Overview

Data Mining for Business Analytics in Python

Shmueli, Bruce, Gedeck, & Patel

 rev 10/17/19

Core Ideas in Data Mining

- Classification
- Prediction
- Association Rules & Recommenders
- Predictive Analytics
- Data & Dimension Reduction
- Data Exploration & Visualization
- Supervised & Unsupervised Learning

Paradigms for Data Mining (variations)

SEMMA (from SAS)

- Sample
- Explore
- Modify
- Model
- Assess

CRISP-DM (SPSS/IBM)

- Business Understanding
- Data Understanding
- Data Preparation
- Modeling
- Evaluation
- Deployment

Supervised Learning

Goal: Predict a single “target” or “outcome” variable

Training data, where target value is known

Score to data where target value is not known

Methods: Classification and Prediction

Unsupervised Learning

Goal: Segment data into meaningful segments; detect patterns

There is no target (outcome) variable to predict or classify

Methods: Association rules, collaborative filters, data reduction & exploration, visualization

Supervised: Classification

Goal: Predict categorical target (outcome) variable

Examples: Purchase/no purchase, fraud/no fraud, creditworthy/not creditworthy…

Each row is a case (customer, tax return, applicant)

Each column is a variable

Target variable is often binary (yes/no)

Supervised: Prediction

Goal: Predict numerical target (outcome) variable

Examples: sales, revenue, performance

As in classification: Each row is a case (customer, tax return, applicant)

Each column is a variable

Taken together, classification and prediction constitute “predictive analytics”

Unsupervised: Association Rules

Goal: Produce rules that define “what goes with what” in transactions

Example: “If X was purchased, Y was also purchased”

Rows are transactions

Used in recommender systems – “Our records show you bought X, you may also like Y”

Also called “affinity analysis”

Unsupervised: Collaborative Filtering

Goal: Recommend products to purchase

Based on products that customer rates, selects, views, or purchases

Recommend products that “customers like you” purchase (user-based)

Or, recommend products that share a “product purchaser profile” with your purchases (item-based)

Unsupervised: Data Reduction

Distillation of complex/large data into simpler/smaller data

Reducing the number of variables/columns (e.g., principal components)

Reducing the number of records/rows (e.g., clustering)

Unsupervised: Data Visualization

Graphs and plots of data

Histograms, boxplots, bar charts, scatterplots

Especially useful to examine relationships between pairs of variables

Data Exploration

Data sets are typically large, complex & messy

Need to review the data to help refine the task

Use techniques of Reduction and Visualization

The Process of Data Mining

Steps in Data Mining

1. Define/understand purpose
2. Obtain data (may involve random sampling)
3. Explore, clean, pre-process data
4. Reduce the data; if supervised DM, partition it
5. Specify task (classification, clustering, etc.)
6. Choose the techniques (regression, CART, neural networks, etc.)
7. Iterative implementation and “tuning”
8. Assess results – compare models
9. Deploy best model

Example:
West Roxbury Housing Data

TABLE 2.1 DESCRIPTION OF VARIABLES IN WEST ROXBURY (BOSTON) HOME VALUE DATASET

TOTAL VALUE	Total assessed value for property, in thousands of USD
TAX	Tax bill amount based on total assessed value multiplied by the tax rate, in USD
LOT SQ FT	Total lot size of parcel in square feet
YR BUILT	Year the property was built
GROSS AREA	Gross floor area
LIVING AREA	Total living area for residential properties (ft^2)
FLOORS	Number of floors
ROOMS	Total number of rooms
BEDROOMS	Total number of bedrooms
FULL BATH	Total number of full baths
HALF BATH	Total number of half baths
KITCHEN	Total number of kitchens
FIREPLACE	Total number of fireplaces
REMODEL	When the house was remodeled (Recent/Old/None)

