In [1]:
```
%matplotlib inline
import matplotlib
import seaborn as sns
sns.set()
matplotlib.rcParams['figure.dpi'] = 144
```

In [2]:
```
import numpy as np
```

Transformers and Preprocessing

Raw data is often unsuitable for direct use by Machine Learning algorithms. This is most obvious when data is not in the format that the algorithm expects. For example, suppose we are trying to use Linear Regression on the following:

$$X_{raw} = \begin{bmatrix} \text{left} & 3 \\ \text{right} & 1 \\ \text{left} & 2 \\ \text{right} & 5 \\ \text{right} & 4 \end{bmatrix} \qquad y = \begin{bmatrix} 5 \\ 7 \\ 4 \\ 11 \\ 10 \end{bmatrix}$$

We can't do numerical computations with strings, so it's impossible to apply the algorithm directly, but we can use an encoding (`left=0`, `right=1`) to represent each string as a number. This yields a transformed data set which can be fit by the algorithm.

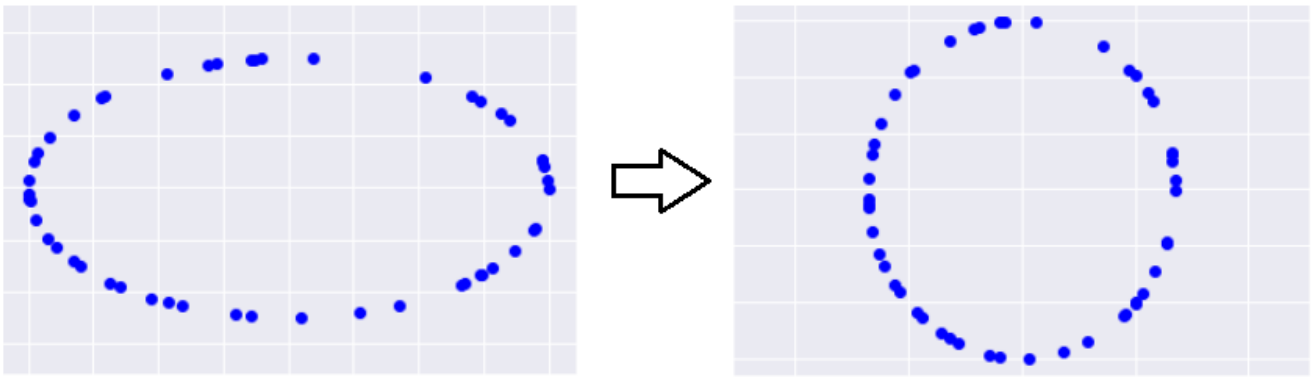
$$X = \begin{bmatrix} 0 & 3 \\ 1 & 1 \\ 0 & 2 \\ 1 & 5 \\ 1 & 4 \end{bmatrix} \qquad y = \begin{bmatrix} 5 \\ 7 \\ 4 \\ 11 \\ 10 \end{bmatrix}$$

In this case, the model $y = 4x_1 + x_2 + 2$ is a perfect fit for the data. We just need to remember that `left` and `right` mean $x_1 = 0$ and $x_1 = 1$ respectively when the time comes to make predictions for new data.

Preprocessing is a general term for transformation steps like this and **transformers** are objects that we use to learn and apply transformations.

- Preprocessing can improve the performance of algorithms even when data begins in a valid format.
- Usually a transformer is more than just a single function.

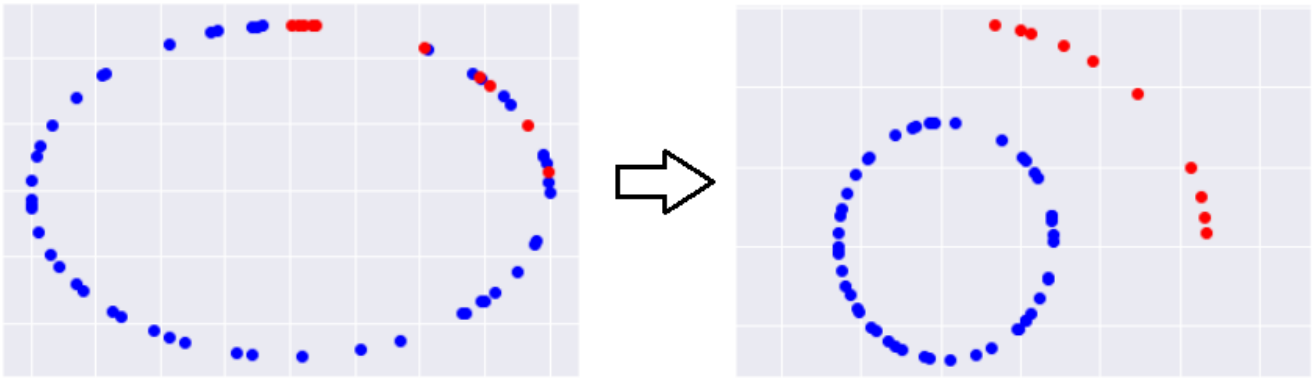
For example, suppose that our data has two continuous features. The first varies from 0 to 10 and the second varies from 0 to $10,000$. Distance based algorithms (like K-Nearest Neighbors) will neglect the first feature because of its smaller scale and make predictions that are almost exclusively based on the latter. If we want to remove this bias, then we should normalize the data so that both features are on a comparable scale.



