Data Preprocessing

COMP3314
Machine Learning

Introduction

- Preprocessing a dataset is a crucial step
  - Garbage in, garbage out
  - Quality of data and amount of useful information it contains are key factors
- Data-gathering methods are often loosely controlled, resulting in out-of-range values (e.g., Income: −100), impossible data combinations (e.g., Sex: Male, Pregnant: Yes), missing values, etc.
- Preprocessing is often the most important phase of a machine learning project

Outline

- In this chapter you will learn how to …
  - Remove and impute missing values from the dataset
  - Get categorical data into shape
  - Select relevant features
- Specifically, we will looking at the following topics
  - Dealing with missing data
  - Nominal and ordinal features
  - Partitioning a dataset into training and testing sets
  - Bringing features onto the same scale
  - Selecting meaningful features
  - Sequential feature selection algorithms
  - Random forests

Dealing with Missing Data

- Missing data is common in real-world applications
  - Samples might be missing one or more values
- ML models are unable to handle this
- Two ways to handle this
    - Remove entries
    - Imputing missing values from other samples and features

Identifying Missing Values

- Consider the following simple example generated from CSV

```
import pandas as pd
from io import StringIO
import sys
csv_data = \
'''A,B,C,D
1.0,2.0,3.0,4.0
5.0,6.0,,8.0
10.0,11.0,12.0,'''
df = pd.read_csv(StringIO(csv_data))
df
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

Identifying Missing Values

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

- For larger data, it can be tedious to look for missing values
  - Use the isnull method to return a DataFrame with Boolean values that indicate whether a cell
    - contains a numeric value (False), or if
    - data is missing (True)
- Use sum() to count the number of missing values per column

```
df.isnull().sum()
```

```
A    0
B    0
C    1
D    1
dtype: int64
```

Remove Missing Data

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

- One option is to simply remove the corresponding features (columns) or samples (rows)
- Rows with missing values can be dropped via the dropna method with argument axis=0

```
df.dropna(axis=0)
```

	A	B	C	D
0	1.0	2.0	3.0	4.0

- Columns with missing values can be dropped via the dropna method with argument axis=1

```
df.dropna(axis=1)
```

	A	B
0	1.0	2.0
1	5.0	6.0
2	10.0	11.0

Dropna

- The dropna method supports several additional parameters that can come in handy

only drop rows where all columns are NaN

```
df.dropna(how='all')
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

drop rows that have less than 4 real values

```
df.dropna(thresh=4)
```

	A	B	C	D
0	1.0	2.0	3.0	4.0

only drop rows where NaN appear in specific columns (here: 'C')

```
df.dropna(subset=['C'])
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
2	10.0	11.0	12.0	NaN

Remove Missing Data

- Convenient approach
- Disadvantage
  - May remove too many samples
    - Risk losing valuable information
    - Our classifier may need them to discriminate between classes
      - Could make a reliable analysis impossible
- Alternative approach: Interpolation

Interpolation

- Estimate missing values from the other training samples in our dataset
- Example: Mean imputation
  - Replace missing value with the mean value of the entire feature column

```
from sklearn.impute import SimpleImputer
import numpy as np
imr = SimpleImputer(missing_values=np.nan, strategy='mean')
print(df.values)
imr = imr.fit(df.values)
imputed_data = imr.transform(df.values)
print(imputed_data)
```