Preliminary Exploration in Python

loading data, viewing it, summary statistics

1. Open Anaconda-Navigator and launch a 'jupyter' notebook. It opens a new browser window.
2. Navigate to the directory where your csv file is saved and open a new Python notebook.

```
import pandas as pd

# Load data
housing_df = pd.read_csv('WestRoxbury.csv')
housing_df.shape #find dimension of data
frame
housing_df.head()  #show the 1st five rows
print(housing_df)  #show all the data
```

Data Exploration in Python, cont.

```
# Rename columns: replace spaces with '_'
housing_df = housing_df.rename
    (columns={'TOTAL VALUE ': 'TOTAL_VALUE'})
    # explicit
housing_df.columns = [s.strip().replace(' ',
    '_') for s in housing_df.columns] # all
    columns

# Show first four rows of the data
housing_df.loc[0:3]
  # loc[a:b] gives rows a to b, inclusive
housing_df.iloc[0:4]
 # iloc[a:b] gives rows a to b-1
```

Data Exploration in Python, cont.

```
# Different ways of showing the first 10
# values in column TOTAL_VALUE

housing_df['TOTAL_VALUE'].iloc[0:10]
housing_df.iloc[0:10]['TOTAL_VALUE']
housing_df.iloc[0:10].TOTAL_VALUE  # use dot
   notation if the column name has no spaces

# Show the fifth row of the first 10 columns
housing_df.iloc[4][0:10]
housing_df.iloc[4, 0:10]
housing_df.iloc[4:5, 0:10]  # use a slice to
   return a data frame
```

Data Exploration in Python, cont.

```
# Use pd.concat to combine non-consecutive
# columns into a new data frame. Axis
# argument specifies dimension along which
# concatenation happens, 0=rows, 1=columns.
pd.concat([housing_df.iloc[4:6,0:2],
housing_df.iloc[4:6,4:6]], axis=1)

# To specify a full column, use:
housing.iloc[:,0:1]
housing.TOTAL_VALUE
housing_df['TOTAL_VALUE'][0:10]  # show the
  first 10 rows of the first column
```

Data Exploration in Python, cont.

```
# Descriptive statistics

print('Number of rows ',
len(housing_df['TOTAL_VALUE'])) # show length
  of first column
print('Mean of TOTAL_VALUE ',
housing_df['TOTAL_VALUE'].mean()) # show mean
  of column
housing_df.describe() # show summary
  statistics for each column
```

Import Needed Functionality

```
import numpy as np
import pandas as pd # repeating - we did this
  earlier
from sklearn.model_selection import
train_test_split
from sklearn.metrics import r2_score
from sklearn.linear_model import
  LinearRegression
```

The abbreviations *pd*, *np*, and *sm* are commonly used in the data science community.

Obtaining Data: Sampling

Data mining typically deals with huge databases

For piloting/prototyping, algorithms and models are typically applied to a sample from a database, to produce statistically-valid results

Once you develop and select a final model, you use it to “score” (predict values or classes for) the observations in the larger database

Rare Event Oversampling

The event of interest is rare

Examples: response to mailing, fraud in taxes, …

Sampling may yield too few “interesting” cases to effectively train a model

A popular solution: oversample the rare cases to obtain a more balanced training set

Later, need to adjust results for the oversampling

Sampling & Oversampling

```
# random sample of 5 observations
housing_df.sample(5)

# oversample houses with over 10 rooms
weights = [0.9 if rooms > 10 else 0.1 for
  rooms in housing_df.ROOMS]
housing_df.sample(5, weights=weights)
```

Types of Variables

Determine the types of pre-processing needed, and algorithms used

Main distinction: Categorical vs. numeric

Numeric

- Continuous
- Integer

Categorical

- Ordered (low, medium, high)
- Unordered (male, female)

Variable handling

- Numeric
  - Most algorithms can handle numeric data
  - May occasionally need to “bin” into categories
- Categorical
  - Naïve Bayes can use as-is
  - In most other algorithms, must create binary dummies (number of dummies = number of categories – 1) [see Table 2.6 for R code]