A simple recipe for normalization is to divide each feature by its standard deviation. It seems natural to express this as a function:

$$\text{norm}(X) = \frac{X}{\sigma_X}$$

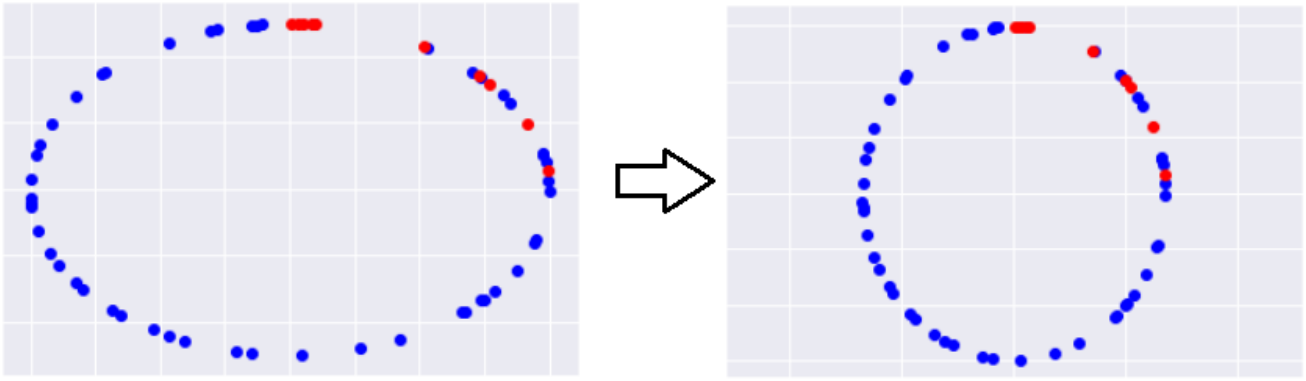
But what happens when we apply this function to a second data set?



The second set is scaled differently because its standard deviations are different from the first set. If you imagine that the blue points represent our training set and the red points represent our test set, then its clear we've done something wrong.

The solution is to break the process into two steps:

- **Fit:** Compute and store the parameter $\sigma = \sigma_X$ based only on the training set.
- **Transform:** Apply the function $X \mapsto X/\sigma$, using the same stored value of σ for both training and test sets.



Now both data sets are transformed in a consistent way.

In general, a transformer is an object in Scikit-learn with the following methods:

- **`.fit(X)`:** Learn and store parameters based on X (if any). Returns the transformer itself (to allow for method chaining).
- **`.transform(X)`:** Applies a transformation function to X (using stored parameters, if any) and returns the result.
- **`.fit_transform(X)`:** Shorthand for `.fit(X).transform(X)`

As you might expect, Scikit-learn implements transformation algorithms using classes. For example, `MaxAbsScaler` is a recipe for a specific type of scaling transformer, and `MaxAbsScaler()` is an object that implements that recipe.

In [3]:
```
from sklearn.preprocessing import MaxAbsScaler

trans = MaxAbsScaler()

X = np.arange(6).reshape(-1,1)
print(X)

trans.fit(X)
trans.transform(X)
```

```
[[0]
 [1]
 [2]
 [3]
 [4]
 [5]]
```

Out[3]:
```
array([[0. ],
       [0.2],
       [0.4],
       [0.6],
       [0.8],
       [1. ]])
```

In [4]:
```
X_new = np.arange(6).reshape(-1,1) + 5
X_new
```

Out[4]:
```
array([[ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
       [10]])
```

Question: What will each of the following cells output?

