Done 10 out of 12 | elapsed: 0.1s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 12 out of 12 | elapsed: 0.1s finished

[2018-11-09 17:22:36] Features: 11/3 -- score: -0.1485795105902746[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 6 out of 11 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 11 out of 11 | elapsed: 0.0s finished

[2018-11-09 17:22:36] Features: 10/3 -- score: -0.14499979151449888[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 5 out of 10 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 10 out of 10 | elapsed: 0.0s finished

[2018-11-09 17:22:36] Features: 9/3 -- score: -0.1438340298394421[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 6 out of 9 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 9 out of 9 | elapsed: 0.0s finished

[2018-11-09 17:22:36] Features: 8/3 -- score: -0.14369708753126084[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 5 out of 8 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 8 out of 8 | elapsed: 0.0s finished

[2018-11-09 17:22:36] Features: 7/3 -- score: -0.1430932673849256[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 4 out of 7 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 7 out of 7 | elapsed: 0.0s finished

[2018-11-09 17:22:36] Features: 6/3 -- score: -0.14334647052366745[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 3 out of 6 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 6 out of 6 | elapsed: 0.0s finished

[2018-11-09 17:22:36] Features: 5/3 -- score: -0.13983516375007882[Parallel(n\_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.  
[Parallel(n\_jobs=-1)]: Done 2 out of 5 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n\_jobs=-1)]: Done 5 out of 5 | elapsed: 0.0s finished

```
[Parallel(n_jobs=-1)]: Done 3 out of 5 | elapsed: 0.0s remaining: 0.0s  
[Parallel(n_jobs=-1)]: Done 5 out of 5 | elapsed: 0.0s finished
```

```
[2018-11-09 17:22:36] Features: 4/3 -- score: -0.14186925844563356[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
```

```
[Parallel(n_jobs=-1)]: Done 2 out of 4 | elapsed: 0.0s remaining: 0.0s
```

```
[Parallel(n_jobs=-1)]: Done 4 out of 4 | elapsed: 0.0s finished
```

```
[2018-11-09 17:22:36] Features: 3/3 -- score: -0.14433923917553568
```

Out[40]:

```
SequentialFeatureSelector(clone_estimator=True, cv=5,  
    estimator=LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,  
    intercept_scaling=1, max_iter=100, multi_class='warn',  
    n_jobs=None, penalty='l2', random_state=17, solver='lbfgs',  
    tol=0.0001, verbose=0, warm_start=False),  
    floating=False, forward=False, k_features=3, n_jobs=-1,  
    pre_dispatch='2*n_jobs', scoring='neg_log_loss', verbose=2)
```

Take a look how this approach was done in one [simple yet elegant Kaggle kernel](#).

This website does not host notebooks, it only renders notebooks available on other websites.

Delivered by Fastly, Rendered by Rackspace

nbviewer GitHub repository.

nbviewer version: aa567da

nbconvert version: 5.3.1

Rendered 5 minutes ago