Reviewing Variables

```
housing_df.columns  # print a list of
  variables

# REMODEL needs to be converted to a
  categorical variable
housing_df.REMODEL =
housing_df.REMODEL.astype('category')
housing_df.REMODEL.cat.categories  # Show
  number of categories
housing_df.REMODEL.dtype  # Check type of
  converted variable
```

output on next slide…

Partial output of code on previous slide

```
> housing_df.columns
Index(['TOTAL_VALUE', 'TAX', 'LOT_SQFT',
   'YR_BUILT', 'GROSS_AREA','LIVING_AREA',
   'FLOORS', 'ROOMS', 'BEDROOMS', 'FULL_BATH',
   'HALF_BATH','KITCHEN', 'FIREPLACE',
   'REMODEL'], dtype='object')

> housing_df.REMODEL.cat.categories
Index(['None', 'Old', 'Recent'], dtype='object')

> housing_df.REMODEL.dtype category
```

Creating binary dummies

```
# the missing values will create a third
category
# use drop_first=True to drop the first dummy
  variable
# use dummy_na=True to add a column for NaNs
(Not a Number)
housing_df = pd.get_dummies(housing_df,
  prefix_sep='_', dtype=int)
housing_df.columns
housing_df.loc[:,'REMODEL_Old':
  'REMODEL_Recent'].head(5)
```

output on next slide…

Creating Binary Dummies – Output

	REMODEL_Old	REMODEL_Recent
0	0	0
1	0	1
2	0	0
3	0	0
4	0	0

Detecting Outliers

An outlier is an observation that is “extreme”, being distant from the rest of the data (definition of “distant” is deliberately vague)

Outliers can have disproportionate influence on models (a problem if it is spurious)

An important step in data pre-processing is detecting outliers

Once detected, domain knowledge is required to determine if it is an error, or truly extreme.

Detecting Outliers

In some contexts, finding outliers is the purpose of the DM exercise (airport security screening). This is called “anomaly detection”.

Handling Missing Data

Most algorithms will not process records with missing values. Default is to drop those records.

Solution 1: Omission

- If a small number of records have missing values, can omit them
- If many records are missing values on a small set of variables, can drop those variables (or use proxies)
- If many records have missing values, omission is not practical

Solution 2: Imputation

- Replace missing values with reasonable substitutes
- Lets you keep the record and use the rest of its (non-missing) information

Replacing Missing Data with Median

To illustrate missing data procedures, we first convert a few entries for bedrooms to NA's. Then we impute these missing values using the median of the remaining values.

```
missingRows = housing_df.sample(10).index
housing_df.loc[missingRows, 'BEDROOMS'] = np.nan
print('Number of rows with valid BEDROOMS values
  after setting to NAN: ',
  housing_df['BEDROOMS'].count())

# remove rows with missing values
reduced_df = housing_df.dropna()
print('Number of rows after removing rows with
missing values: ', len(reduced_df))
```

Replacing Missing Data with Median, cont.l

To illustrate missing data procedures, we first convert a few entries for bedrooms to NA's. Then we impute these missing values using the median of the remaining values.

```
# replace the missing values using the median of
  the remaining values.
medianBedrooms = housing_df['BEDROOMS'].median()
housing_df.BEDROOMS =
  housing_df.BEDROOMS.fillna(value=medianBedrooms)
print('Number of rows with valid BEDROOMS values
after filling NA values: ',
  housing_df['BEDROOMS'].count())
```

Normalizing (Standardizing) Dat

- Used in some techniques when variables with the largest scales would dominate and skew results
- Puts all variables on same scale
- Normalizing function: Subtract mean and divide by standard deviation
- Alternative function: scale to 0-1 by subtracting minimum and dividing by the range
    - Useful when the data contain dummies and numeric