```
[[ 1.  2.  3.  4.]
 [ 5.  6. nan  8.]
 [10. 11. 12. nan]]
[[ 1.   2.   3.   4. ]
 [ 5.   6.   7.5  8. ]
 [10.  11.  12.   6. ]]
```

Try to change to:
- median
- most_frequent
- constant, fill_value=42

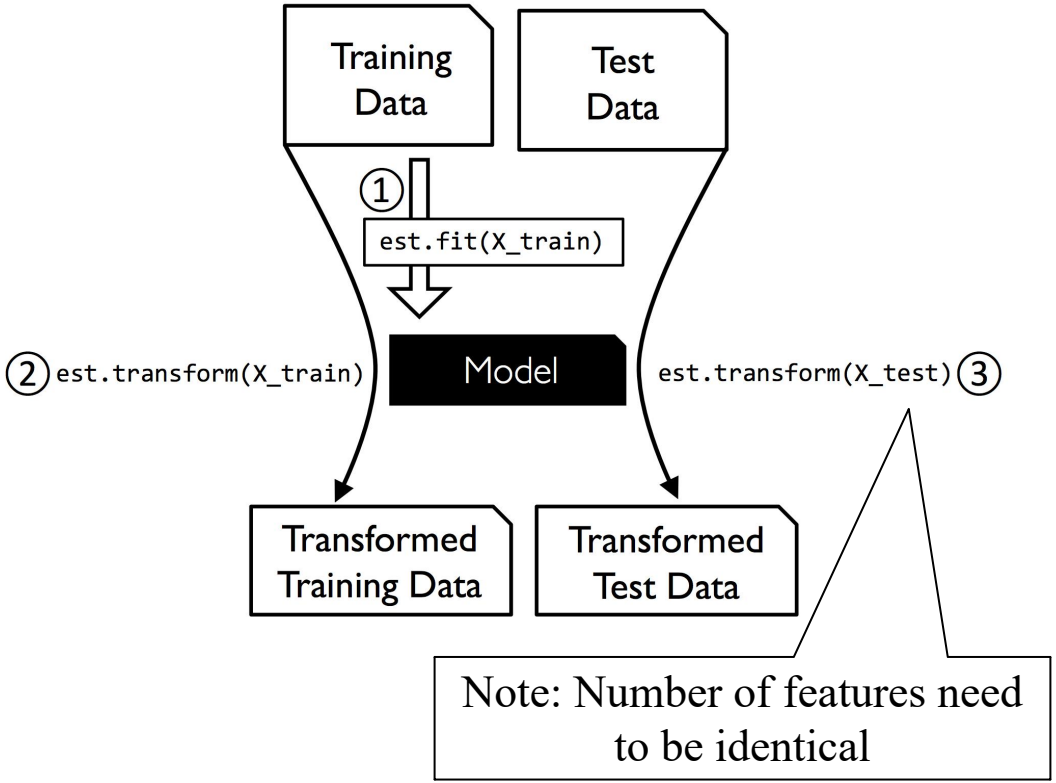
mean and median are for numerical data only, most_frequent and constant can be used for numerical data or strings

Scikit-Learn Estimator API

- SimpleImputer is a Transformer class
  - Used for data transformation
  - Two essential methods
    - fit
    - transform
- Estimator class
  - Very similar to transformer class
  - Two essential methods
    - fit
    - predict
    - Transform (optional)

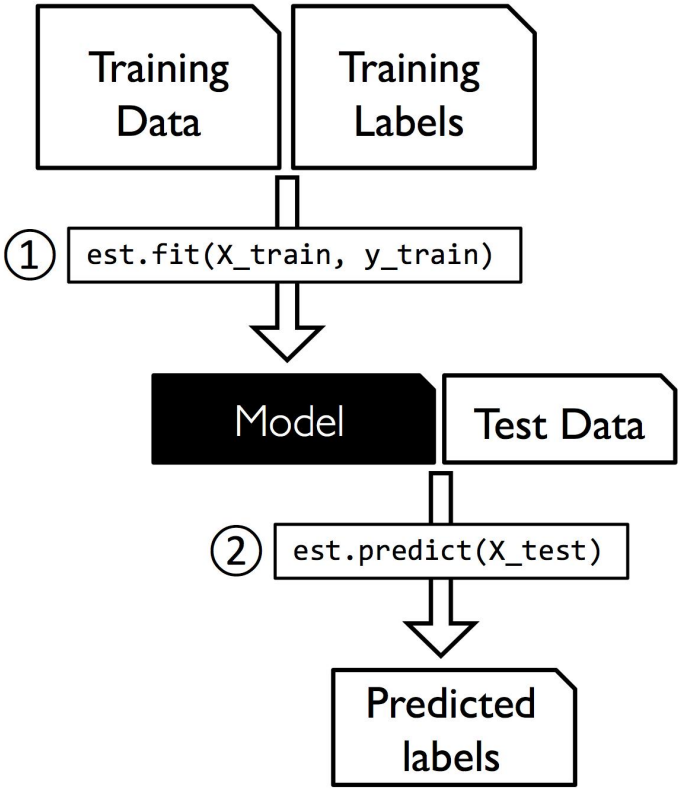
Transformer - Fit and Transform

- fit method
  - Used to learn the parameters from the training data
- transform method
  - Uses those parameters to transform the data



Estimator - Fit and Predict

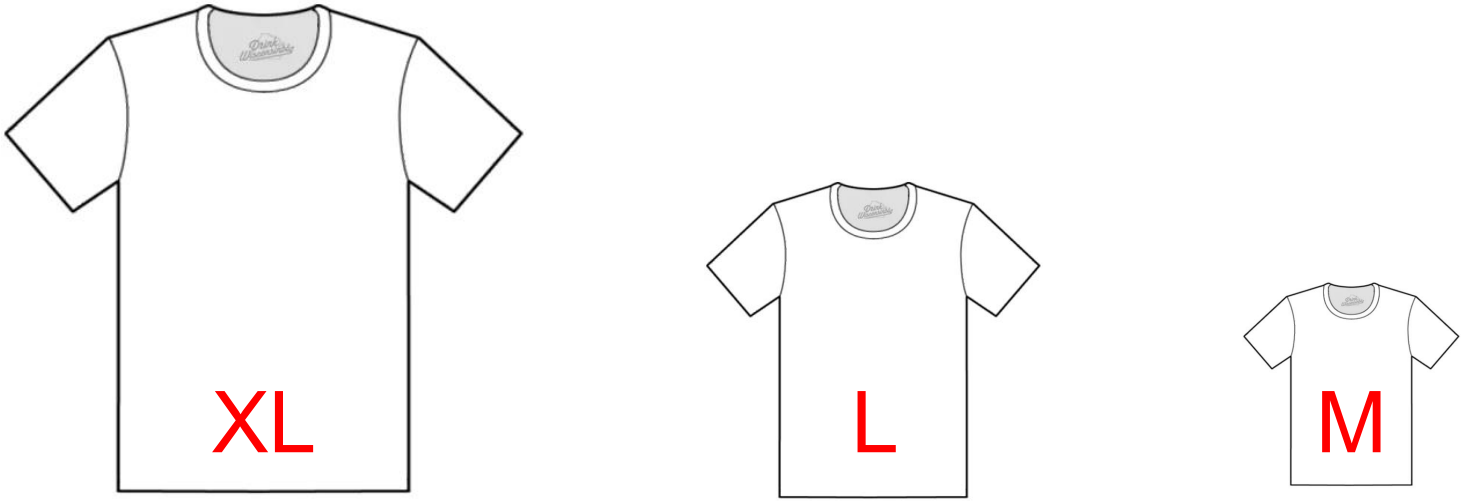
- Use fit method to learn parameters
  - Additionally provide class labels
- Use predict method to make predictions about unlabeled data



Handling Categorical Data

- We have been exclusively working with numerical data
- How to handle categorical data?
- Example of categorical data

A categorical feature can take on one of a limited, and usually fixed, number of possible values



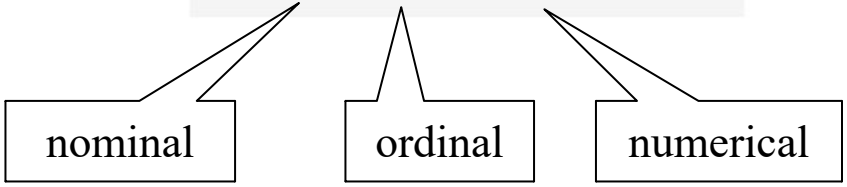
Categorical Data

- It is common that real-world datasets contain categorical features
  - How to deal with this type of data?
- Nominal features vs ordinal features
  - Ordinal features can be sorted / ordered
    - E.g., t-shirt size, because we can define an order XL>L>M
  - Nominal features don't imply any order
    - E.g., t-shirt color

Example Dataset

