Babeș-Bolyai University

Faculty of Mathematics and Computer Science

Curs opțional

Modele de inteligență artificială în schimbarea climatică

Preprocesarea seturilor de date

Standardization

Standardization is a transformation that centers the data by removing the mean value of each feature and then scale it by dividing (non-constant) features by their standard deviation.

After standardizing data the mean will be zero and the standard deviation one.

In practice, we often ignore the shape of the distribution and just transform the data to center it by removing the mean value of each feature, then scale it by dividing non-constant features by their standard deviation.

Methods:

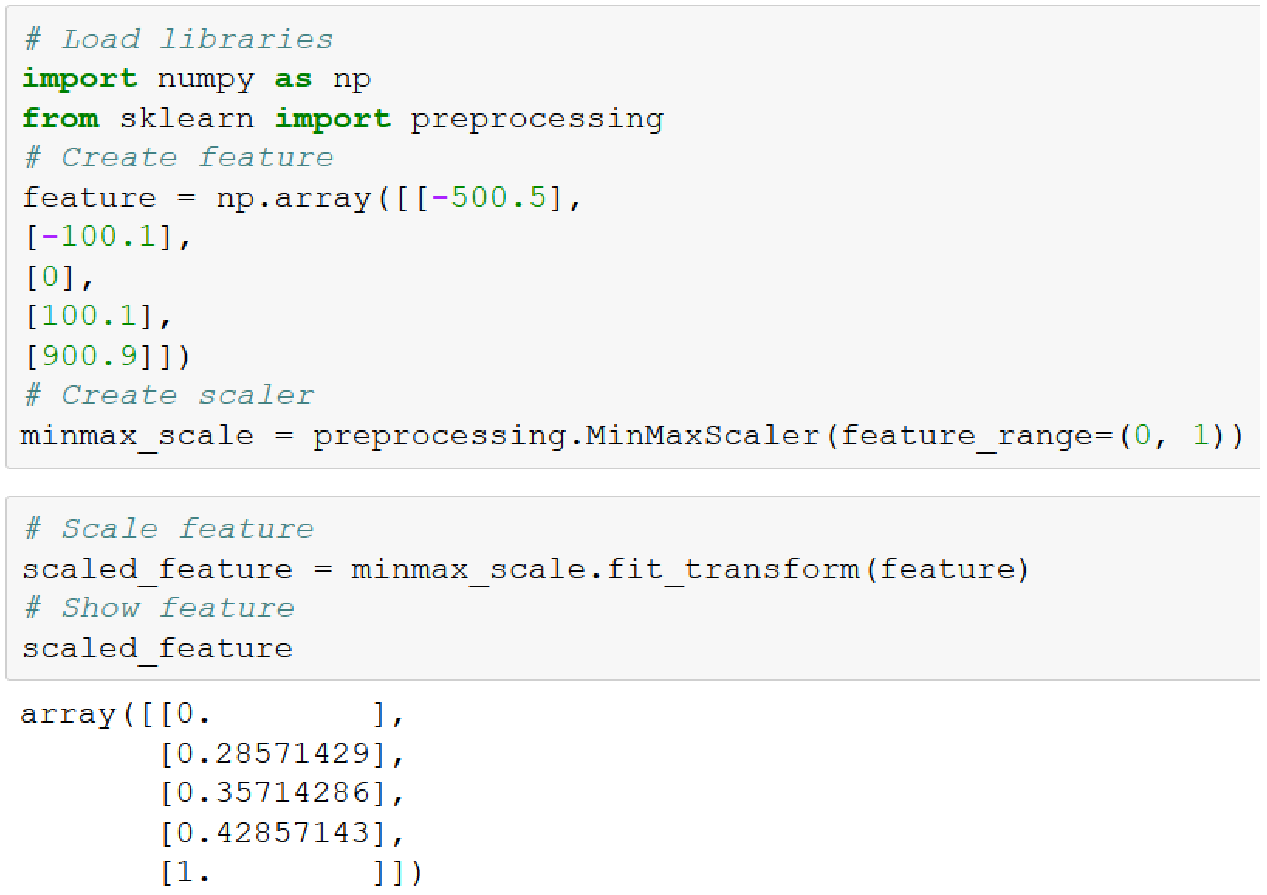
- Standard Scaler
- MinMaxScaler
- RobustScaler

Min-max scaling

- Rescaling is a common preprocessing task in machine learning.
- Many of the ML assume all features are on the same scale, typically 0 to 1 or –1 to 1.
- One of the simplest rescaling techniques is called min-max scaling.
- Min-max scaling uses the minimum and maximum values of a feature to rescale values to within a range.
- Specifically, min-max calculates:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

- where x is the feature vector, xi is an individual element of feature x, and x’i is the rescaled element.



```
# Load libraries
import numpy as np
from sklearn import preprocessing
# Create feature
feature = np.array([[-500.5],
[-100.1],
[0],
[100.1],
[900.9]])
# Create scaler
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))
```

```
# Scale feature
scaled_feature = minmax_scale.fit_transform(feature)
# Show feature
scaled_feature
```

```
array([[0.        ],
       [0.28571429],
       [0.35714286],
       [0.42857143],
       [1.        ]])
```

Standard Scaler

Allows to transform the data such that it has a mean, x̄, of 0 and a standard deviation, σ, of 1.

Specifically, each element in the feature is transformed so that:

$$x'_i = \frac{x_i - \bar{x}}{\sigma}$$

where x'i is our standardized form of xi.

The transformed feature represents the number of standard deviations the original value is away from the feature's mean value.

For example, principal component analysis often works better using standardization, while min-max scaling is often recommended for neural networks.

As a general rule, I'd recommend defaulting to standardization unless you have a specific reason to use an alternative.

Standardization is a common go-to scaling method for machine learning preprocessing and is used more than min-max scaling.

However, it depends on the learning algorithm.

```python
# Load libraries
import numpy as np
from sklearn import preprocessing
# Create feature
x = np.array([[-1000.1],
[-200.2],
[500.5],
[600.6],
[9000.9]])
# Create scaler
scaler = preprocessing.StandardScaler()
# Transform the feature
standardized = scaler.fit_transform(x)
# Show feature
standardized
```

```
array([[-0.76058269],
       [-0.54177196],
       [-0.35009716],
       [-0.32271504],
       [ 1.97516685]])
```

```python
# Print mean and standard deviation
print("Mean:", round(standardized.mean()))
print("Standard deviation:", standardized.std())
```

```
Mean: 0.0
Standard deviation: 1.0
```

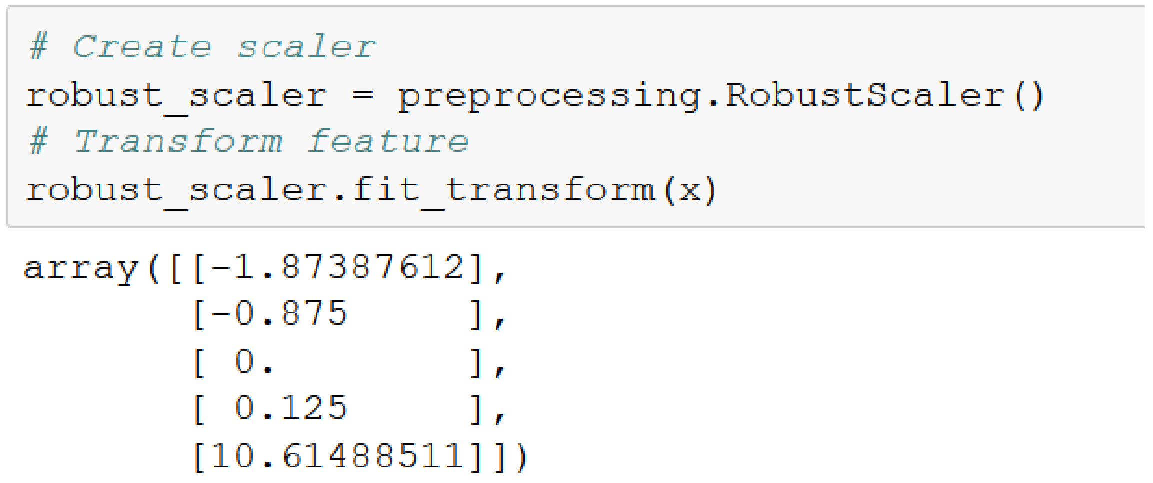
Robust Scaler

This Scaler removes the median and scales the data according to the quantile range.

The IQR is the range between the 1st quartile (25th quantile) and the 3rd quartile (75th quantile).

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set.

If our data has significant outliers, it can negatively impact our standardization by affecting the feature's mean and variance, in this case Robus Scaler is recommended.



```
# Create scaler
robust_scaler = preprocessing.RobustScaler()
# Transform feature
robust_scaler.fit_transform(x)
```

```
array([[-1.87387612],
       [-0.875     ],
       [ 0.        ],
       [ 0.125     ],
       [10.61488511]])
```

Normalization

- Normalization is the process of scaling individual samples to have a unit norm.
- We need to normalize data when the algorithm predicts based on the weighted relationships formed between data points.
- Scaling inputs to unit norms is a common operation for text classification or clustering.
- One of the key differences between scaling (e.g. standardizing) and normalizing, is that normalizing is performed row-wise whereas scaling is a column-wise operation.

We want to rescale the feature values of instances to have unit norm (a total length of 1).

Many rescaling methods (e.g., min-max scaling and standardization) operate on features; however, we can also rescale across individual instances.

Normalizer rescales the values on individual instances to have unit norm (the sum of their lengths is 1).

This type of rescaling is often used when we have many equivalent features (e.g., text classification when every word or n-word group is a feature).

Normalizer provides three norm options with Euclidean norm (often called L2) being the default argument:

$$\| x \|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

where x is an individual observation and xi is that observation's value for the i-th feature.

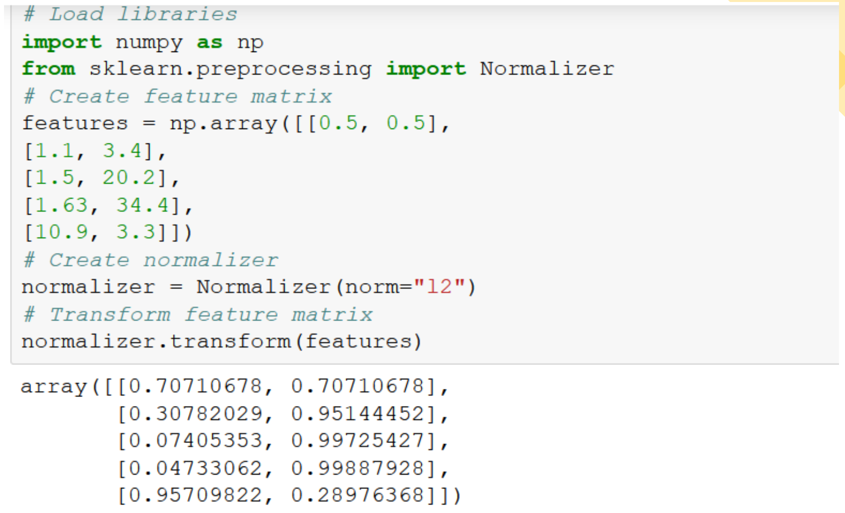
Alternatively, we can specify Manhattan norm (L1):

$$\| x \|_1 = \sum_{i=1}^{n} \left| x_i \right| .$$

Normalization – L2

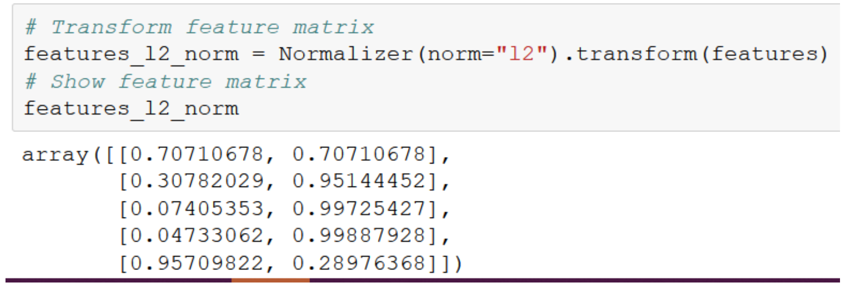
Intuitively, L2 norm can be thought of as the distance between two points for a bird (i.e., a straight line), while L1 can be thought of as the distance for a human walking on the street (walk north one block, east one block, north one block, east one block, etc.), which is why it is called “Manhattan norm” or “Taxicab norm.”