Code for Normalizing Data

```
from sklearn.preprocessing import MinMaxScaler,
StandardScaler
df = housing_df.copy()

# pandas:
norm_df = (housing_df - housing_df.mean()) /
    housing_df.std()

# scikit-learn:
scaler = StandardScaler()
norm_df =
pd.DataFrame(scaler.fit_transform(housing_df),
   index=housing_df.index,
   columns=housing_df.columns)
# the result of the transformation is a numpy
   array, we convert it into a dataframe
```

Code for Normalizing Data, cont.

```
# Rescaling a data frame

# pandas:
norm_df = (housing_df - housing_df.min()) /
    (housing_df.max() - housing_df.min())

# scikit-learn:
scaler = MinMaxScaler()
norm_df =
pd.DataFrame(scaler.fit_transform(housing_df),
    index=housing_df.index,
    columns=housing_df.columns)
```

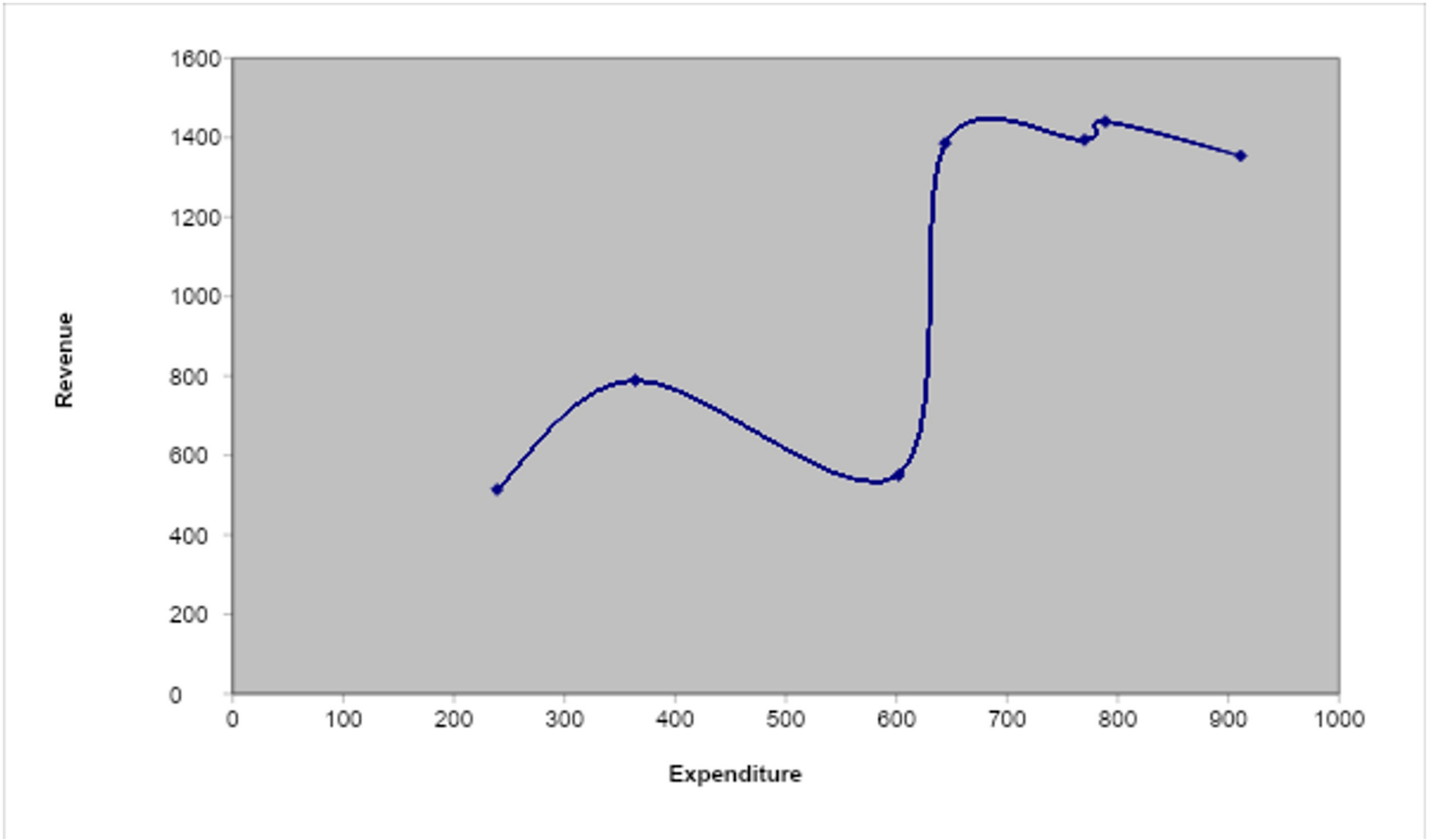
The Problem of Overfitting

Statistical models can produce highly complex explanations of relationships between variables

The “fit” may be excellent

When used with new data, models of great complexity do not do so well.