```
import pandas as pd
df2 = pd.DataFrame([['green', 'M', 10.1, 'class2'],
                    ['red', 'L', 13.5, 'class1'],
                    ['blue', 'XL', 15.3, 'class2']])
df2.columns = ['color', 'size', 'price', 'classlabel']
df2
```

	color	size	price	classlabel
0	green	M	10.1	class2
1	red	L	13.5	class1
2	blue	XL	15.3	class2



Mapping Ordinal Features

	color	size	price	classlabel
0	green	M	10.1	class2
1	red	L	13.5	class1
2	blue	XL	15.3	class2

- To ensure correct interpretation of ordinal features, convert string values to integers

```
size_mapping = {'XL': 3, 'L': 2, 'M': 1}
df2['size'] = df2['size'].map(size_mapping)
df2
```

	color	size	price	classlabel
0	green	1	10.1	class2
1	red	2	13.5	class1
2	blue	3	15.3	class2

- Reverse-mapping to go back

```
inv_size_mapping = {v: k for k, v in size_mapping.items()}
df2['size'] = df2['size'].map(inv_size_mapping)
df2
```

	color	size	price	classlabel
0	green	M	10.1	class2
1	red	L	13.5	class1
2	blue	XL	15.3	class2

Encoding Class Labels

- Most models require integer encoding for class labels
  - Note: class labels are not ordinal, and it doesn't matter which integer number we assign to a particular string label

```
import numpy as np
class_mapping = {label: idx for idx, label in enumerate(np.unique(df2['classlabel']))}
df2['classlabel'] = df2['classlabel'].map(class_mapping)
df2
```

	color	size	price	classlabel
0	green	1	10.1	1
1	red	2	13.5	0
2	blue	3	15.3	1

```
inv_class_mapping = {v: k for k, v in class_mapping.items()}
df2['classlabel'] = df2['classlabel'].map(inv_class_mapping)
df2
```

	color	size	price	classlabel
0	green	1	10.1	class2
1	red	2	13.5	class1
2	blue	3	15.3	class2

LabelEncoder

- Alternatively, there is a convenient LabelEncoder class directly implemented in scikit-learn to achieve this

```
from sklearn.preprocessing import LabelEncoder
class_le = LabelEncoder()
y = class_le.fit_transform(df2['classlabel'].values)
y
```

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

```
class_le.inverse_transform(y)
```

```
array(['class2', 'class1', 'class2'], dtype=object)
```

Shortcut of calling fit and transform separately

One-Hot Encoding

- We could use a similar approach to transform the nominal color column of our dataset, as follows

```
X = df2[['color', 'size', 'price']].values
color_le = LabelEncoder()
X[:, 0] = color_le.fit_transform(X[:, 0])
```

```
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

  - Problem:
    - Model may assume that green > blue, and red > green
    - This could result in suboptimal model
- Workaround: Use one-hot encoding
  - Create a dummy feature for each unique value of nominal features
    - E.g., a blue sample is encoded as blue = 1 , green = 0 , red = 0

One-Hot Encoding

- Use the OneHotEncoder available in scikit-learn's preprocessing module

```
1 from sklearn.preprocessing import OneHotEncoder
2 X = df2[['color', 'size', 'price']].values
3 print(X)
4 color_ohe = OneHotEncoder()
5 print(color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray())
6
```

```
[['green' 1 10.1]
 ['red' 2 13.5]
 ['blue' 3 15.3]]
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
```

-1 means unknown dimension and we want numpy to figure it out

Apply to only a single column

One-Hot Encoding via ColumnTransformer

- To selectively transform columns in a multi-feature array, use ColumnTransformer
  - Accepts a list of (name, transformer, column(s)) tuple

```
1 from sklearn.compose import ColumnTransformer
2 X = df2[['color', 'size', 'price']].values
3 c_transf = ColumnTransformer([
4                                ('onehot', OneHotEncoder(), [0]),
5                                ('nothing', 'passthrough', [1, 2])
6                               ])
7 c_transf.fit_transform(X).astype(float)
```

```
array([[ 0. ,  1. ,  0. ,  1. , 10.1],
       [ 0. ,  0. ,  1. ,  2. , 13.5],
       [ 1. ,  0. ,  0. ,  3. , 15.3]])
```