Practically, notice that norm='l1' rescales an observation’s values so they sum to 1, which can sometimes be an advantage.



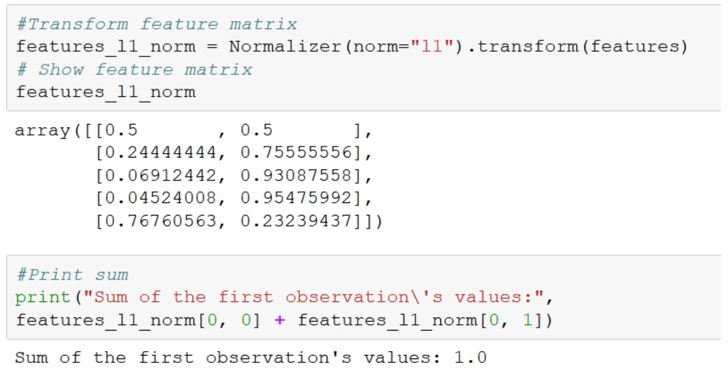
```
# Load libraries
import numpy as np
from sklearn.preprocessing import Normalizer
# Create feature matrix
features = np.array([[0.5, 0.5],
[1.1, 3.4],
[1.5, 20.2],
[1.63, 34.4],
[10.9, 3.3]])
# Create normalizer
normalizer = Normalizer(norm="l2")
# Transform feature matrix
normalizer.transform(features)
```

```
array([[0.70710678, 0.70710678],
       [0.30782029, 0.95144452],
       [0.07405353, 0.99725427],
       [0.04733062, 0.99887928],
       [0.95709822, 0.28976368]])
```



```
# Transform feature matrix
features_l2_norm = Normalizer(norm="l2").transform(features)
# Show feature matrix
features_l2_norm
```

```
array([[0.70710678, 0.70710678],
       [0.30782029, 0.95144452],
       [0.07405353, 0.99725427],
       [0.04733062, 0.99887928],
       [0.95709822, 0.28976368]])
```



```
#Transform feature matrix
features_l1_norm = Normalizer(norm="l1").transform(features)
# Show feature matrix
features_l1_norm
```

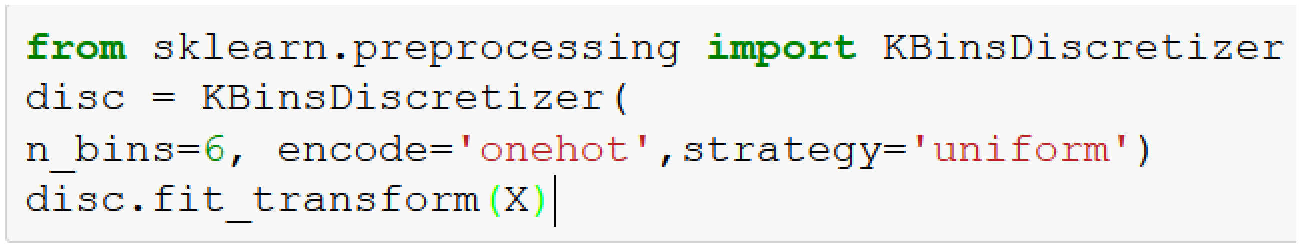
```
array([[0.5       , 0.5       ],
       [0.24444444, 0.75555556],
       [0.06912442, 0.93087558],
       [0.04524008, 0.95475992],
       [0.76760563, 0.23239437]])
```

```
#Print sum
print("Sum of the first observation\'s values:",
features_l1_norm[0, 0] + features_l1_norm[0, 1])
```

```
Sum of the first observation's values: 1.0
```

Discretization

- Data discretization refers to a method of converting a huge number of data values into smaller ones so that the evaluation and management of data become easy.
- In other words, data discretization is a method of converting attributes values of continuous data into a finite set of intervals with minimum data loss.
- There are two forms of data discretization first is supervised discretization, and the second is unsupervised discretization.
- Supervised discretization refers to a method in which the class data is used.
- Unsupervised discretization refers to a method depending upon the way which operation proceeds
- Sklearn provides a KBinsDiscretizer class that can take care of this. The only thing you have to specify, is the number of bins (n_bins) for each feature and how to encode these bins (ordinal, onehot or onehot-dense).



```
from sklearn.preprocessing import KBinsDiscretizer
disc = KBinsDiscretizer(
n_bins=6, encode='onehot',strategy='uniform')
disc.fit_transform(X)
```

Encoding categorical features

- Managing categorical data is another essential process during data preprocessing.
- Even for tree-based models, it is important to convert categorical features to a numerical representation.
- Label Encoding refers to converting the labels into the numeric form so as to convert them to machine-readable form.
- Machine learning algorithms can then decide in a better way how those labels must be operated.
- It is an important pre-processing step for the structured dataset in supervised learning.

Encoding categorical features

Theoretically, the proper strategy is to assign each class a numerical value (e.g., Texas = 1, California = 2).

Practically, when our classes have no intrinsic ordering (e.g., Texas isn't "less" than California), our numerical values create an ordering that is not present.

The proper strategy is to create a binary feature for each class in the original feature.

This is often called one-hot encoding.

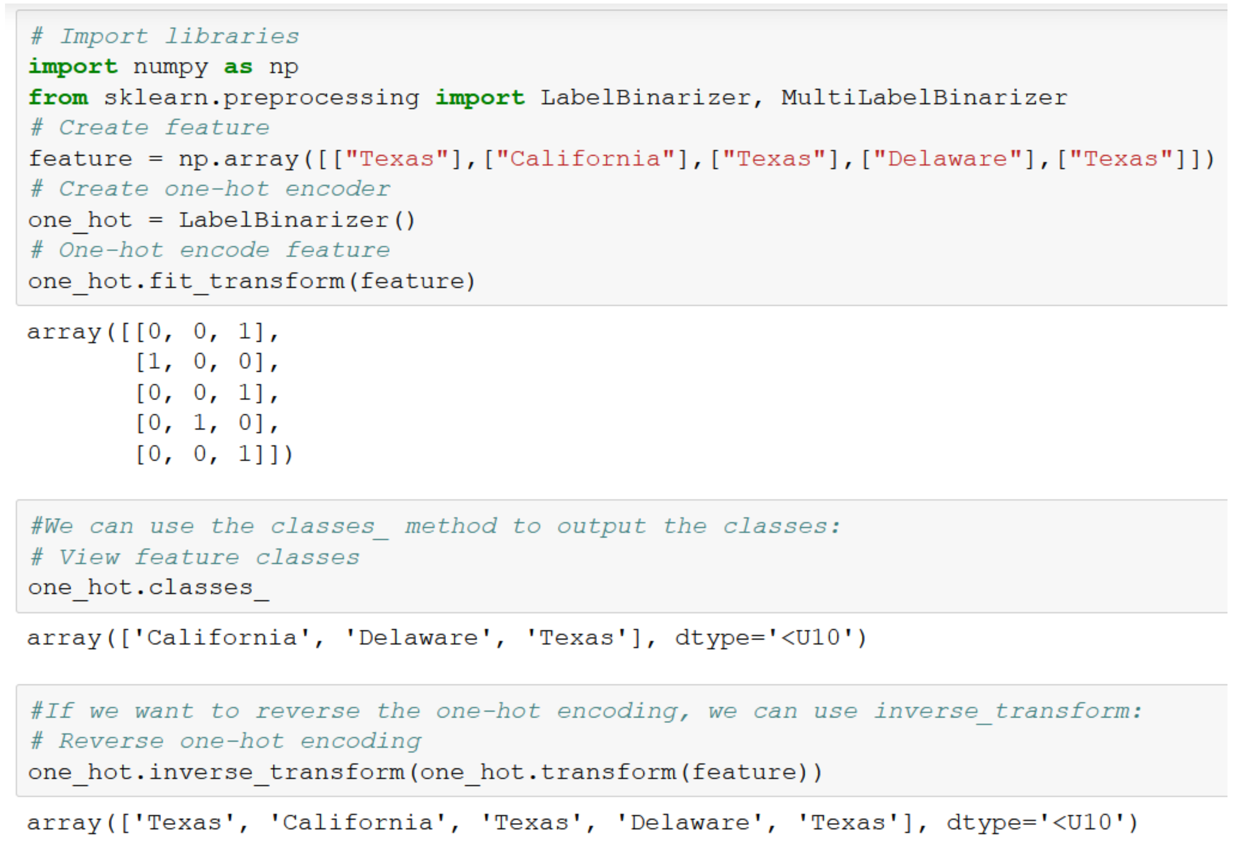
Our solution's feature was a vector containing three classes (i.e., Texas, California, and Delaware).

In one-hot encoding, each class becomes its own feature with 1s when the class appears and 0s otherwise.

Because our feature had three classes, one-hot encoding returned three binary features (one for each class).

By using one-hot encoding we can capture the membership of an instance in a class while preserving the notion that the class lacks any sort of hierarchy.

Finally, it is worthwhile to note that it is often recommended that after one-hot encoding a feature, we drop one of the one-hot encoded features in the resulting matrix to avoid linear dependence.



```
# Import libraries
import numpy as np
from sklearn.preprocessing import LabelBinarizer, MultiLabelBinarizer
# Create feature
feature = np.array([["Texas"],["California"],["Texas"],["Delaware"],["Texas"]])
# Create one-hot encoder
one_hot = LabelBinarizer()
# One-hot encode feature
one_hot.fit_transform(feature)
```

```
array([[0, 0, 1],
       [1, 0, 0],
       [0, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])
```

```
#We can use the classes_ method to output the classes:
# View feature classes
one_hot.classes_
```

```
array(['California', 'Delaware', 'Texas'], dtype='<U10')
```

```
#If we want to reverse the one-hot encoding, we can use inverse_transform:
# Reverse one-hot encoding
one_hot.inverse_transform(one_hot.transform(feature))
```

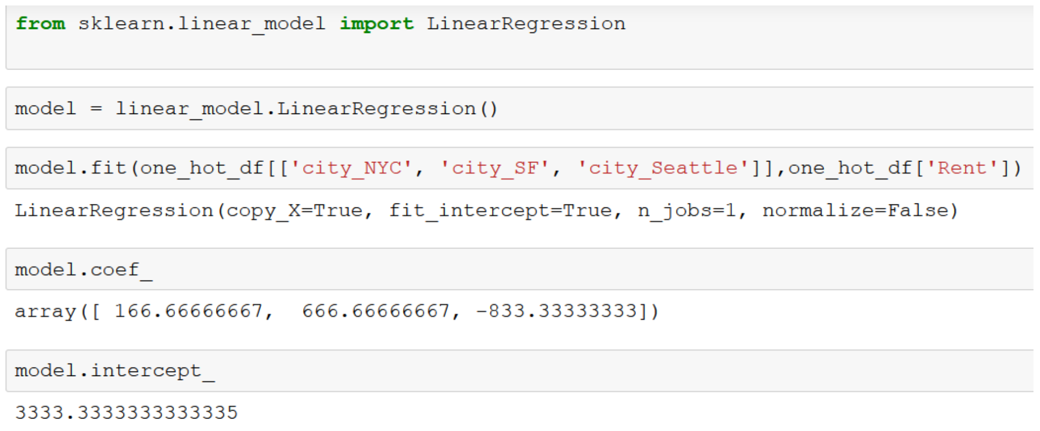
```
array(['Texas', 'California', 'Texas', 'Delaware', 'Texas'], dtype='<U10')
```

One hot and dummy

With one-hot encoding, the intercept term represents the global mean of the target variable, and each of the linear coefficients represents how much that city’s average rent differs from the global mean.