In [5]:
```
trans.transform(X_new)
```

Out[5]:
```
array([[1. ],
       [1.2],
       [1.4],
       [1.6],
       [1.8],
       [2. ]])
```

In [6]:
```
trans.fit_transform(X_new)
```

Out[6]:
```
array([[0.5],
       [0.6],
       [0.7],
       [0.8],
       [0.9],
       [1. ]])
```

In [7]:
```
trans.transform(X)
```

Out[7]:
```
array([[0. ],
       [0.1],
       [0.2],
       [0.3],
       [0.4],
       [0.5]])
```

In [8]:
```
trans.fit(X).transform(X_new)
```

Out[8]:
```
array([[1. ],
       [1.2],
       [1.4],
       [1.6],
       [1.8],
       [2. ]])
```

Feature Scaling

There are many situations in which it is beneficial to normalize data so that all features have comparable scales:

- To avoid biasing distance based algorithms (like K-Nearest Neighbors)
- When we plan to interpret coefficients in a linear model as a measure of feature importance
- When using regularization (a penalty based on coefficient size used to combat overfitting)

There are also many different ways to normalize data (also referred to as performing feature scaling):

- We can divide by the standard deviation or the range of the data in each dimension.
- We may want to center the data by shifting it so that the mean is zero in each dimension.
- We may want to transform data so that each feature is mapped to the same range (e.g. the interval $[0, 1]$).

One of the most popular options is the `StandardScaler` transformer which scales data to have a mean of 0 and a standard deviation of 1. More precisely, its `.transform` method applies the linear transformation

$$X \mapsto \frac{X - \mu}{\sigma}$$

where μ and σ are the mean and standard deviation learned from the training set.

In [9]:
```
from sklearn.preprocessing import StandardScaler

X = np.arange(6, dtype=np.float32).reshape(-1,1)
X_transformed = StandardScaler().fit_transform(X)

print("X:\n", X, "\n")
print("X_transformed:\n",X_transformed, "\n")
print("mean:",X_transformed.mean())
print("standard deviation:",X_transformed.std())
```

```
X:
 [[0.]
 [1.]
 [2.]
 [3.]
 [4.]
 [5.]]

X_transformed:
 [[-1.46385]
 [-0.87831]
 [-0.29277]
 [ 0.29277]
 [ 0.87831]
 [ 1.46385]]

mean: 0.0
standard deviation: 0.99999994
```

In [10]:
```
X_new = np.arange(6, dtype=np.float32).reshape(-1,1)*2 + 5
X_new_transformed = StandardScaler().fit(X).transform(X_new)

print("X_new:\n", X_new, "\n")
print("X_new_transformed:\n",X_new_transformed, "\n")
print("mean:",X_new_transformed.mean())
print("standard deviation:",X_new_transformed.std())
```

```
X_new:
 [[ 5.]
 [ 7.]
 [ 9.]
 [11.]
 [13.]
 [15.]]

X_new_transformed:
 [[1.46385  ]
 [2.6349301]
 [3.8060102]
 [4.9770904]
 [6.1481705]
 [7.31925  ]]

mean: 4.3915505
standard deviation: 1.9999998
```

It's fine to use `StandardScaler` for most applications, but you should consider other options (https://scikit-learn.org/stable/modules/preprocessing.html#standardization-or-mean-removal-and-variance-scaling) if you want to control the range of the output or if you're worried about outliers distorting the transformation.

The most common edge case to worry about is scaling sparse data. A data matrix is called sparse when most of its values are 0.

In [11]:
```
matr = np.eye(8,3,0) * 5
matr
```

Out[11]:
```
array([[5., 0., 0.],
       [0., 5., 0.],
       [0., 0., 5.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

It's most efficient to store sparse data in a compressed format where we only record the location and value of the non-zero entries.

In [12]:
```
from scipy import sparse

sparse_matr = sparse.csr_matrix(matr)
print(sparse_matr)
```

```
  (0, 0)        5.0
  (1, 1)        5.0
  (2, 2)        5.0
```

`StandardScaler` is a poor choice for sparse data:

- Subtracting the mean from each column *breaks* the sparsity of the matrix, forcing us to use a less efficient representation. If the data set is large, this can cause memory errors.
- The standard deviations of sparse features are abnormally small (since most values of each feature are 0). Dividing by these standard deviations can distort the data instead of normalizing it.

In [13]:
```
StandardScaler().fit_transform(sparse_matr.todense()) # What happens if we rem
ove .todense()? Why?
```

Out[13]:
```
array([[ 2.64575131, -0.37796447, -0.37796447],
       [-0.37796447,  2.64575131, -0.37796447],
       [-0.37796447, -0.37796447,  2.64575131],
       [-0.37796447, -0.37796447, -0.37796447],
       [-0.37796447, -0.37796447, -0.37796447],
       [-0.37796447, -0.37796447, -0.37796447],
       [-0.37796447, -0.37796447, -0.37796447],
       [-0.37796447, -0.37796447, -0.37796447]])
```

A better alternative is `MaxAbsScaler`, which divides each feature by its maximum absolute value and does not perform any shift.

In [14]:
```
from sklearn.preprocessing import MaxAbsScaler

