CHAPTER 10

Dimensionality Reduction Using Feature Selection

10.0 Introduction

In Chapter 9, we discussed 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. This is called feature extraction. In this chapter we will cover an alternative approach: selecting high-quality, informative features and dropping less useful features. This is called feature selection.

There are three types of feature selection methods: filter, wrapper, and embedded. 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. Finally, embedded methods select the best feature subset as part or as an extension of a learning algorithm's training process.

Ideally, we'd describe all three methods in this chapter. However, since embedded methods are closely intertwined with specific learning algorithms, they are difficult to explain prior to a deeper dive into the algorithms themselves. Therefore, in this chapter we cover only filter and wrapper feature selection methods, leaving the discussion of particular embedded methods until the chapters where those learning algorithms are discussed in depth.

10.1 Thresholding Numerical Feature Variance

Problem

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

Solution

Select a subset of features with variances above a given 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]])
```

Discussion

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 calculates the variance of each feature:

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

where x is the feature vector, x_i 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.

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 (or use a model selection technique described in Chapter 12). We can see the variance for each feature using `variances_`:

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

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

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.])
```

10.2 Thresholding Binary Feature Variance

Problem

You have a set of binary categorical features and want to remove those with low variance (i.e., likely containing little information).

Solution

Select a subset of features with a Bernoulli random variable variance above a given threshold:

```
# 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]])
```

Discussion

Just like with numerical features, one strategy for selecting highly informative categorical features is to examine their variances. In binary features (i.e., Bernoulli random variables), variance is calculated as:

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

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

10.3 Handling Highly Correlated Features

Problem

You have a feature matrix and suspect some features are highly correlated.

Solution

Use a correlation matrix to check for highly correlated features. If highly correlated features exist, consider dropping one of the correlated features:

```
# 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

Discussion

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.

In our solution, first we create a correlation matrix of all features:

```
# 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

Second, we look at the upper triangle of the correlation matrix to identify pairs of highly correlated features:

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

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

Third, we remove one feature from each of those pairs from the feature set.

10.4 Removing Irrelevant Features for Classification

Problem

You have a categorical target vector and want to remove uninformative features.

Solution

If the features are categorical, calculate a chi-square (χ^2) statistic between each feature and the 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
```

Discussion

Chi-square statistics examines the independence of two categorical vectors. That is, the statistic is the difference between the observed number of observations 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 O_i is the number of observations in class i and E_i is the number of observations in class i we would expect if there is no relationship between the feature and target vector.

A chi-squared statistic is a single number that tells you how much difference exists between your observed counts and the counts you would expect if there were no relationship at all in the population. By calculating the chi-squared statistic between a feature and the target vector, we obtain a measurement of the independence between the two. If the target is independent of the feature variable, then it is irrelevant for our purposes because it contains no information we can use for classification. On the other hand, if the two features are highly dependent, they likely are very informative for training our model.

To use chi-squared in feature selection, we calculate the chi-squared statistic between each feature and the target vector, then select the features with the best chi-square statistics. In scikit-learn, we can use `SelectKBest` to select the features with the best statistics. The parameter `k` determines the number of features we want to keep.

It is important to note that chi-square statistics can only be calculated between two categorical vectors. For this reason, chi-squared for feature selection requires that both the target vector and the features are categorical. However, if we have a numerical feature we can use the chi-squared technique by first transforming the quantitative feature into a categorical feature. Finally, to use our chi-squared approach, all values need to be non-negative.

Alternatively, if we have a numerical feature we can use `f_classif` to calculate the ANOVA F-value statistic with each feature and the target vector. F-value scores examine if, when we group the numerical feature by the target vector, the means for each group are significantly different. For example, if we had a binary target vector, gender, and a quantitative feature, test scores, the F-value score would tell us if the mean test score for men is different than the mean test score for women. If it is not, then test score doesn't help us predict gender and therefore the feature is irrelevant.

10.5 Recursively Eliminating Features

Problem

You want to automatically select the best features to keep.

Solution

Use scikit-learn's `RFECV` to conduct recursive feature elimination (RFE) using cross-validation (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:

```
# 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 , -1.2416911 , -0.25651883, -0.10738769],
       [-1.07500204,  2.56148527,  0.5540926 , -0.72602474, -0.91773159],
       [ 1.37940721, -1.77039484, -0.59609275,  0.51485979, -1.17442094],
       ...,
       [-0.80331656, -1.60648007,  0.37195763,  0.78006511, -0.20756972],
       [ 0.39508844, -1.34564911, -0.9639982 ,  1.7983361 , -0.61308782],
       [-0.55383035,  0.82880112,  0.24597833, -1.71411248,  0.3816852 ]])
```

Once we have conducted RFE, we can see the number of features we should keep:

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

```
5
```

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,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False,  True, 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], dtype=bool)
```

We can even view the rankings of the features:

```
# Rank features best (1) to worst
rfecv.ranking_
```

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

Discussion

This is likely the most advanced recipe in this book up to this point, combining a number of topics we have yet to address in detail. However, the intuition is straightforward enough that we can address it here rather than holding off until a later chapter. 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.

The obvious question then is: 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). We will discuss cross-validation in detail in the next chapter, but here is the general idea.

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.

See Also

- Recursive feature elimination with cross-validation