With dummy coding, the bias coefficient represents the mean value of the response variable y for the reference category (which in the example is the city NYC).

The coefficient for the ith feature is equal to the difference between the mean response value for the ith category and the mean of the reference category.



```
from sklearn.linear_model import LinearRegression
```

```
model = linear_model.LinearRegression()
```

```
model.fit(one_hot_df[['city_NYC', 'city_SF', 'city_Seattle']],one_hot_df['Rent'])
```

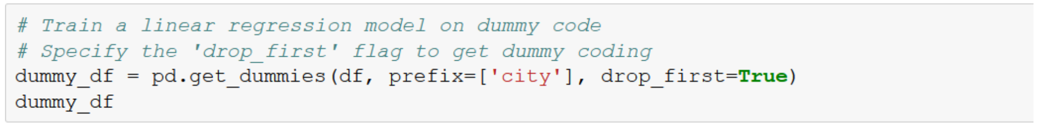
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

```
model.coef_
```

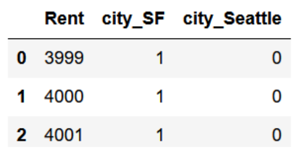
```
array([ 166.66666667,  666.66666667, -833.33333333])
```

```
model.intercept_
```

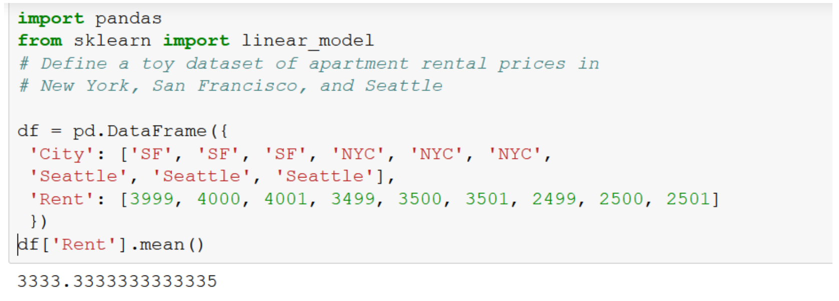
```
3333.3333333333335
```



```
# Train a linear regression model on dummy code
# Specify the 'drop_first' flag to get dummy coding
dummy_df = pd.get_dummies(df, prefix=['city'], drop_first=True)
dummy_df
```



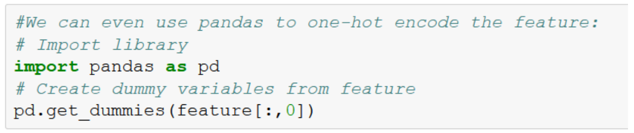
	Rent	city_SF	city_Seattle
0	3999	1	0
1	4000	1	0
2	4001	1	0



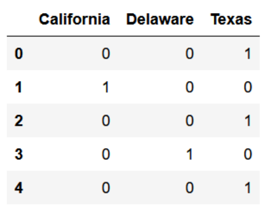
```
import pandas
from sklearn import linear_model
# Define a toy dataset of apartment rental prices in
# New York, San Francisco, and Seattle

df = pd.DataFrame({
 'City': ['SF', 'SF', 'SF', 'NYC', 'NYC', 'NYC',
 'Seattle', 'Seattle', 'Seattle'],
 'Rent': [3999, 4000, 4001, 3499, 3500, 3501, 2499, 2500, 2501]
 })
df['Rent'].mean()
```

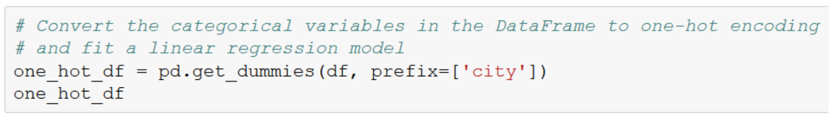
```
3333.3333333333335
```



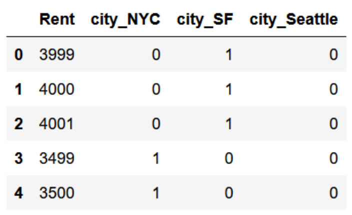
```
#We can even use pandas to one-hot encode the feature:
# Import library
import pandas as pd
# Create dummy variables from feature
pd.get_dummies(feature[:,0])
```



	California	Delaware	Texas
0	0	0	1
1	1	0	0
2	0	0	1
3	0	1	0
4	0	0	1



```
# Convert the categorical variables in the DataFrame to one-hot encoding
# and fit a linear regression model
one_hot_df = pd.get_dummies(df, prefix=['city'])
one_hot_df
```

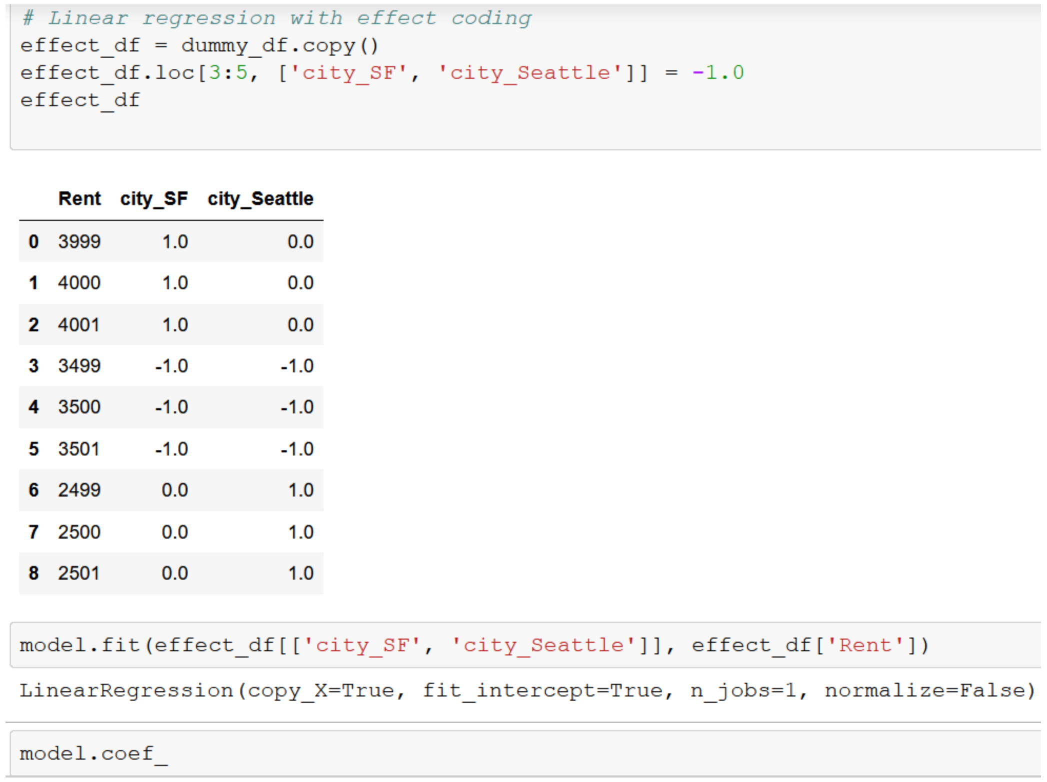


	Rent	city_NYC	city_SF	city_Seattle
0	3999	0	1	0
1	4000	0	1	0
2	4001	0	1	0
3	3499	1	0	0
4	3500	1	0	0

Effect Coding

Other variant of categorical variable encoding is effect coding.

Effect coding is very similar to dummy coding, with the difference that the reference category is now represented by the vector of all –1's.



```
# Linear regression with effect coding
effect_df = dummy_df.copy()
effect_df.loc[3:5, ['city_SF', 'city_Seattle']] = -1.0
effect_df
```

	Rent	city_SF	city_Seattle
0	3999	1.0	0.0
1	4000	1.0	0.0
2	4001	1.0	0.0
3	3499	-1.0	-1.0
4	3500	-1.0	-1.0
5	3501	-1.0	-1.0
6	2499	0.0	1.0
7	2500	0.0	1.0
8	2501	0.0	1.0

```
model.fit(effect_df[['city_SF', 'city_Seattle']], effect_df['Rent'])
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

```
model.coef_
```

Pros and Cons

One-hot, dummy, and effect coding are very similar to one another.

One-hot encoding is redundant, which allows for multiple valid models for the same problem.

The non-uniqueness is sometimes problematic for interpretation, but the advantage is that each feature clearly corresponds to a category.

Moreover, missing data can be encoded as the all-zeros vector, and the output should be the overall mean of the target variable.

Dummy coding and **effect coding** are not redundant. They give rise to unique and interpretable models.

The downside of dummy coding is that it cannot easily handle missing data, since the all-zeros vector is already mapped to the reference category.

It also encodes the effect of each category relative to the reference category, which may look strange.

Effect coding avoids this problem by using a different code for the reference category, but the vector of all –1's is a dense vector, which is expensive for both storage and computation.

(For this reason, popular ML software packages such as Pandas and scikit-learn have opted for dummy coding or one-hot encoding instead of effect coding.)

Encoding Ordinal Categorical Features

We have an ordinal categorical feature (e.g., high, medium, low). We have to transform string labels to numerical equivalents.

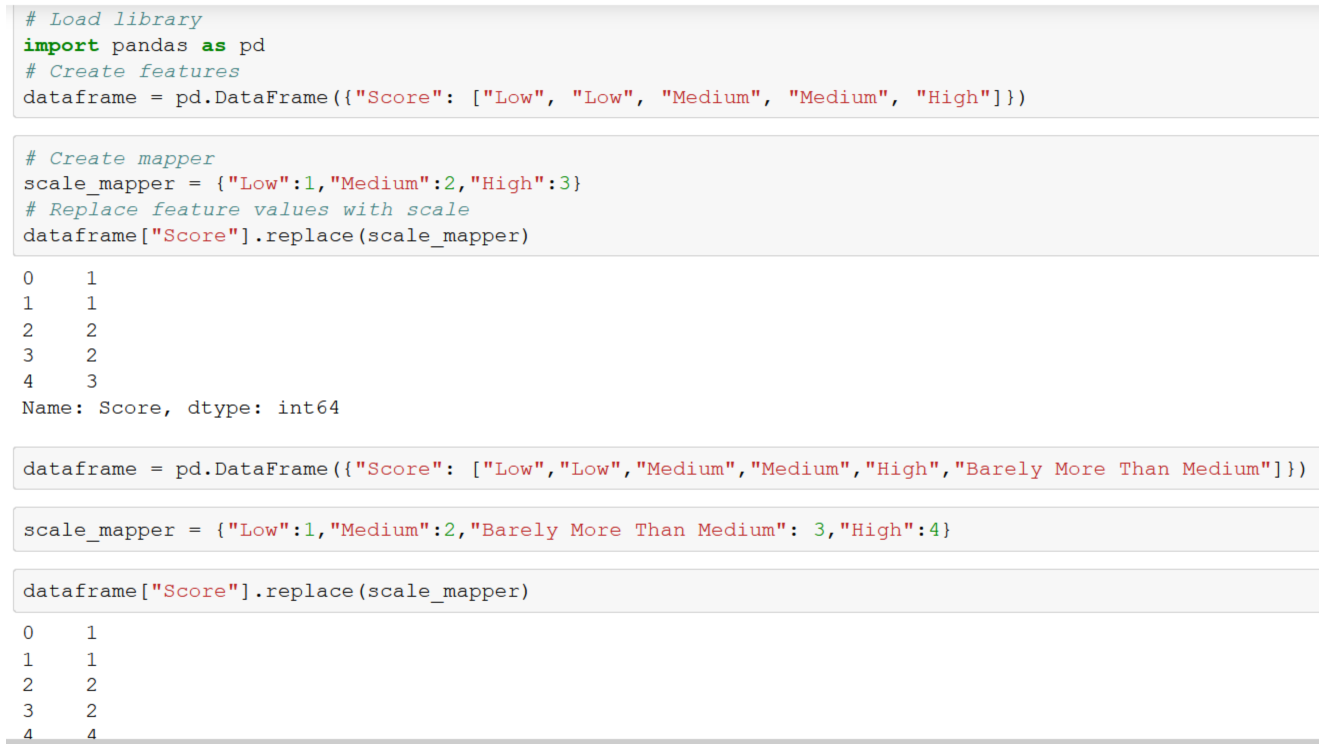
Often we have a feature with classes that have some kind of natural ordering.

A famous example is the Likert scale:

- Strongly Agree
- Agree
- Neutral
- Disagree
- Strongly Disagree

When encoding the feature for use in machine learning, we need to transform the ordinal classes into numerical values that maintain the notion of ordering.

The most common approach is to create a dictionary that maps the string label of the class to a number and then apply that map to the feature.