print(MaxAbsScaler().fit_transform(sparse_matr))
```

```
  (0, 0)        1.0
  (1, 1)        1.0
  (2, 2)        1.0
```

Questions:

1. `MinMaxScaler` (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#) let's us scale features to a range whose minimum and maximum we choose explicitly. Can you think of any disadvantages of scaling to a fixed range (compared to the flexible scaling done by `StandardScaler`)?
2. Can you think of any situations in which feature scaling might *hurt* the performance of a Machine Learning algorithm?

Encoding Categorical Variables

Strings and other categorical values need to be given a numerical representation before most Machine Learning algorithms can be applied. The simplest solution, called **label encoding**, is to assign each unique value a different number. For example, if we have a color feature, then we might encode its values as follows.

$$\begin{aligned} \text{blue} &\mapsto 0 \\ \text{green} &\mapsto 1 \\ \text{red} &\mapsto 2 \\ \text{yellow} &\mapsto 3 \end{aligned}$$

In [15]:
```
from sklearn.preprocessing import LabelEncoder

ex = ["red", "green", "red", "yellow", "blue", "red"]
print(ex)
print(LabelEncoder().fit_transform(ex))
```

```
['red', 'green', 'red', 'yellow', 'blue', 'red']
[2 1 2 3 0 2]
```

Label encoding is easy, but the values it produces aren't particularly meaningful. Is "yellow" three times as much as "green"? Is "red" more than "green"? Not really.

This can be a big problem for Machine Learning algorithms. For example, suppose we are trying to use Linear Regression to predict car prices using features like the model and mileage. With label encoding, our data matrix will look something like this:

$$X_{raw} = \begin{matrix} \text{model} & \text{mileage} \end{matrix} \begin{bmatrix} \text{Accord} & 50,000 \\ \text{Accord} & 40,000 \\ \text{Camry} & 110,000 \\ \text{Camry} & 65,000 \\ \text{Civic} & 10,000 \end{bmatrix} \qquad X = \begin{matrix} \text{model} & \text{mileage} \end{matrix} \begin{bmatrix} 0 & 50,000 \\ 0 & 40,000 \\ 1 & 110,000 \\ 1 & 65,000 \\ 2 & 10,000 \end{bmatrix}$$

Using a Linear Regression algorithm means finding an approximation of the form

$$\text{price} \approx c_0 + c_1 \cdot \text{model} + c_2 \cdot \text{mileage}$$

The algorithm will probably learn some negative coefficient for the `mileage` variable. It makes intuitive sense that the price of each car will decrease as it is driven more. But what can the algorithm do with the `model` variable? More than you might expect, but less than we'd like. On the one hand, increases in the value of `model` indirectly convey information about what model the car is, so we should expect some correlation between `model` and `price`. On the other hand, our setup forces us to conflate different scenarios (does an increase of 1 mean that the model changed from Accord to Camry or from Camry to Civic?), which essentially means that we're throwing away useful information.

A better approach is to disentangle different category values by creating a new feature (or indicator variable) for each. This approach, called **one-hot encoding**, yields the following for our car example:

$$X_{raw} = \begin{matrix} \text{model} & \text{mileage} \end{matrix} \begin{bmatrix} \text{Accord} & 50,000 \\ \text{Accord} & 40,000 \\ \text{Camry} & 110,000 \\ \text{Camry} & 65,000 \\ \text{Civic} & 10,000 \end{bmatrix} \qquad X = \begin{matrix} \text{is\_accord} & \text{is\_camry} & \text{is\_civic} & \text{mileage} \end{matrix} \begin{bmatrix} 1 & 0 & 0 & 50,000 \\ 1 & 0 & 0 & 40,000 \\ 0 & 1 & 0 & 110,000 \\ 0 & 1 & 0 & 65,000 \\ 0 & 0 & 1 & 10,000 \end{bmatrix}$$

Now our Linear Regression is trying to find an approximation of the form

$$\text{price} \approx c_0 + c_1 \cdot \text{is\_accord} + c_2 \cdot \text{is\_camry} + c_3 \cdot \text{is\_civic} + c_4 \cdot \text{mileage}$$

and we're getting more use out of the `model` variable.

How should we implement one-hot encoding in practice? The `pandas.get_dummies` function is an excellent option, especially if we only need to encode a single data set.

In [16]:
```
import pandas as pd

models  = ["Accord", "Accord", "Camry", "Camry", "Civic"]
mileages = [50000, 40000, 110000, 65000, 10000]