Only modify the first column

One-Hot Encoding - Via Pandas

- An even more convenient way to create those dummy features via one-hot encoding is to use the get_dummies method implemented in pandas
  - get_dummies will only convert string columns

```
pd.get_dummies(df2[['price', 'color', 'size']])
```

	price	size	color_blue	color_green	color_red
0	10.1	1	0	1	0
1	13.5	2	0	0	1
2	15.3	3	1	0	0

One-Hot Encoding - Dropping First Feature

- Note that we do not lose any information by removing one dummy column
  - E.g., if we remove the column color_blue, the feature information is still preserved since if we observe color_green=0 and color_red=0, it implies that the observation must be blue

```
pd.get_dummies(df2[['price', 'color', 'size']], drop_first=True)
```

	price	size	color_green	color_red
0	10.1	1	1	0
1	13.5	2	0	1
2	15.3	3	0	0

```
ohct.fit_transform(X)[:, 1:]
```

```
array([[1.0, 0.0, 1, 10.1],
       [0.0, 1.0, 2, 13.5],
       [0.0, 0.0, 3, 15.3]], dtype=object)
```

UCI Wine Dataset

- The UCI wine dataset consists of 178 wine samples with 13 features describing their different chemical properties

```
df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data', header=None)
print(df_wine.shape)
```

```
(178, 14)
```

```
df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanaoid phenols', 'Proanthocyanins', 'Color intensity', 'Hue',
                    'OD280/OD315 of diluted wines', 'Proline']
print(np.unique(df_wine['Class label']))
df_wine.head()
```

```
[1 2 3]
```

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanaoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

UCI Wine Dataset: Training-Testing

- Let's first divide the dataset into separate training and testing sets

```
from sklearn.model_selection import train_test_split
X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0, stratify=y)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(124, 13)
(54, 13)
(124,)
(54,)
```

UCI Wine Dataset: Training-Testing

- It is important to balance the trade-off between inaccurate estimation of generalization error and withholding too much information from the learning algorithm
- In practice, the most commonly used splits are 60:40, 70:30, or 80:20, depending on the size of the initial dataset
  - For large datasets, 90:10 or 99:1 splits are also common and appropriate
- Instead of discarding the allocated test data after model training and evaluation, we can retrain a classifier on the entire dataset as it could improve the predictive performance of the model
  - While this approach is generally recommended, it could lead to worse generalization performance

Feature Scaling

- The majority of ML algorithms require feature scaling
  - Decision trees and random forests are two of few ML algorithms that don't require feature scaling
- Importance
  - Consider the squared error function in Adaline for two dimensional features where one feature is measured on a scale from 1 to 10 and the second feature is measured on a scale from 1 to 100,000
    - The second feature would contribute to the error with a much higher significance
- Two common approaches to bring different features onto the same scale
  - Normalization
    - E.g., rescaling features to a range of [0, 1]
  - Standardization
    - E.g., center features at mean 0 with standard deviation 1

Feature Scaling - Normalization

- Most often, normalization refers to the rescaling of features to a range of [0, 1]
- To normalize our data, we can simply apply a min-max scaling to each feature column
  - A new value $x^{(i)}_{norm}$ of a sample $x^{(i)}$ is calculated as follows

$$x^{(i)}_{norm} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$$

  - Here x_{min} is the smallest value in a feature column and x_{max} the largest

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.transform(X_test)
```

```
print(X_train[:,0].max(axis=0))
print(X_train[:,0].min(axis=0))
print(X_train_norm[:,0].max(axis=0))
print(X_train_norm[:,0].min(axis=0))
print(X_test_norm[:,0].max(axis=0))
print(X_test_norm[:,0].min(axis=0))
```

```
14.83
11.41
1.0
0.0
0.871345029239766
-0.11111111111111116
```

Feature Scaling - Standardization

- Standardization is more practical for various reasons including retaining useful information about outliers
- A new value $x^{(i)}_{std}$ of a sample $x^{(i)}$ is calculated as follows

$$x^{(i)}_{std} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

- Here μ_x is the sample mean of feature column and σ_x the corresponding standard deviation
- Similar to the MinMaxScaler class, scikit-learn also implements a class for standardization

```
from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.fit_transform(X_test)
```

```
print(X_train[:,0].mean(axis=0))
print(X_train[:,0].std(axis=0))
print(X_train_std[:,0].mean(axis=0))
print(X_train_std[:,0].std(axis=0))
```

```
13.033548387096777
0.8233685663454646
6.87935371750596e-15
0.9999999999999998
-1.133095674900999e-15
1.0
```

Normalization vs. Standardization

- The following example illustrates the difference between standardization and normalization

```
ex = np.array([0, 1, 2, 3, 4, 5])
```

```
print('standardized: ', (ex-ex.mean()) / ex.std())
print('normalized: ', (ex-ex.min()) / ex.max() - ex.min())
```