```
# Load library
import pandas as pd
# Create features
dataframe = pd.DataFrame({"Score": ["Low", "Low", "Medium", "Medium", "High"]})
```

```
# Create mapper
scale_mapper = {"Low":1,"Medium":2,"High":3}
# Replace feature values with scale
dataframe["Score"].replace(scale_mapper)
```

```
0    1
1    1
2    2
3    2
4    3
Name: Score, dtype: int64
```

```
dataframe = pd.DataFrame({"Score": ["Low","Low","Medium","Medium","High","Barely More Than Medium"]})
```

```
scale_mapper = {"Low":1,"Medium":2,"Barely More Than Medium": 3,"High":4}
```

```
dataframe["Score"].replace(scale_mapper)
```

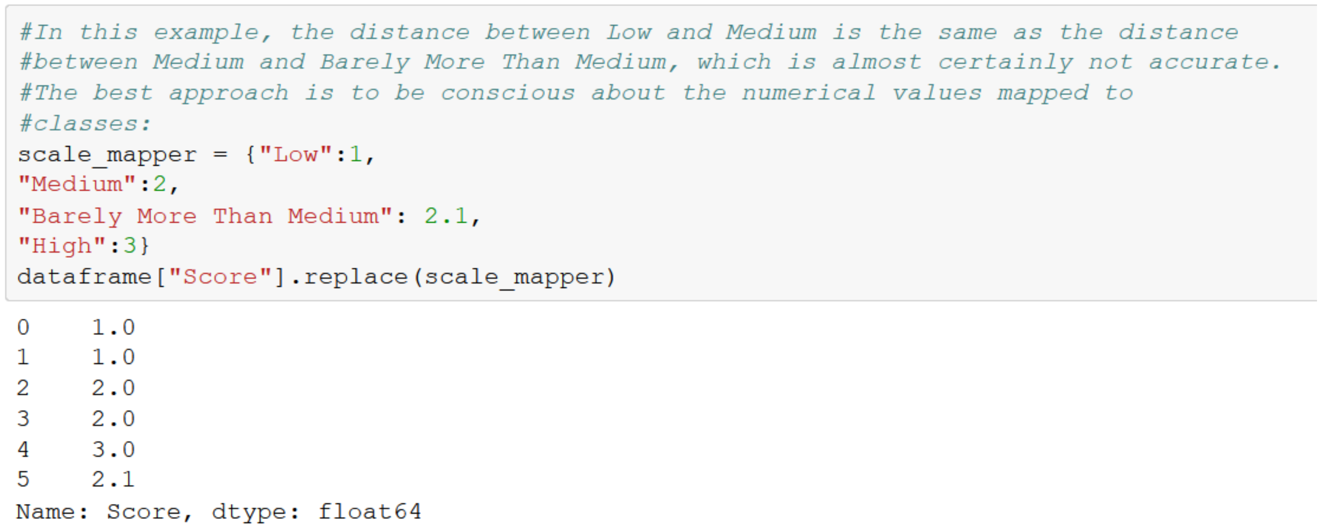
```
0    1
1    1
2    2
3    2
4    4
```

Encoding Ordinal Categorical Features - 2

It is important that our choice of numeric values is based on our prior information on the ordinal classes.

In our solution, high is literally three times larger than low.

This is fine in any instances, but can break down if the assumed intervals between the classes are not equal:



```python
#In this example, the distance between Low and Medium is the same as the distance
#between Medium and Barely More Than Medium, which is almost certainly not accurate.
#The best approach is to be conscious about the numerical values mapped to
#classes:
scale_mapper = {"Low":1,
"Medium":2,
"Barely More Than Medium": 2.1,
"High":3}
dataframe["Score"].replace(scale_mapper)
```

```
0    1.0
1    1.0
2    2.0
3    2.0
4    3.0
5    2.1
Name: Score, dtype: float64
```

Features engineering

Feature Extraction - how to reduce the dimensionality of our feature matrix by creating new features with (ideally) similar ability to train quality models but with significantly fewer dimensions.

Feature Selection - selecting high-quality, informative features and dropping less useful features.

- **Filter methods** select the best features by examining their statistical properties.
- **Wrapper methods** use trial and error to find the subset of features that produce models with the highest quality predictions.
- **Embedded methods** select the best feature subset as part or as an extension of a learning algorithm's training process.

Feature dimensionality reduction through feature extraction

- The goal of feature extraction for dimensionality reduction is to transform our set of features**, p_original**, such that we end up with a new set, **p_new**, where **p_original > p_new**, while still keeping much of the underlying information.

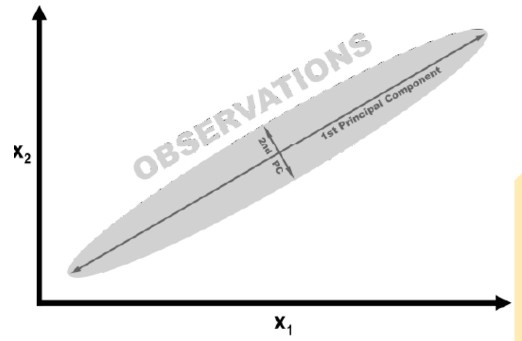
- In other way, we reduce the number of features with only a small loss in our data's ability to generate high-quality predictions.

- One downside of the feature extraction techniques we discuss is that the new features we generate will not be interpretable by humans.

- They will contain as much or nearly as much ability to train our models, but will appear to the human eye as a collection of random numbers.

- If we wanted to maintain our ability to interpret our models, dimensionality reduction through feature selection is a better option.

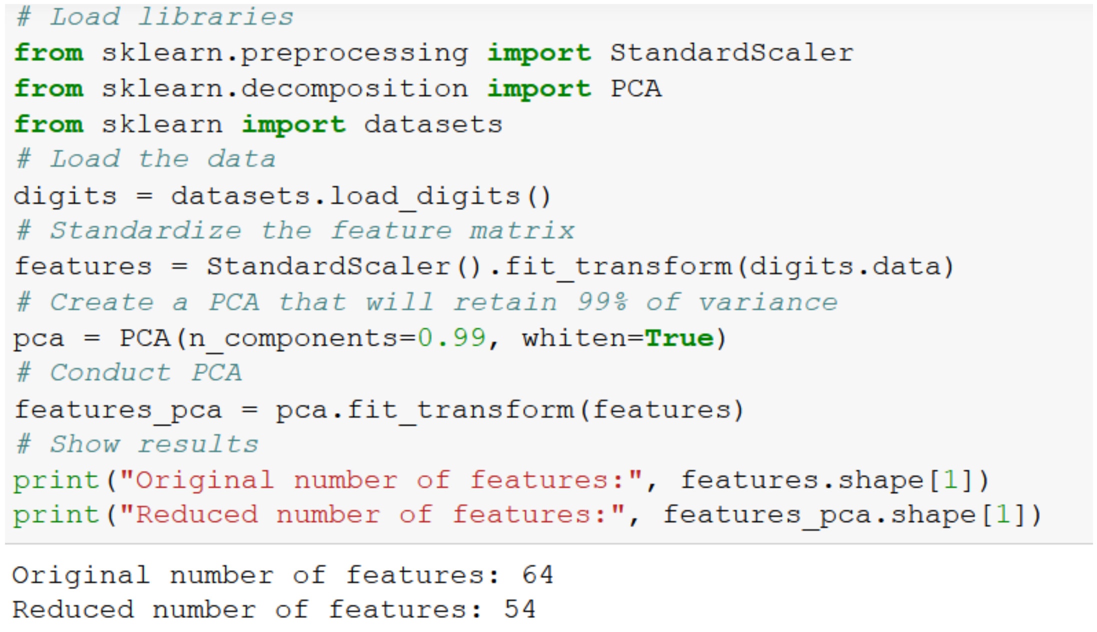
Principal component analysis (PCA)



- PCA is a popular linear dimensionality reduction technique. PCA projects instances onto the principal components of the feature matrix that retain the most variance.
- PCA is an unsupervised technique, meaning that it does not use the information from the target vector and instead only considers the feature matrix.
- In the figure, our data contains two features, x1 and x2.
- Looking at the visualization, it should be clear that instances (observations) are spread out like a cigar, with a lot of length and very little height. More specifically, we can say that the variance of the "length" is significantly greater than the "height."
- Instead of length and height, we refer to the "directions" with the most variance as the first principal component
- and the "direction" with the second-most variance as the second principal component (and so on).
- If we wanted to reduce our features, one strategy would be to project all instances in our 2D space onto the 1D principal component.
- We would lose the information captured in the second principal component, but in some situations that would be an acceptable trade-off.

PCA

- PCA is implemented in scikit-learn using the pca method. n_components has two operations, depending on the argument provided.
- If the argument is greater than 1, n_components will return that many features.
- This leads to the question of how to select the number of features that is optimal.
- Fortunately for us, if the argument to n_components is between 0 and 1, pca returns the minimum amount of features that retain that much variance.
- It is common to use values of 0.95 and 0.99, meaning 95% and 99% of the variance of the original features has been retained, respectively.
- The param whiten=True transforms the values of each principal component so that they have zero mean and unit variance.
- Another parameter and argument is svd_solver="randomized", which implements a stochastic algorithm to find the first principal components in often significantly less time.
- The output of our solution shows that PCA let us reduce our dimensionality by 10
- features while still retaining 99% of the information (variance) in the feature matrix.

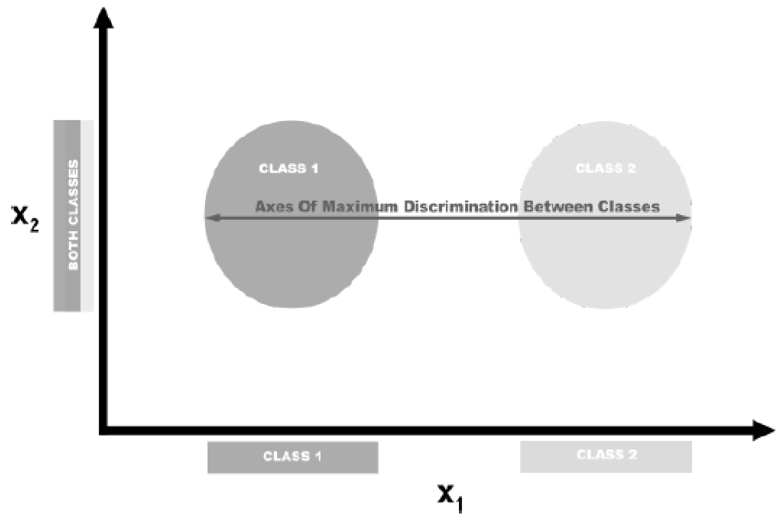