100% fit – not useful for new data



Overfitting (cont.)

Causes:

- Too many predictors
- A model with too many parameters
- Trying many different models

Consequence: Deployed model will not work as well as expected with completely new data.

Another example: Height of people on contributing to a charity

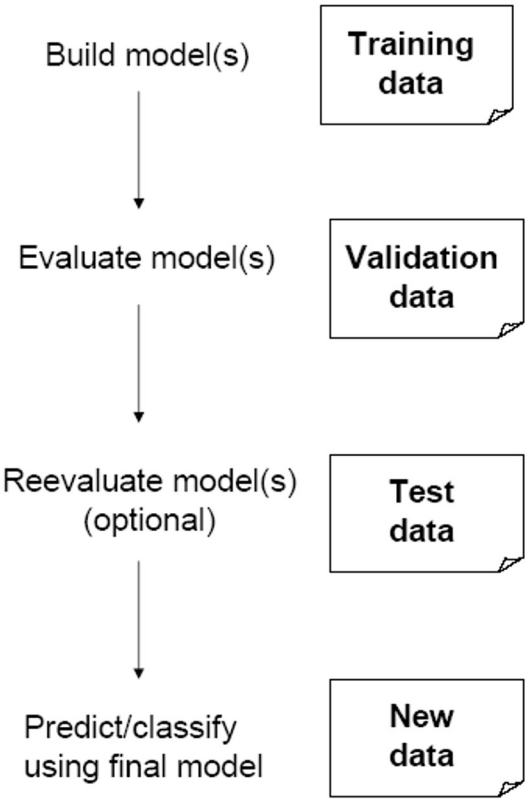
Partitioning the Data

Problem: How well will our model perform with new data?

Solution: Separate data into two parts

- Training partition to develop the model
- Validation partition to implement the model and evaluate its performance on "new" data

Addresses the issue of overfitting



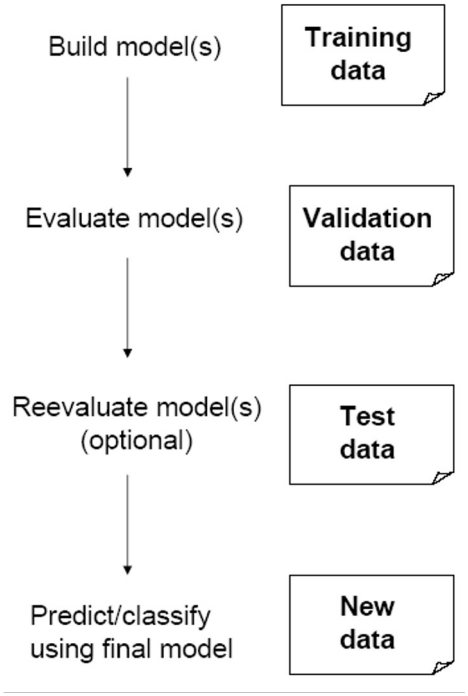
Test Partition

When a model is developed on **training data**, it can overfit the training data (hence need to assess on validation)

Assessing multiple models on same **validation data** can overfit validation data

Some methods use the validation data to choose a parameter. This too can lead to overfitting the validation data