```
standardized:  [-1.46385011 -0.87831007 -0.29277002  0.29277002  0.87831007  1.46385011]
normalized:  [0.  0.2 0.4 0.6 0.8 1. ]
```

Robust Scaler

- More advanced methods for feature scaling are available in sklearn
- The RobustScaler is especially helpful and recommended if working with small datasets that contain many outliers

Feature Selection

- Selects a subset of relevant features
  - Simplify model for easier interpretation
  - Shorten training time
  - Avoid curse of dimensionality
  - Reduce overfitting
- Feature selection ≠ feature extraction (covered in next chapter)
  - Selecting subset of the features ≠ creating new features
- We are going to look at two techniques for feature selection
  - L1 Regularization
  - Sequential Backward Selection (SBS)

L1 vs. L2 Regularization

- L2 regularization (penalty) used in chapter 3

$$L2: \|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{m} w_j^2$$

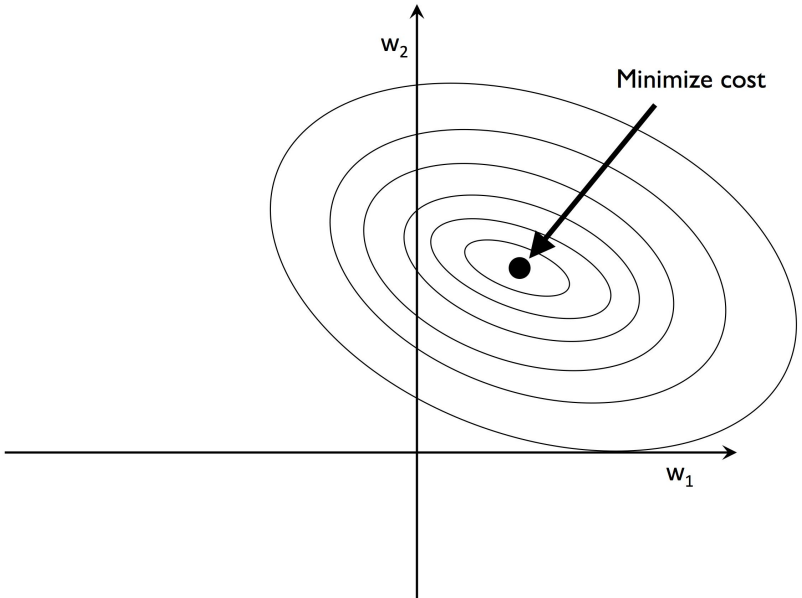
- Another approach: L1 regularization (penalty)

$$L1: \|\boldsymbol{w}\|_1 = \sum_{j=1}^{m} |w_j|$$

  - This will usually yield sparse feature weights
    - Most feature weights will be zero
  - Sparsity can be useful in practice if we have a high dimensional dataset with many features that are irrelevant
  - L1 regularization can be taken as a technique for feature selection

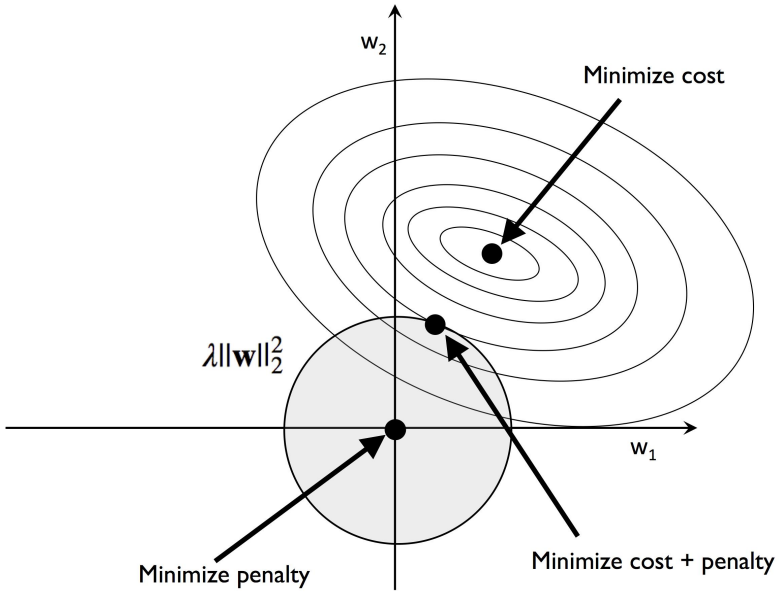
Geometric Interpretation

- To better understand how L1 regularization encourages sparsity, let's take a look at a geometric interpretation of regularization
- Consider the sum of squared errors cost function used for Adaline
- Plot of the contours of a convex cost function for two coefficients w_1 and w_2



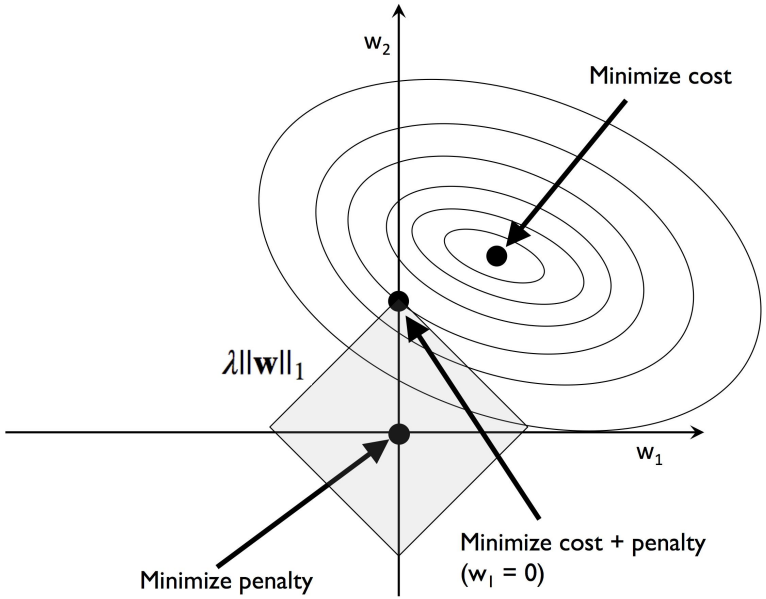
Geometric Interpretation: L2 Regularization

- Regularization adds a penalty to the cost function to encourage smaller weights
  - By increasing the regularization strength λ we shrink the weights towards zero and decrease the dependency of our model on the training data



Geometric Interpretation: L1 Regularization

- Since the L1 penalty is the sum of the absolute weight coefficients we can represent it as a diamond-shape
- It is more likely that the optimum is located on the axes, which encourages sparsity



Mathematical details can be found in
Section 3.4 of
The Elements of Statistical Learning

Sparse Solution

- We can simply set the penalty parameter to ‘l1’ for models in scikit-learn that support L1 regularization

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(penalty = 'l1', C=1.0, solver='liblinear', multi_class='ovr')
lr.fit(X_train_std, y_train)
print('Training accuracy:',lr.score(X_train_std, y_train))
print('Test accuracy:', lr.score(X_test_std, y_test))
```