```python
# Load libraries
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets
# Load the data
digits = datasets.load_digits()
# Standardize the feature matrix
features = StandardScaler().fit_transform(digits.data)
# Create a PCA that will retain 99% of variance
pca = PCA(n_components=0.99, whiten=True)
# Conduct PCA
features_pca = pca.fit_transform(features)
# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_pca.shape[1])
```

```
Original number of features: 64
Reduced number of features: 54
```

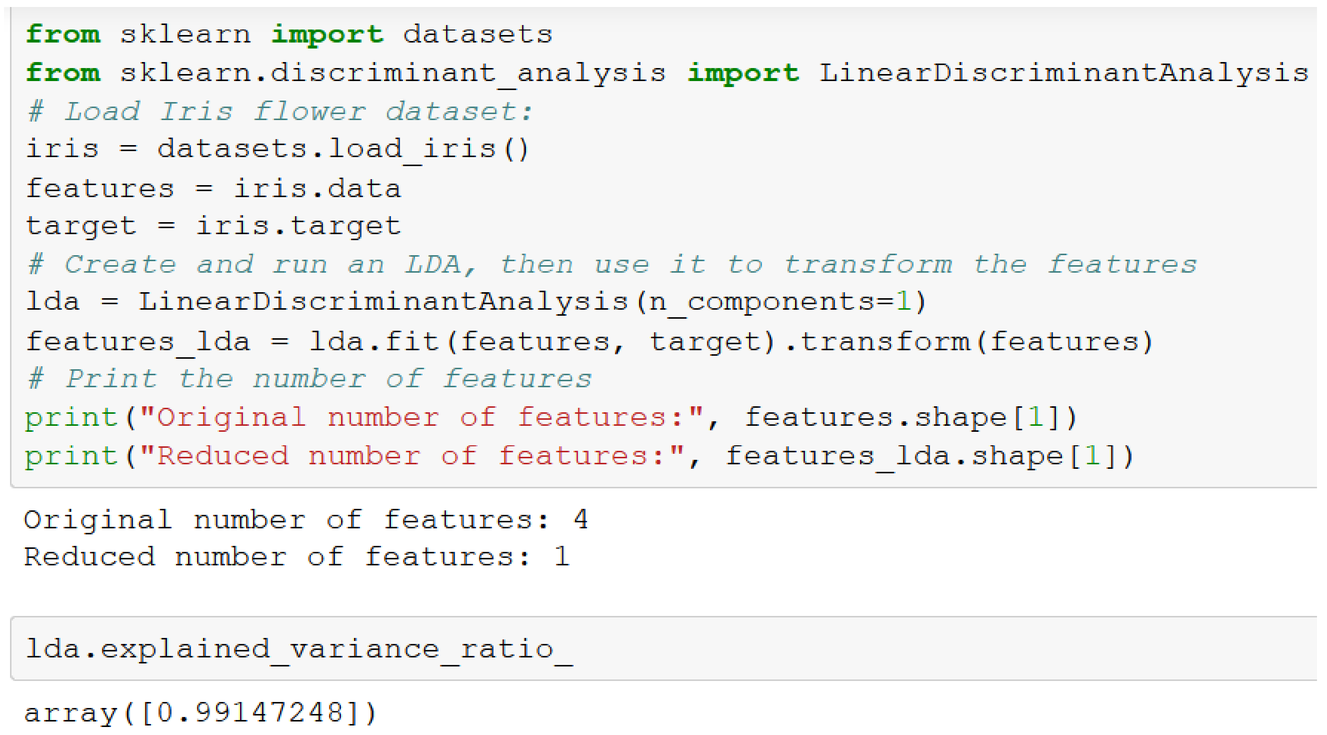
Reducing Features by Maximizing Class Separability using Linear Discriminant Analysis

- LDA is a classification method that is also a popular technique for dimensionality reduction.
- LDA works similarly to principal component analysis (PCA) in that it projects our feature space onto a lower-dimensional space.
- However, in PCA we were only interested in the component axes that maximize the variance in the data, while in LDA we have the additional goal of maximizing the differences between classes.
- In the picture, we have data comprising two target classes and two features.
- If we project the data onto the y-axis, the two classes are not easily separable (i.e., they overlap), while if we project the data onto the x-axis, we are left with a feature vector (i.e., we reduced our dimensionality by one) that still preserves class separability.
- In the real world, of course, the relationship between the classes will be more complex and the dimensionality will be higher, but the concept remains the same.



Reducing Features by Maximizing Class Separability

- Reducing the features to be used by a classifier using linear discriminant analysis (LDA) to project the features onto component axes that maximize the separation of classes.
- We can use explained_variance_ratio_ to view the amount of variance explained by each component.
- In our solution the single component explained over 99% of the variance:



```python
from sklearn import datasets
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
# Load Iris flower dataset:
iris = datasets.load_iris()
features = iris.data
target = iris.target
# Create and run an LDA, then use it to transform the features
lda = LinearDiscriminantAnalysis(n_components=1)
features_lda = lda.fit(features, target).transform(features)
# Print the number of features
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_lda.shape[1])
```

```
Original number of features: 4
Reduced number of features: 1
```

```python
lda.explained_variance_ratio_
```

```
array([0.99147248])
```

LDA example

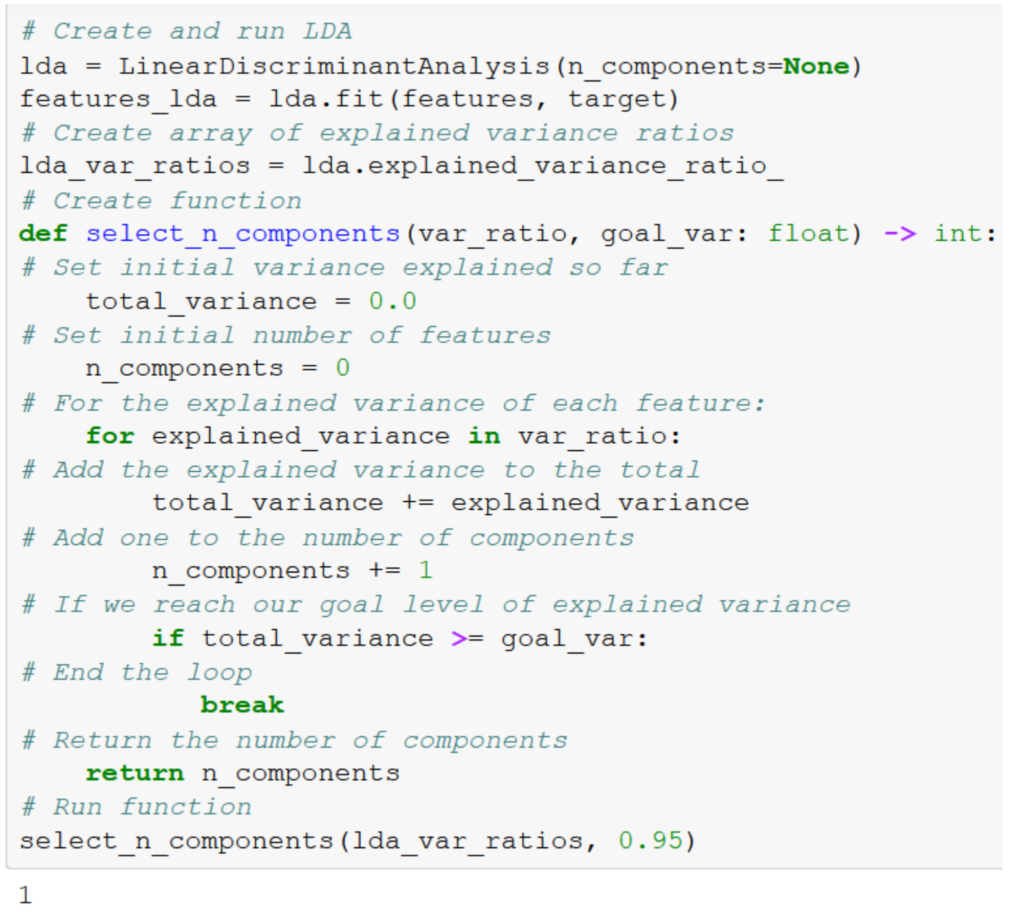
In scikit-learn, LDA is implemented using LinearDiscriminantAnalysis, which includes a parameter, n_components, indicating the number of features we want returned.

To figure out what argument value to use with n_components (e.g., how many parameters to keep), we can take advantage of the fact that explained_variance_ratio_ tells us the variance explained by each outputted feature and is a sorted array.

For example:

lda.explained_variance_ratio_

Specifically, we can run LinearDiscriminantAnalysis with n_components set to None to return the ratio of variance explained by every component feature, then calculate how many components are required to get above some threshold of variance explained (often 0.95 or 0.99):



```
# Create and run LDA
lda = LinearDiscriminantAnalysis(n_components=None)
features_lda = lda.fit(features, target)
# Create array of explained variance ratios
lda_var_ratios = lda.explained_variance_ratio_
# Create function
def select_n_components(var_ratio, goal_var: float) -> int:
# Set initial variance explained so far
    total_variance = 0.0
# Set initial number of features
    n_components = 0
# For the explained variance of each feature:
    for explained_variance in var_ratio:
# Add the explained variance to the total
        total_variance += explained_variance
# Add one to the number of components
        n_components += 1
# If we reach our goal level of explained variance
        if total_variance >= goal_var:
# End the loop
            break
# Return the number of components
    return n_components
# Run function
select_n_components(lda_var_ratios, 0.95)
```