X = pd.DataFrame({"model": models, "mileage": mileages})
X
```

Out[16]:

	model	mileage
0	Accord	50000
1	Accord	40000
2	Camry	110000
3	Camry	65000
4	Civic	10000

In [17]:
```
pd.get_dummies(X)
```

Out[17]:

	mileage	model_Accord	model_Camry	model_Civic
0	50000	1	0	0
1	40000	1	0	0
2	110000	0	1	0
3	65000	0	1	0
4	10000	0	0	1

Question: Why wasn't the `mileage` column encoded?

The drawback of using `get_dummies` is that it is not a transformer. If we want to encode a second data set in a consistent way, then we need to keep track of what labels we saw the first time and be careful about what columns represent which labels. We can't simply apply `get_dummies` a second time.

In [18]:
```
X_new = pd.DataFrame({"model": ["Civic", "Civic", "Camry"], "mileage": [30000, 40000, 50000]})
X_new
```

Out[18]:

	model	mileage
0	Civic	30000
1	Civic	40000
2	Camry	50000

In [19]:
```
pd.get_dummies(X_new)
```

Out[19]:

	mileage	model_Camry	model_Civic
0	30000	0	1
1	40000	0	1
2	50000	1	0

In this example, we've lost the `model_Accord` column. This simple issue could block us completely if we lose access to the original data set, but it's easily handled if we've done the proper bookkeeping.

In [20]:
```
# Get the columns of the original output
X_trans = pd.get_dummies(X)
old_cols = set(X_trans)

# Get the columns of the new output
X_new_trans = pd.get_dummies(X_new)
new_cols = set(X_new_trans)

# Add the missing columns to the new output
for col in old_cols.difference(new_cols):
    X_new_trans[col] = 0

# Put new columns in order (they were added on the right by default)
X_new_trans.sort_index(axis=1, inplace=True)
X_new_trans
```

Out[20]:

	mileage	model_Accord	model_Camry	model_Civic
0	30000	0	0	1
1	40000	0	0	1
2	50000	0	1	0

In an ideal world, we would have a single transformer handle all of this for us. If you have the very latest version of Scikit-learn (`0.20.1` or greater), then `OneHotEncoder` (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html) does the trick. Unfortunately, older versions of this transformer assume that data is label encoded to start with and cannot handle string values. `LabelEncoder` isn't built for 2-D arrays, so trying to encode data using the built in tools has traditionally been a pain point.

In [21]:
```
from sklearn import __version__ as version
print(version)
```

```
0.19.1
```

If you're using an older version and want a self contained solution using the Scikit-learn API, you may want to write a custom transformer to do the job. (For example, see the Custom Transformers section below)

There are many other encoding methods, but one-hot encoding is the most common and is usually a fine choice. If you're interested in alternatives, you may want to look at the category encoder package (http://contrib.scikit-learn.org/categorical-encoding/).

Question: One alternate encoding method, called **binary encoding**, uses N binary features to represent a categorical feature with $\leq 2^N$ unique values. For example, a binary encoding of the colors from before would look something like this:

$$\begin{aligned}\text{blue} &\mapsto 0 \mapsto (0, 0)\\ \text{green} &\mapsto 1 \mapsto (0, 1)\\ \text{red} &\mapsto 2 \mapsto (1, 0)\\ \text{yellow} &\mapsto 3 \mapsto (1, 1)\end{aligned}$$

What do you think the advantages and disadvantages of this method are?

Imputation

When data is missing, it's often preferable to impute or artificially assign values to empty fields rather than disregarding incomplete observations entirely. This is especially important when we expect the model we are training to be applied in situations with incomplete information.

Scikit-learn offers the `Imputer` transformer, which replaces missing values (instances of `np.nan`) with the average value of the appropriate feature (choose your preferred definition of 'average' using the `strategy` argument). More sophisticated imputation is better preformed in NumPy or Pandas, but `Imputer` has the advantage of convenience. Being a transformer means that it is easy to reuse, behaves consistently, and can be incorporated into pipelines.

In [22]:
```
from sklearn.preprocessing import Imputer

X_train = np.arange(8).reshape(4,2)*1.
X_train[0][1] = np.nan
X_train[3][0] = np.nan

print(X_train)

trans = Imputer(strategy="mean")

print("\n", trans.fit_transform(X_train))
```

```
[[ 0. nan]
 [ 2.  3.]
 [ 4.  5.]
 [nan  7.]]

 [[0. 5.]
 [2. 3.]
 [4. 5.]
 [2. 7.]]
```

In [23]:
```
# The means learned by .fit
trans.statistics_
```

Out[23]:
```
array([2., 5.])
```

In [24]:
```
X_test = np.array([[8, np.nan], [np.nan, 12]])
print(X_test)

print("\n", trans.transform(X_test))
```

```
[[ 8. nan]
 [nan 12.]]