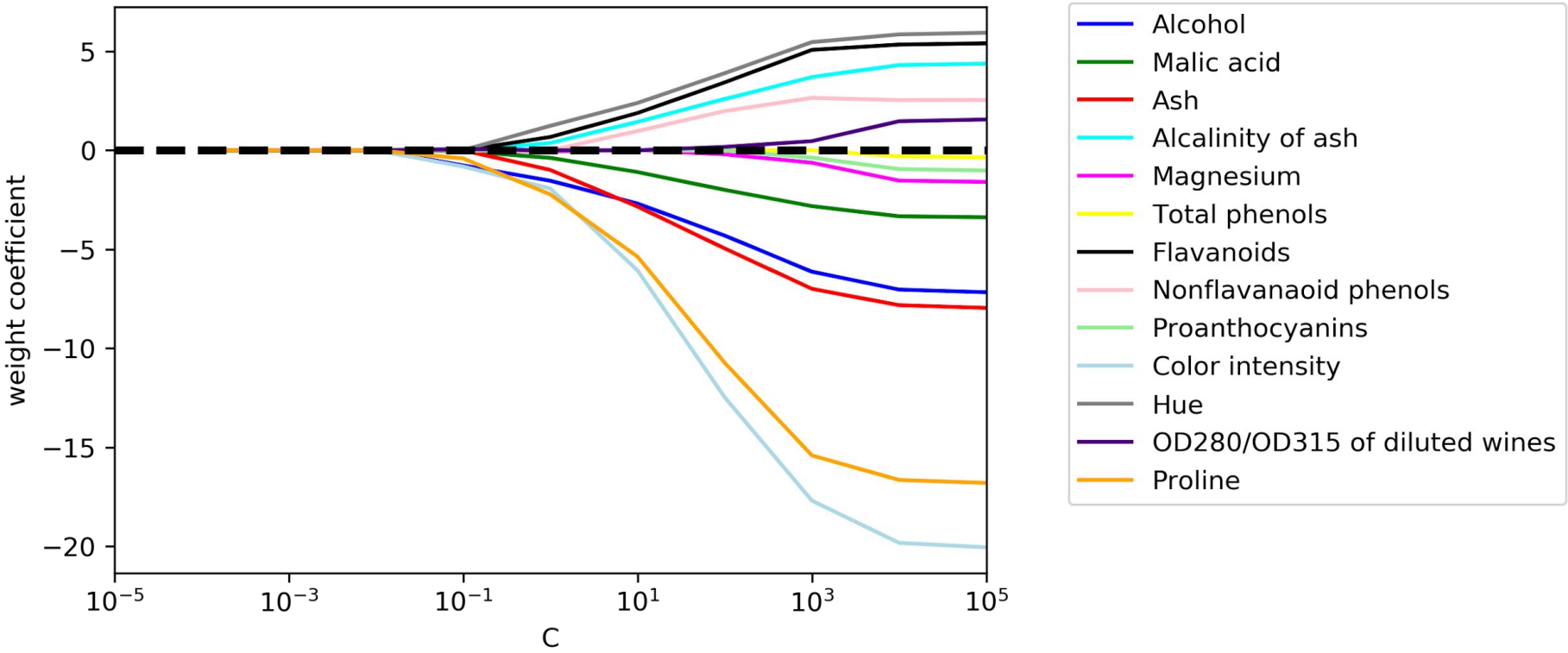
```
Training accuracy: 1.0
Test accuracy: 1.0
```

- In scikit-learn, w_0 corresponds to intercept_ and w_j (for $j > 0$) corresponds to the values in coef_

```
print(lr.intercept_)
print(lr.coef_)
```

```
[-1.26353461 -1.21581767 -2.37019736]
[[ 1.24608399  0.18063288  0.74634041 -1.16426973  0.          0.
   1.15915337  0.          0.          0.          0.          0.55745008
   2.50895528]
 [-1.53798763 -0.38660209 -0.99544411  0.36444033 -0.059204    0.
   0.6676744   0.          0.         -1.93317791  1.23538017  0.
  -2.232327  ]
 [ 0.13571903  0.16849144  0.35719201  0.          0.          0.
  -2.4378634   0.          0.          1.56356565 -0.81863112 -0.49263067
   0.        ]]
```

Sparse Solution - Regularization Strength



Sequential Backward Selection (SBS)