```
1
```

Reducing Features Using Matrix Factorization

- We have a feature matrix of nonnegative values and want to reduce the dimensionality using non-negative matrix factorization (NMF) methd.

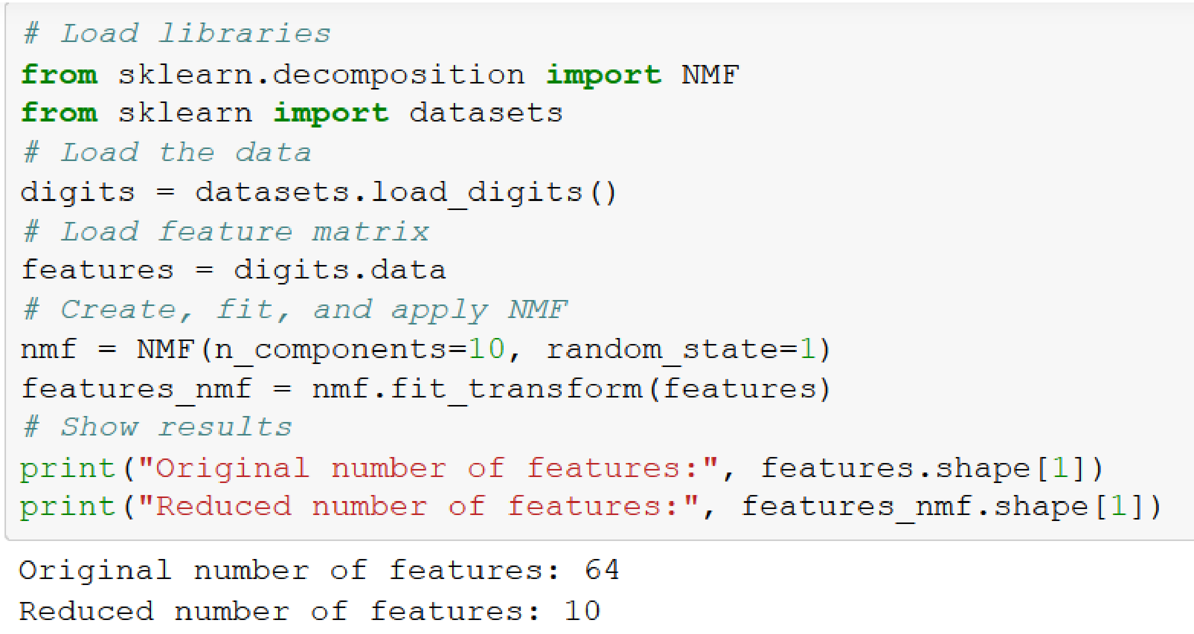
- NMF is an unsupervised technique for linear dimensionality reduction that factorizes (i.e., breaks up into multiple matrices whose product approximates the original matrix) the feature matrix into matrices representing the latent relationship between instances and their features.
- Intuitively, NMF can reduce dimensionality because in matrix multiplication, the two factors (matrices being multiplied) can have significantly fewer dimensions than the product matrix.
- Formally, given a desired number of returned features, r, NMF factorizes our feature matrix such that: $V \approx WH$ where V is our $d \times$ _n feature matrix (i.e., d features, n instances), W is a $d \times r$, and H is an $r \times n$ matrix.

  By adjusting the value of r we can set the amount of dimensionality reduction desired.

  One major requirement of the methid is that, as the name implies, the feature matrix cannot contain negative values.

  Additionally, unlike PCA and other techniques we have examined, NMA does not provide us with the explained variance of the outputted features.

  Thus, the best way for us to find the optimum value of n_components is by trying a range of values to find the one that produces the best result in our end model

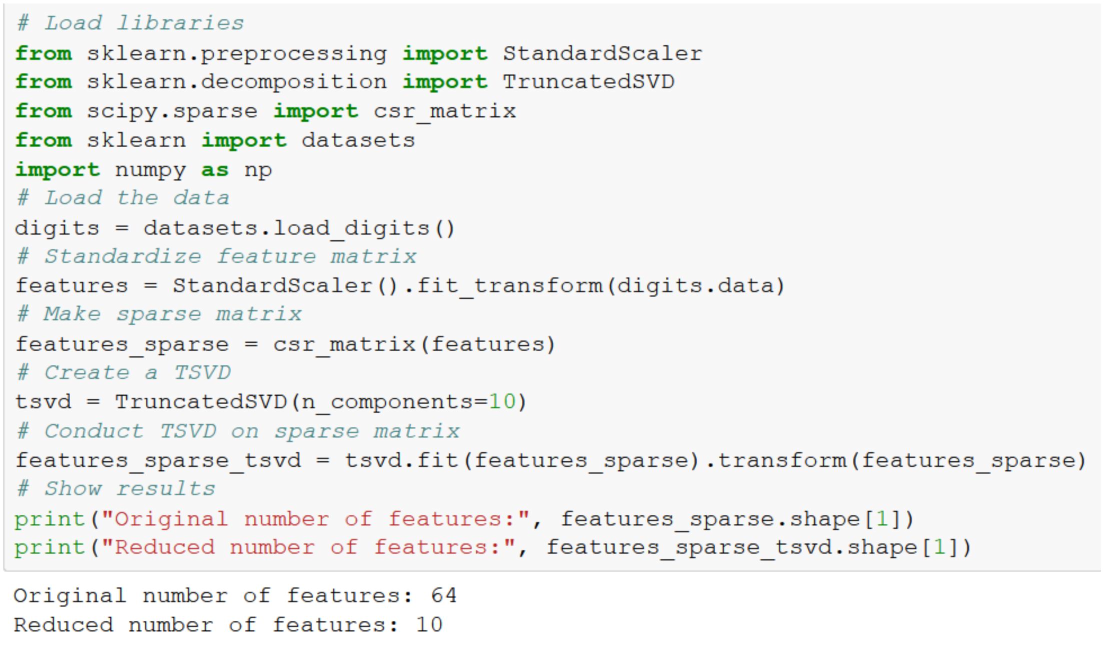


```python
# Load libraries
from sklearn.decomposition import NMF
from sklearn import datasets
# Load the data
digits = datasets.load_digits()
# Load feature matrix
features = digits.data
# Create, fit, and apply NMF
nmf = NMF(n_components=10, random_state=1)
features_nmf = nmf.fit_transform(features)
# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_nmf.shape[1])
```

```
Original number of features: 64
Reduced number of features: 10
```

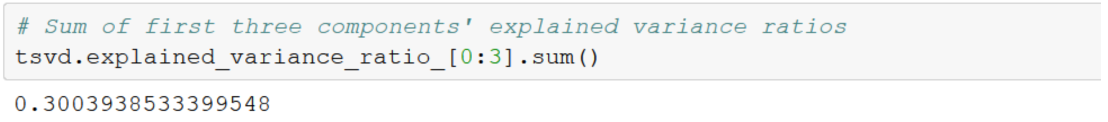
Reducing Features on Sparse Data

- Truncated Singular Value Decomposition (TSVD) is similar to PCA and in fact, PCA actually often uses non-truncated Singular Value Decomposition (SVD) in one of its steps.
- In regular SVD, given d features, SVD will create factor matrices that are d × d, whereas TSVD will return factors that are n × n, where n is previously specified by a parameter.
- The practical advantage of TSVD is that unlike PCA, it works on sparse feature matrices.
- One issue with TSVD is that because of how it uses a random number generator, the signs of the output can flip between fittings.
- An easy workaround is to use fit only once per preprocessing pipeline, then use transform multiple times.
- As with linear discriminant analysis, we have to specify the number of features (components) we want outputted.
- This is done with the n_components parameter.
- A natural question is then: what is the optimum number of components? One strategy is to include n_components as a hyperparameter to optimize during model selection (i.e.,choose the value for n_components that produces the best trained model).
- Alternatively,because TSVD provides us with the ratio of the original feature matrix's variance explained by each component, we can select the number of components thatexplain a desired amount of variance (95% or 99% are common values).
- For example, in our solution the first three outputted components explain approximately 30% of the original data's variance:
- We can automate the process by creating a function that runs TSVD with n_components set to one less than the number of original features and then calculate the numberof components that explain a desired amount of the original data's variance:



```python
# Load libraries
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import TruncatedSVD
from scipy.sparse import csr_matrix
from sklearn import datasets
import numpy as np
# Load the data
digits = datasets.load_digits()
# Standardize feature matrix
features = StandardScaler().fit_transform(digits.data)
# Make sparse matrix
features_sparse = csr_matrix(features)
# Create a TSVD
tsvd = TruncatedSVD(n_components=10)
# Conduct TSVD on sparse matrix
features_sparse_tsvd = tsvd.fit(features_sparse).transform(features_sparse)
# Show results
print("Original number of features:", features_sparse.shape[1])
print("Reduced number of features:", features_sparse_tsvd.shape[1])
```

```
Original number of features: 64
Reduced number of features: 10
```



```python
# Sum of first three components' explained variance ratios
tsvd.explained_variance_ratio_[0:3].sum()
```

```
0.3003938533399548
```

Thresholding Numerical Feature Variance

We have a set of numerical features and we want to remove those with low variance (i.e., likely containing little information).

Selecting a subset of features with variances above a given threshold.

Variance thresholding (VT) is one of the most basic approaches to feature selection.

It is motivated by the idea that features with low variance are likely less interesting (and useful) than features with high variance.

VT first computes the variance of each feature:

$$operatornameVar(x) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

where x is the feature vector, xi is an individual feature value, and μ is that feature's mean value.

Next, it drops all features whose variance does not meet that threshold.

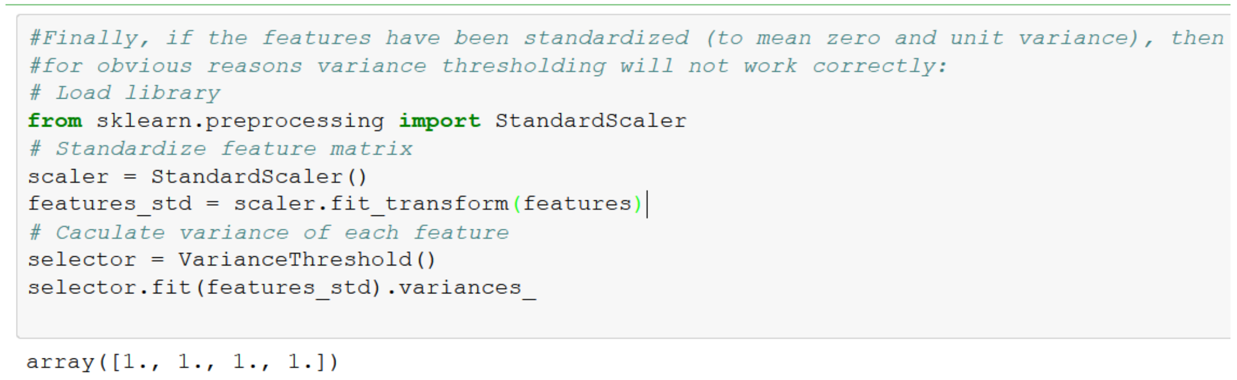
```
#Load libraries
from sklearn import datasets
from sklearn.feature_selection import VarianceThreshold
# import some data to play with
iris = datasets.load_iris()
# Create features and target
features = iris.data
target = iris.target
# Create thresholder
thresholder = VarianceThreshold(threshold=.5)
# Create high variance feature matrix
features_high_variance = thresholder.fit_transform(features)
# View high variance feature matrix
features_high_variance[0:3]
```

```
array([[5.1, 1.4, 0.2],
       [4.9, 1.4, 0.2],
       [4.7, 1.3, 0.2]])
```

```
# View variances
thresholder.fit(features).variances_
```

```
array([0.68112222, 0.18675067, 3.09242489, 0.57853156])
```

Variance Threshold (VT)



```
#Finally, if the features have been standardized (to mean zero and unit variance), then
#for obvious reasons variance thresholding will not work correctly:
# Load library
from sklearn.preprocessing import StandardScaler
# Standardize feature matrix
scaler = StandardScaler()
features_std = scaler.fit_transform(features)
# Caculate variance of each feature
selector = VarianceThreshold()
selector.fit(features_std).variances_
```