 [[ 8.  5.]
 [ 2. 12.]]
```

As noted above, transformers like `Imputer` are easily incorporated into Scikit-learn pipelines.

In [25]:
```
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline

# This pipeline makes LinearRegression 'robust' by
# adding imputation as an automatic preprocessing step
RobustRegressor = Pipeline([("Imputer",   Imputer(strategy="mean")),
                            ("Regressor", LinearRegression())])

# Toy data so we can demo .fit/.predict syntax
y_train = np.arange(4)

RobustRegressor.fit(X_train, y_train)
RobustRegressor.predict(X_test)
```

Out[25]:
```
array([4.5, 5. ])
```

Note that **`RobustRegressor.fit`** is equivalent to **`Imputer().fit_transform`** followed by **`LinearRegression().fit`**.

Similarly, **`RobustRegressor.predict`** is equivalent to **`Imputer().transform`** followed by **`LinearRegression().predict`**.

Question: While they correctly convey the behavior of the pipeline, the preceding two statements are technically inaccurate. Why should we avoid running code like `Imputer().fit(X_train)` or `Imputer().fit_transform(X_train)` followed by `Imputer().transform(X_test)`?

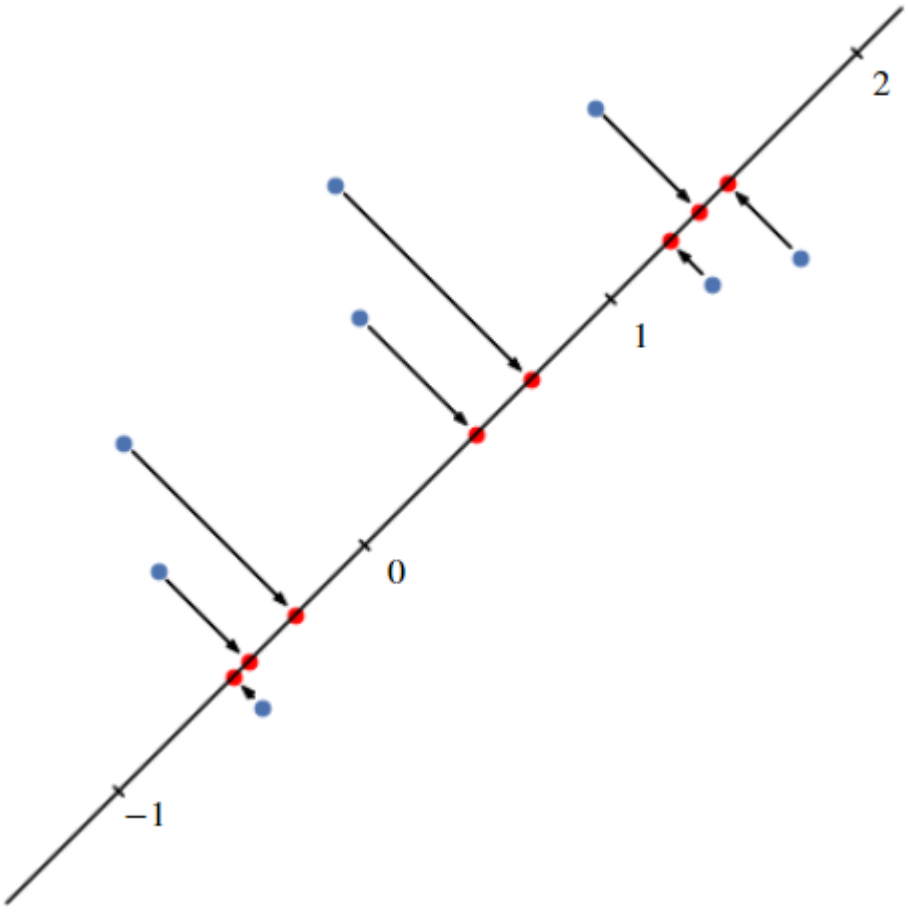
Dimension Reduction

The term **dimension reduction** refers to a variety of techniques for reducing the number of features in a data set. There are many reasons why this may be desirable:

- To save memory or reduce computation
- For visualization (since it's very hard to plot data with more than 2 or 3 dimensions)
- To combat overfitting

Caution is advised on this final point as premature dimension reduction is one of the easiest ways to *underfit*. After all, we are essentially throwing away information.

In general, the goal is to keep as much information as possible while using as few features as we can get away with. Rather than using a subset of the original features, we create new artificial features which are linear combinations of the originals. Mathematically, if we think of our data set as a collection of points in N dimensional space, then dimension reduction is simply projection onto a lower dimensional subspace.



How should we choose this subspace? There are several methods to choose from, but the most popular is **principal component analysis**, which maximizes the proportion of the data's variance that is conserved when the projection is applied. This is implemented in Scikit-learn as the PCA (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html) transformer.

In [26]:
```
points = np.array([2*np.arange(10), -np.arange(10)]).T + np.round(np.random.ra
ndom((10,2))/10,3)
points
```

Out[26]:
```
array([[ 6.0000e-03,  2.8000e-02],
       [ 2.0640e+00, -9.4700e-01],
       [ 4.0160e+00, -1.9320e+00],
       [ 6.0350e+00, -2.9230e+00],
       [ 8.0920e+00, -3.9980e+00],
       [ 1.0023e+01, -4.9810e+00],
       [ 1.2023e+01, -5.9070e+00],
       [ 1.4019e+01, -6.9890e+00],
       [ 1.6028e+01, -7.9950e+00],
       [ 1.8041e+01, -8.9460e+00]])
```

In [27]:
```
from sklearn.decomposition import PCA