- Reduces an initial d-dimensional space to a k-dimensional subspace ($k < d$) by automatically selecting features that are most relevant
- Idea:
  - Sequentially remove features until desired feature number is reached
  - Define a criterion function J to be maximized
    - E.g., performance of the classifier after removal
      - Use a validation subset of the training set for performance evaluation
  - Eliminate the feature that causes the least performance loss

SBS

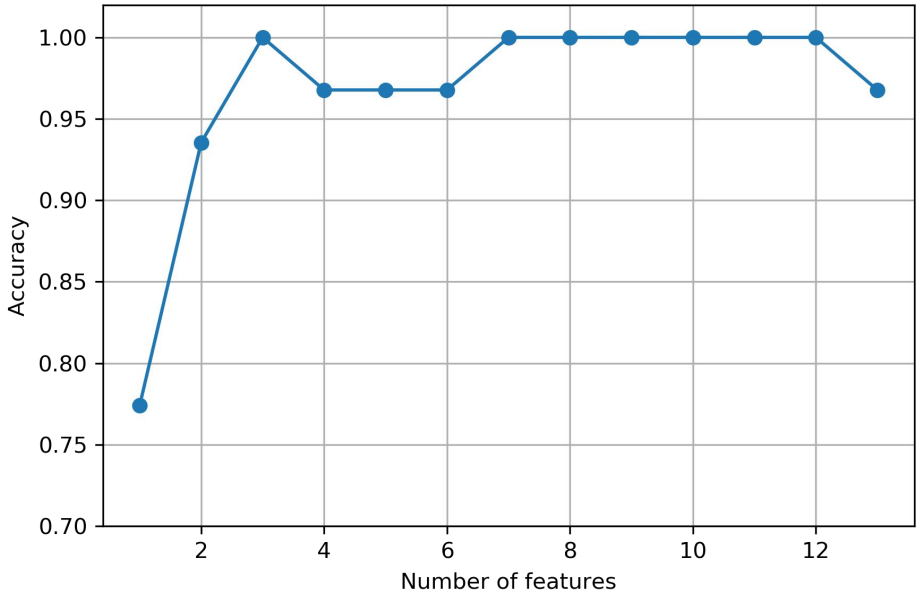
Steps:

1. Initialize the algorithm with $k = d$
   d is the dimensionality of the full feature space $\mathbf{X_d}$
2. Determine the feature $\mathbf{x}^- = \text{argmax } J(\mathbf{X_k} - \mathbf{x})$ that maximizes the criterion function J
3. Remove the feature $\mathbf{x}^-$ from the feature set
   $\mathbf{X_{k-1}} = \mathbf{X_k} - \mathbf{x}^-$
   $k = k - 1$
4. Terminate if k equals the number of desired features;
   otherwise, go to step 2

- In the following we will implement SBS in Python from scratch

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
class SBS():
    def __init__(self, estimator, k_features, scoring=accuracy_score, test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state
    def fit(self, X, y):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=self.test_size, random_state=self.random_state)
        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train, X_test, y_test, self.indices_)
        self.scores_ = [score]
        while dim > self.k_features:
            scores = []; subsets = []
            for p in combinations(self.indices_, r=dim - 1):
                score = self._calc_score(X_train, y_train, X_test, y_test, p)
                scores.append(score)
                subsets.append(p)
            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1
            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]
        return self
    def transform(self, X):
        return X[:, self.indices_]
    def _calc_score(self, X_train, y_train, X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

```
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5)
sbs = SBS(knn, k_features=1)
sbs.fit(X_train_std, y_train)
k_feat = [len(k) for k in sbs.subsets_]
plt.plot(k_feat, sbs.scores_, marker='o')
plt.ylim([0.7, 1.02])
plt.ylabel('Accuracy')
plt.xlabel('Number of features')
plt.grid()
plt.tight_layout()
plt.show()
```



SBS - Analyzing the Result

- The smallest feature subset (k = 3) that yielded such a good performance on the validation dataset has the following features

```
k3 = list(sbs.subsets_[10])
print(df_wine.columns[1:][k3])
```

```
Index(['Alcohol', 'Malic acid', 'OD280/OD315 of diluted wines'], dtype='object')
```

- The accuracy of the KNN classifier on the original test set is as follows

```
knn.fit(X_train_std, y_train)
print('Training accuracy:', knn.score(X_train_std, y_train))
print('Test accuracy:', knn.score(X_test_std, y_test))
```

```
Training accuracy: 0.967741935483871
Test accuracy: 0.9814814814814815
```

- The three-feature subset has the following accuracy

```
knn.fit(X_train_std[:, k3], y_train)
print('Training accuracy:', knn.score(X_train_std[:, k3], y_train))
print('Test accuracy:', knn.score(X_test_std[:, k3], y_test))
```

```
Training accuracy: 0.9516129032258065
Test accuracy: 0.9259259259259259
```

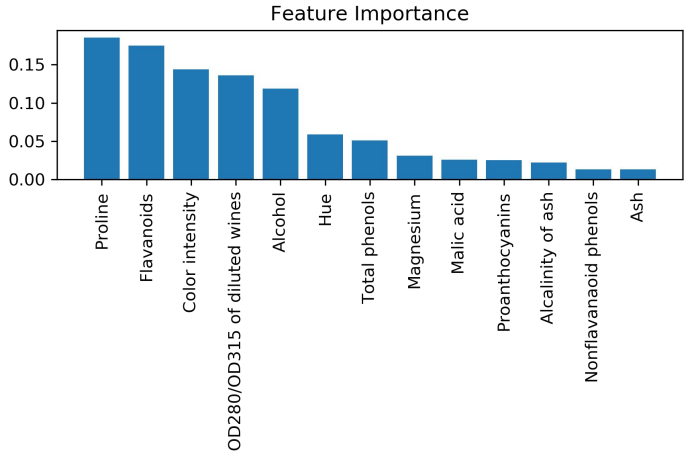
Feature Selection Algorithms in scikit-learn