```
array([1., 1., 1., 1.])
```

There are two things to keep in mind when employing VT.

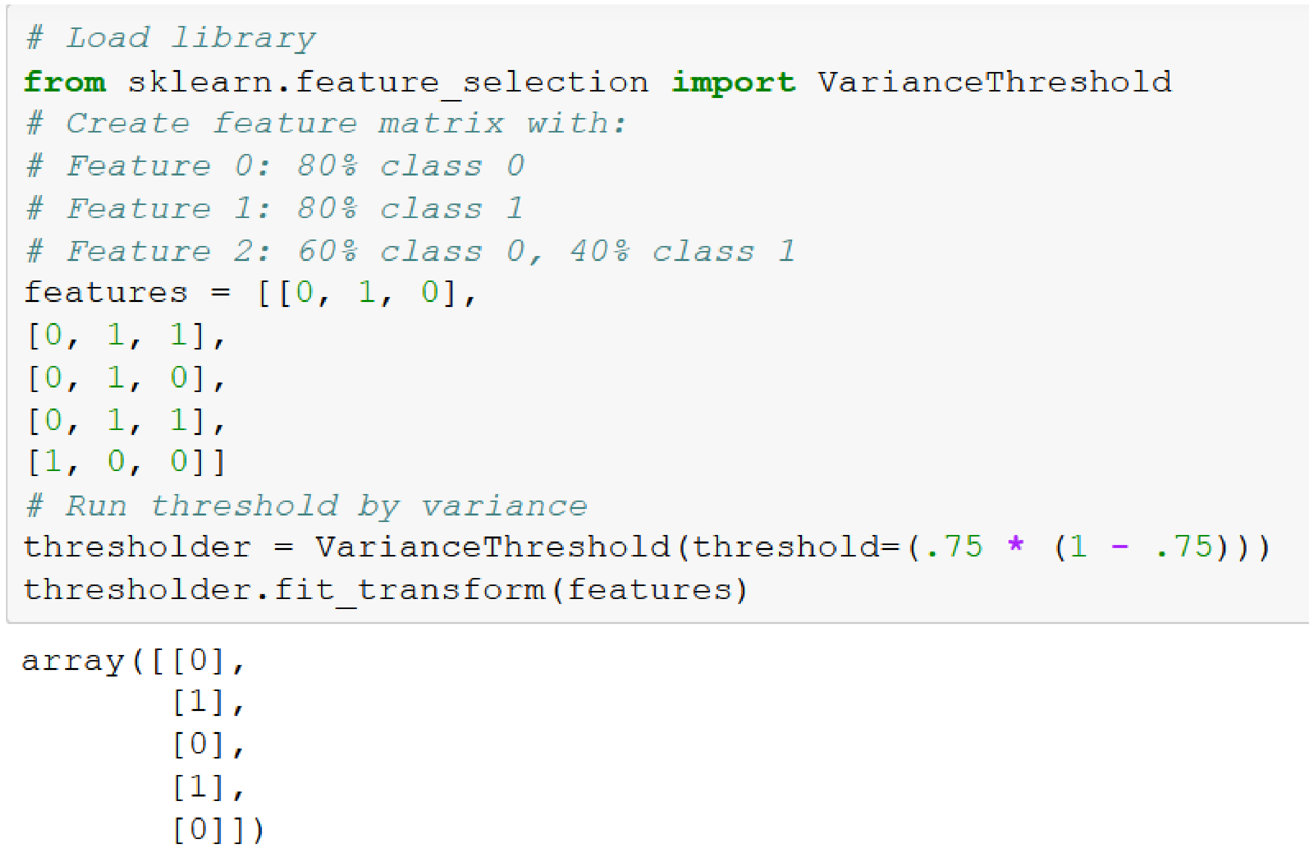
First, the variance is not centered; that is, it is in the squared unit of the feature itself.

Therefore, the VT will not work when feature sets contain different units (e.g., one feature is in years while a different feature is in dollars).

Second, the variance threshold is selected manually, so we have to use our own judgment for a good value to select. We can see the variance for each feature using variances_.

Finally, if the features have been standardized (to mean zero and unit variance), then for obvious reasons variance thresholding will not work correctly.

Thresholding Binary Feature Variance



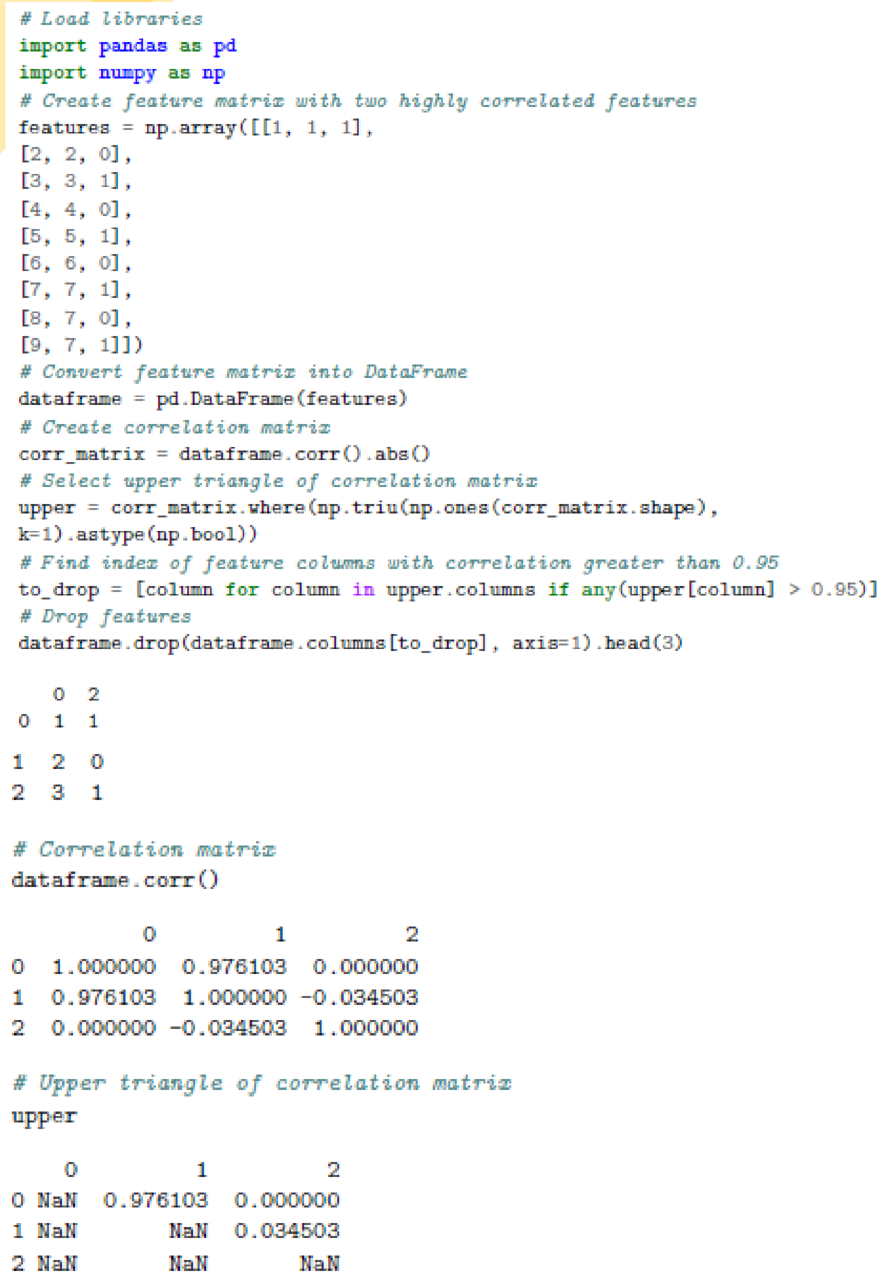
```
# Load library
from sklearn.feature_selection import VarianceThreshold
# Create feature matrix with:
# Feature 0: 80% class 0
# Feature 1: 80% class 1
# Feature 2: 60% class 0, 40% class 1
features = [[0, 1, 0],
[0, 1, 1],
[0, 1, 0],
[0, 1, 1],
[1, 0, 0]]
# Run threshold by variance
thresholder = VarianceThreshold(threshold=(.75 * (1 - .75)))
thresholder.fit_transform(features)
```

```
array([[0],
       [1],
       [0],
       [1],
       [0]])
```

- We have a set of binary categorical features and want to remove those with low variance (i.e., likely containing little information).
- Selecting a subset of features with a Bernoulli random variable variance above a given threshold.
- Just like with numerical features, one strategy for selecting highly informative categorical features is to examine their variances.
- In binary features, variance is calculated as:

$$\text{Var}(x) = p(1 - p)$$

- where p is the proportion of instances of class 1.
- Therefore, by setting p, we can remove features where the vast majority of instances are one class.



```
# Load libraries
import pandas as pd
import numpy as np
# Create feature matrix with two highly correlated features
features = np.array([[1, 1, 1],
[2, 2, 0],
[3, 3, 1],
[4, 4, 0],
[5, 5, 1],
[6, 6, 0],
[7, 7, 1],
[8, 7, 0],
[9, 7, 1]])
# Convert feature matrix into DataFrame
dataframe = pd.DataFrame(features)
# Create correlation matrix
corr_matrix = dataframe.corr().abs()
# Select upper triangle of correlation matrix
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape),
k=1).astype(np.bool))
# Find index of feature columns with correlation greater than 0.95
to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]
# Drop features
dataframe.drop(dataframe.columns[to_drop], axis=1).head(3)
```

```
   0  2
0  1  1
1  2  0
2  3  1
```

```
# Correlation matrix
dataframe.corr()
```

```
          0          1          2
0  1.000000   0.976103   0.000000
1  0.976103   1.000000  -0.034503
2  0.000000  -0.034503   1.000000
```