PCA(n_components=1).fit_transform(points)
```

Out[27]:
```
array([[ 10.08213878],
       [  7.80541466],
       [  5.61898154],
       [  3.36995866],
       [  1.04933405],
       [ -1.1174279 ],
       [ -3.32034716],
       [ -5.58956968],
       [ -7.83637113],
       [-10.06211182]])
```

It's recommended to center data (subtract the mean in each dimension) before applying `PCA`, but you should be cautious about normalization (dividing by the standard deviation in each dimension) as this has the potential to inflate features that `PCA` would otherwise ignore.

For more information about dimension reduction, please refer to `ML_Unsupervised_Learning.ipynb`.

Natural Language Processing

One of the fundamental challenges of natural language processing is how to encode text data in a numerical format. The most common method, called the **bag of words** approach, is simply to count how many times each word occurs in each document. For example, if our 'documents' are short phrases, the bag of words encoding would look something like this:

Phrase		and	brown	fox	happy	moose	proud	quick	the
"the quick brown fox"		0	1	1	0	0	0	1	1
"the proud happy moose"	$\rightarrow$	0	0	0	1	1	1	0	1
"happy and proud"		1	0	0	1	0	1	0	0

We lose information about word order, but this easy and robust approach is sufficient for a wide range of applications.

In Scikit-learn, bag of words is implemented by the `CountVectorizer` transformer.

In [28]:
```
from sklearn.feature_extraction.text import CountVectorizer

bag_encoder = CountVectorizer()

bag_of_words_array = bag_encoder.fit_transform(["Let's feed the duck.", "Let's
play duck duck goose."])
bag_of_words_array
```

Out[28]:
```
<2x6 sparse matrix of type '<class 'numpy.int64'>'
        with 8 stored elements in Compressed Sparse Row format>
```

If we use a large vocabulary, then most of the counts will be 0, so the output is stored in sparse format by default. Converting to dense format is fine for small outputs, but not recommended when working with large amounts of data.

Note that `CountVectorizer` builds its vocabulary from the words it encounters in the training set. The key-value pairs stored in the `.vocabulary_` attribute are essentially column labels for the transformed output.

In [29]:
```
print(bag_of_words_array.todense())
print()
print(bag_encoder.vocabulary_)
```

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

{'let': 3, 'feed': 1, 'the': 5, 'duck': 0, 'play': 4, 'goose': 2}
```

Question: What are the separate functions of `CountVectorizer`'s `.fit` and `.transform` methods?

Scikit-learn also includes some basic variants of `CountVectorizer`:

- `TfidfVectorizer` (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) applies some NLP-specific feature scaling (TF-IDF normalization) on top of `CountVectorizer`'s normal behavior.
- `HashingVectorizer` (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html) removes the costs of needing to `.fit` a training set by using a hash function in place of an explicit vocabulary.

Custom Transformers

If no built-in transformer suits your needs, you can always write your own. Using the template below guarantees that your custom transformer class is compatible with the rest of Scikit-learn, which is really nice if you need to include custom transformers in a pipeline.

```
class Transformer(base.BaseEstimator, base.TransformerMixin):
  def __init__(self, ...):
    # initialization code

  def fit(self, X, y=None):
    # fit the transformation
    return self

  def transform(self, X):
    return ... # transformation
```

Note that the `.fit` method *must* return `self` in order for `.fit_transform` to work properly.