- There are many more feature selection algorithms available via scikit-learn
- A comprehensive discussion of the different feature selection methods is beyond the scope of this lecture
  - A good summary with illustrative examples can be found here

Assessing Feature Importance

- We can determine relevant features using random forest
  - Measure the feature importance as the averaged information gain
- The random forest implementation in scikit-learn already collects the feature importance values for us
  - Access them via the feature_importances_ attribute after fitting a RandomForestClassifier
- In the following we will train a forest of 500 trees on the Wine dataset and rank the 13 features by their respective importance measures

```
from sklearn.ensemble import RandomForestClassifier
feat_labels = df_wine.columns[1:]
forest = RandomForestClassifier(n_estimators=500, random_state=1)
forest.fit(X_train, y_train)
importances = forest.feature_importances_
indices = np.argsort(importances)[::-1]
for f in range(X_train.shape[1]):
    print("%2d) %-*s %f" % (f + 1, 30, feat_labels[indices[f]], importances[indices[f]]))
plt.title('Feature Importance')
plt.bar(range(X_train.shape[1]), importances[indices], align='center')
plt.xticks(range(X_train.shape[1]), feat_labels[indices], rotation=90)
plt.xlim([-1, X_train.shape[1]])
plt.tight_layout()
plt.show()
```

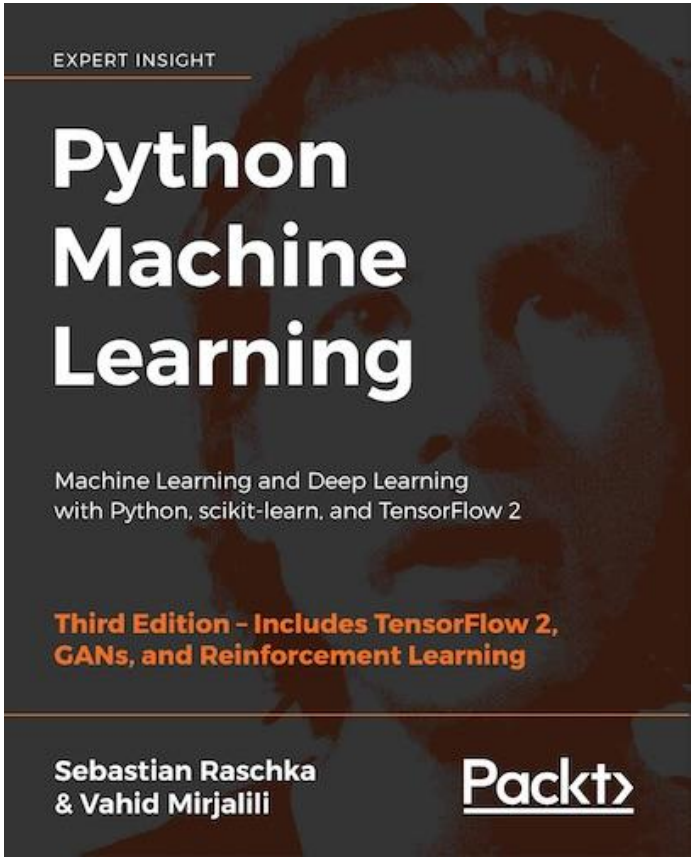


Conclusion

- Handle missing data correctly
- Encode categorical variables correctly
- Map ordinal and nominal feature values to integer representations
- L1 regularization can help us to avoid overfitting by reducing the complexity of a model
- Used a sequential feature selection algorithm to select meaningful features from a dataset

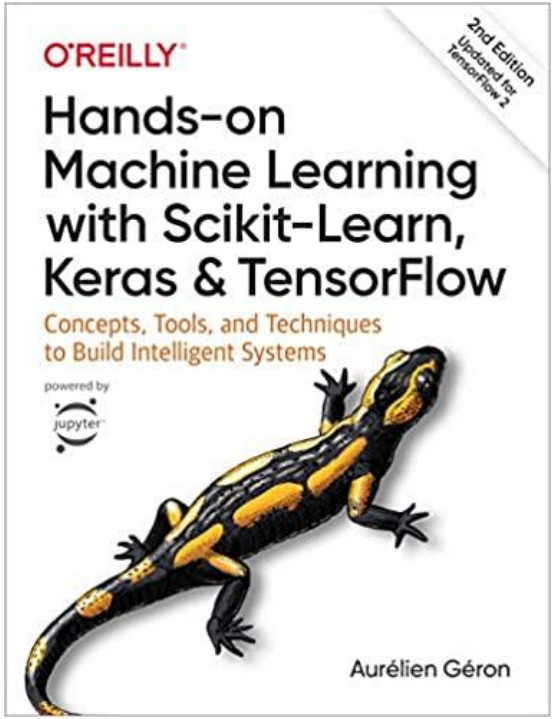
References

- Most materials in this chapter are based on
  - Book
  - Code



References

- Some materials in this chapter are based on
  - Book
  - Code



References

- The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition
  - Trevor Hastie, Robert Tibshirani, Jerome Friedman
- https://web.stanford.edu/~hastie/ElemStatLearn/
- Pandas User Guide: Working with missing data