```
# Upper triangle of correlation matrix
upper
```

```
     0         1         2
0  NaN  0.976103  0.000000
1  NaN       NaN  0.034503
2  NaN       NaN       NaN
```

Handling Highly Correlated Features

We have a feature matrix and we see that some features are highly correlated.

Using a correlation matrix to check for highly correlated features.

If highly correlated features exist, consider dropping one of the correlated features.

One problem we often run into in machine learning is highly correlated features.

If two features are highly correlated, then the information they contain is very similar, and it is likely redundant to include both features.

The solution to highly correlated features is simple: remove one of them from the feature set.

Solution:

1. we create a correlation matrix of all feature
2. we look at the upper triangle of the correlation matrix to identify pairs of highly correlated features:
3. we remove one feature from each of those pairs from the feature set.

From a categorical target vector remove uninformative features

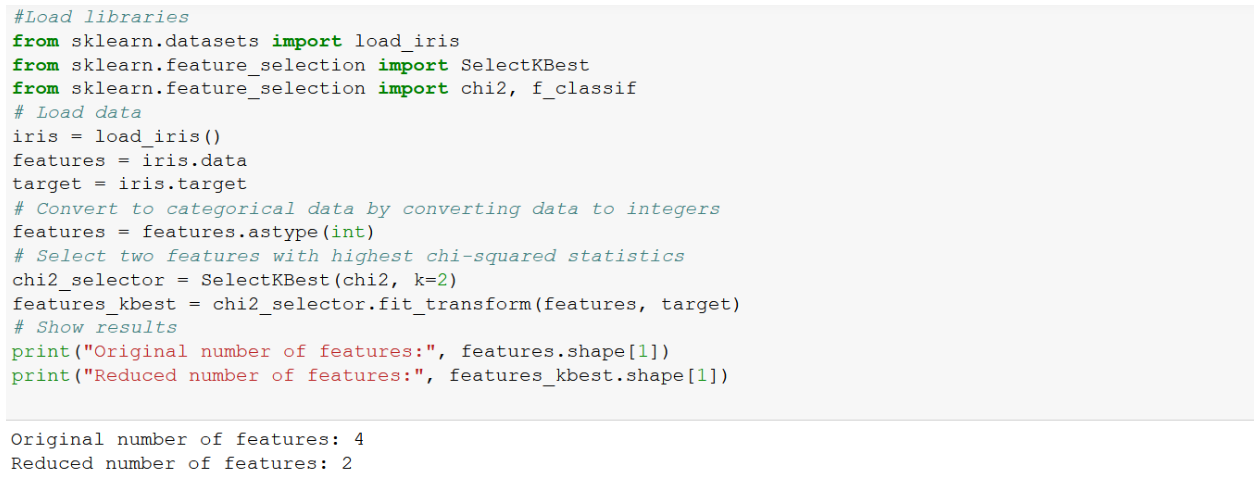
If the features are categorical, calculate a chi-square (χ2) statistic between each feature and the target vector.

(If the features are quantitative, compute the ANOVA F-value between each feature and the target vector instead of selecting a specific number of features, we can also use SelectPercentile to select the top n percent of features).

- Chi-square statistics examines the independence of two categorical vectors.
- That is, the statistic is the difference between the observed number of instances in each class of a categorical feature and what we would expect if that feature was independent (i.e., no relationship) with the target vector:

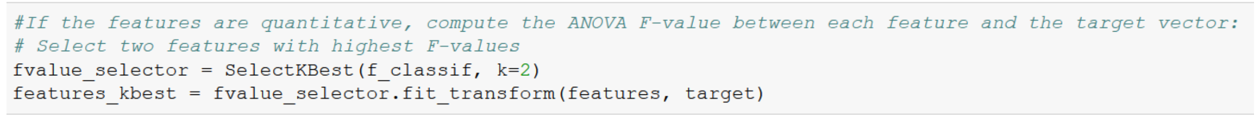
$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

- where *Oi* is the number of instances in class *i* and *Ei* is the number of instances in class *i* we would expect if there is no relationship between the feature and target vector.

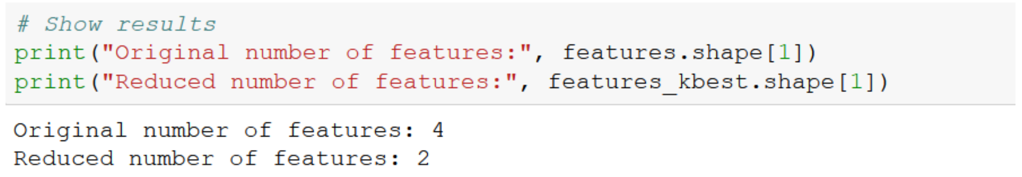


```
#Load libraries
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_classif
# Load data
iris = load_iris()
features = iris.data
target = iris.target
# Convert to categorical data by converting data to integers
features = features.astype(int)
# Select two features with highest chi-squared statistics
chi2_selector = SelectKBest(chi2, k=2)
features_kbest = chi2_selector.fit_transform(features, target)
# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

```
Original number of features: 4
Reduced number of features: 2
```

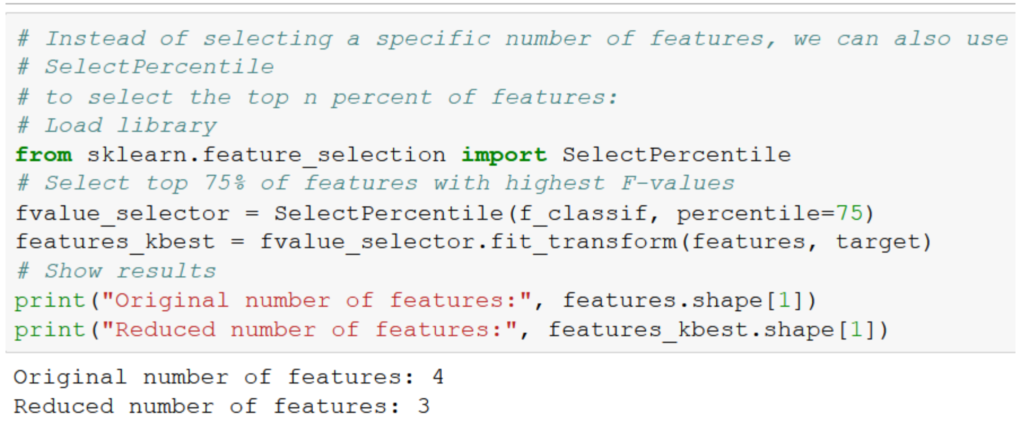


```
#If the features are quantitative, compute the ANOVA F-value between each feature and the target vector:
# Select two features with highest F-values
fvalue_selector = SelectKBest(f_classif, k=2)
features_kbest = fvalue_selector.fit_transform(features, target)
```



```
# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

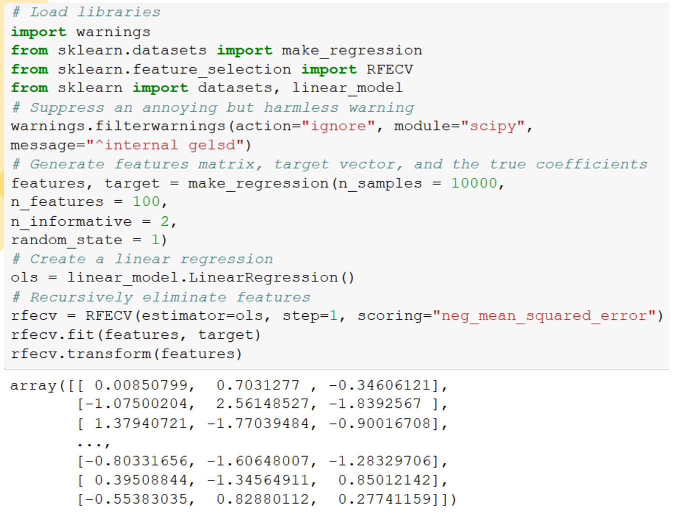
```
Original number of features: 4
Reduced number of features: 2
```



```
# Instead of selecting a specific number of features, we can also use
# SelectPercentile
# to select the top n percent of features:
# Load library
from sklearn.feature_selection import SelectPercentile
# Select top 75% of features with highest F-values
fvalue_selector = SelectPercentile(f_classif, percentile=75)
features_kbest = fvalue_selector.fit_transform(features, target)
# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

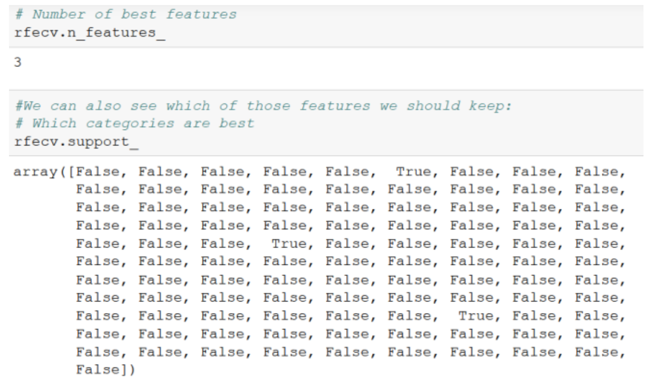
```
Original number of features: 4
Reduced number of features: 3
```

Recursively Eliminating Features



```
# Load libraries
import warnings
from sklearn.datasets import make_regression
from sklearn.feature_selection import RFECV
from sklearn import datasets, linear_model
# Suppress an annoying but harmless warning
warnings.filterwarnings(action="ignore", module="scipy",
message="^internal gelsd")
# Generate features matrix, target vector, and the true coefficients
features, target = make_regression(n_samples = 10000,
n_features = 100,
n_informative = 2,
random_state = 1)
# Create a linear regression
ols = linear_model.LinearRegression()
# Recursively eliminate features
rfecv = RFECV(estimator=ols, step=1, scoring="neg_mean_squared_error")
rfecv.fit(features, target)
rfecv.transform(features)
```

```
array([[ 0.00850799,  0.7031277 , -0.34606121],
       [-1.07500204,  2.56148527, -1.8392567 ],
       [ 1.37940721, -1.77039484, -0.90016708],
       ...,
       [-0.80331656, -1.60648007, -1.28329706],
       [ 0.39508844, -1.34564911,  0.85012142],
       [-0.55383035,  0.82880112,  0.27741159]])
```

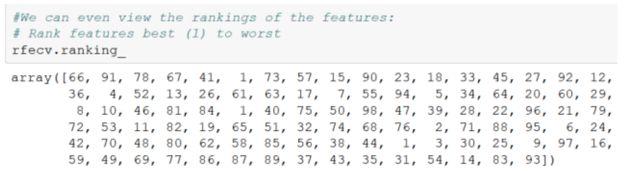


```
# Number of best features
rfecv.n_features_
```

```
3
```

```
#We can also see which of those features we should keep:
# Which categories are best
rfecv.support_
```

```
array([False, False, False, False, False,  True, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False,  True, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False])
```



```
#We can even view the rankings of the features:
# Rank features best (1) to worst
rfecv.ranking_
```

```
array([66, 91, 78, 67, 41,  1, 73, 57, 15, 90, 23, 18, 33, 45, 27, 92, 12,
       36,  4, 52, 13, 26, 61, 63, 17,  7, 55, 94,  5, 34, 64, 20, 60, 29,
        8, 10, 46, 81, 84,  1, 40, 75, 50, 98, 47, 39, 28, 22, 96, 21, 79,
       72, 53, 11, 82, 19, 65, 51, 32, 74, 68, 76,  2, 71, 88, 95,  6, 24,
       42, 70, 48, 80, 62, 58, 85, 56, 38, 44,  1,  3, 30, 25,  9, 97, 16,
       59, 49, 69, 77, 86, 87, 89, 37, 43, 35, 31, 54, 14, 83, 93])
```

- The method RFECV allow recursive feature elimination (RFE) using crossvalidation (CV).
- That is, repeatedly train a model, each time removing a feature until model performance (e.g., accuracy) becomes worse.
- The remaining features are the best: Once we have conducted RFE, we can see the number of features we should keep.
- The idea behind RFE is to train a model that contains some parameters (also called weights or coefficients) like linear regression or support vector machines repeatedly.
- The first time we train the model, we include all the features.
- Then, we find the feature with the smallest parameter (notice that this assumes the features are either rescaled or standardized), meaning it is less important, and remove the feature from the feature set.

- **How many features should we keep?**
- We can (hypothetically) repeat this loop until we only have one feature left. A better approach requires that we include a new concept called cross-validation (CV).
  - Given data containing 1) a target we want to predict and 2) a feature matrix, first we split the data into two groups: a training set and a test set.
  - Second, we train our model using the training set.
  - Third, we pretend that we do not know the target of the test set, and apply our model to the test set's features in order to predict the values of the test set.
  - Finally, we compare our predicted target values with the true target values to evaluate our model.
- We can use CV to find the optimum number of features to keep during RFE.
- Specifically, in RFE with CV after every iteration, we use cross-validation to evaluate our model. If CV shows that our model improved after we eliminated a feature, then we continue on to the next loop.
- However, if CV shows that our model got worse after we eliminated a feature, we put that feature back into the feature set and select those features as the best.
- In scikit-learn, RFE with CV is implemented using RFECV and contains a number of important parameters. The estimator parameter determines the type of model we want to train (e.g., linear regression). The step regression sets the number or proportion of features to drop during each loop. The scoring parameter sets the metric of quality we use to evaluate our model during cross-validation.

Other useful materials:

- Data Preprocessing and Machine Learning with Scikit-Learn
- https://github.com/rasbt/stat451-machine-learning-fs20/blob/master/L05/code/05-preprocessing-and-sklearn__notes.ipynb

- Working with Heterogenous Datasets:
- https://github.com/rasbt/stat451-machine-learning-fs20/blob/master/L05/code/05-bonus-column-transformer.ipynb