As a demonstration of syntax, here's a simple example which centers data by learning and subtracting the mean:

In [30]:
```
from sklearn import base

class CenteringTransformer(base.BaseEstimator, base.TransformerMixin):

    def __init__(self):
        self.mean = 0

    def fit(self, X):
        self.mean = X.mean()
        return self

    def transform(self, X):
        return X - self.mean
```

In [31]:
```
X_train = np.arange(5)
X_test  = np.arange(5) + 5
trans = CenteringTransformer()

trans.fit(X_train)

print(trans.mean)
print()
print("X_train:")
print(X_train, trans.transform(X_train))
print()
print("X_test:")
print(X_test, trans.transform(X_test))
```

```
2.0

X_train:
[0 1 2 3 4] [-2. -1.  0.  1.  2.]

X_test:
[5 6 7 8 9] [3. 4. 5. 6. 7.]
```

As a more practical example, here's the one-hot encoding procedure from before written as a transformer. This version assumes that the inputs are dataframes, but you can easily add branches to handle NumPy arrays.

In [32]:
```
class DummyEncoder(base.BaseEstimator, base.TransformerMixin):

    def __init__(self, columns=None):
        self.columns = columns # The columns to be encoded, if we want to spec
ify manually.
        self.output_cols = []  # The columns that are expected in the output.

    def fit(self, X):
        self.output_cols = set(pd.get_dummies(X))
        return self

    def transform(self, X):
        X_trans = pd.get_dummies(X)
        new_cols = set(X_trans)
        for col in self.output_cols.difference(new_cols):
            X_trans[col] = 0
        for col in new_cols.difference(self.output_cols): # New feature values
are ignored to ensure consistent representation
            X_trans.drop([col], axis=1, inplace=True)
        return X_trans.sort_index(axis=1)
```

In [33]:
```
trans = DummyEncoder()

models  = ["Accord", "Accord", "Camry", "Camry", "Civic"]
mileages = [50000, 40000, 110000, 65000, 10000]

X = pd.DataFrame({"model": models, "mileage": mileages})
trans.fit_transform(X)
```

Out[33]:

	mileage	model_Accord	model_Camry	model_Civic
0	50000	1	0	0
1	40000	1	0	0
2	110000	0	1	0
3	65000	0	1	0
4	10000	0	0	1

In [34]:
```
X_new = pd.DataFrame({"model": ["Ferrari", "Civic", "Camry"], "mileage": [3000
0, 40000, 50000]})
X_new

trans.transform(X_new)
```

Out[34]:

	mileage	model_Accord	model_Camry	model_Civic
0	30000	0	0	0
1	40000	0	0	1
2	50000	0	1	0

Note: We're actually calling `pd.get_dummies` *twice* every time we `.fit_transform`. The `.fit` method could be more efficient since we only need to determine the unique labels, but the inefficiency shouldn't bother us unless we're working with a large amount of data. This is an example of prioritizing developer time over computation time.

Answers to Questions

Feature Scaling

1. This method is much more sensitive to outliers. The maximum and minimum values in the data set (for each feature) completely determine the transformation used by `MinMaxScaler` while the standard deviations used by `StandardScaler` are determined in a more balanced way.
2. Feature scaling sometimes destroys information when features are already comparable and measured in the same units (for example, the lengths and widths in the Iris data set (https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html)). Feature scaling may also be inappropriate when used prior to dimension reduction (if the relative variances of the features are meaningful) although it is still correct to center data in these settings by subtracting the mean.

Encoding Categorical Variables

- The `mileage` column is automatically skipped because it has a numerical data type. Numbers sometimes represent categorical labels, so we can override this behavior using the `columns` argument of `pd.get_dummies` if we choose. Conversely, we should take care that columns with continuous values don't accidentally get the `object` data type (e.g. if numbers represented as strings).
- Binary encoding is a compromise between label encoding and one-hot encoding. The main downside of one-hot encoding is that it can create an unwieldy number of features, especially when the original categorical features take many different values. Binary encoding essentially prevents this explosion of features, but the features it generates are more abstract and less interpretable (similar to label encoding).

Imputation

- Every instance of `Imputer()` is creating a new instance of the `Imputer` class, so the means learned by `.fit` won't be stored. If we want information to persist across steps, then we need to create a persistent object with something like `trans=Imputer()`. This happens automatically inside of the pipeline object. To be pedantic:

`RobustRegressor.fit` is

`RobustRegressor.named_steps['Imputer'].fit_transform` followed by

`RobustRegressor.named_steps['Regressor'].fit`.

And `RobustRegressor.predict` is

`RobustRegressor.named_steps['Imputer'].transform` followed by

`RobustRegressor.named_steps['Regressor'].predict`.

Natural Language Processing

- `.fit` learns a vocabulary consisting of all of the unique words present in the training documents. It also chooses an order for these words (alphabetical by default) and stores that order as a dictionary

(essentially a pairing of words to column numbers). `.transform` uses the stored vocabulary and order to convert the given list of documents to an array of word counts. Any new words in the documents that aren't in the stored vocabulary are simply ignored.