Solution: final selected model is applied to a **test partition** to give unbiased estimate of its performance on new data



“Test” Partition - Terminology

- In *Data Mining for Business Analytics*, the “test” partition is the third partition for unbiased assessment.
- More generally in data science, and in Python syntax, “test” refers to the partition set aside from the training partition

Data Partitioning Approach

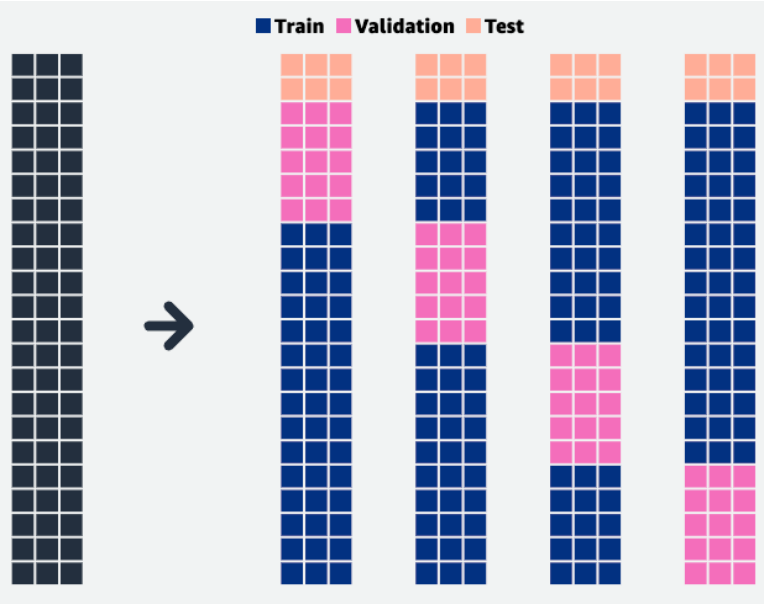
- Single validation set (train-test split/validation set approaches): One-time partitioning of the data into a training set and a validation set (or test set). Common with larger datasets.
- K-fold cross-validation: The dataset is divided into k equally-sized folds. The model is trained and evaluated k times, each time using a different fold as the validation set and the remaining k-1 folds as the training set. Performance metrics are averaged over the k iterations to assess model performance.
- Stratified sampling: Randomly divide the dataset into training and validation/test sets while maintaining the same class distribution as the original dataset.
- Stratified cross-validation: Similar to k-fold cross-validation, but it ensures that each fold maintains the same class distribution as the original dataset. This is especially useful for imbalanced datasets.
- Leave-one-out cross-validation: A special case of k-fold cross-validation where k is set equal to the number of data points. Each data point is used as the validation set once, and the rest are used for training. Very resource-intensive for large datasets but provides an accurate estimate of performance.
- Time-based split: For time series data, partition the data chronologically, with earlier data used for training and later data used for validation or testing.
- Bootstrapping: Create multiple training and test sets by randomly sampling the dataset with replacement. Useful for estimating model performance variability.
- Nested cross-validation: A combination of cross-validation and hyperparameter tuning. An outer loop uses k-fold cross-validation to assess model performance. An inner loop uses another round of k-fold cross-validation to select the best hyperparameters.
- Hold-out validation: Similar to train/validation/test split, but without the use of a validation set. Data is split into a training set and a test set, with no separate validation set. Used when the dataset is sufficiently large or when cross-validation is not practical.

Partitioning the Data

```
# set random_state for reproducibility


# training (60%) and validation (40%)
trainData, validData = train_test_split(housing_df,
   test_size=0.40, random_state=1)

# produces Training: 3481  Validation: 2321


# training (50%), validation (30%), and test (20%)
trainData, temp = train_test_split(housing_df,
test_size=0.5, random_state=1)
# now split temp into validation and test
validData, testData = train_test_split(temp,
test_size=0.4, random_state=1)

# produces  Training:  2901  Validation:  1741
   Test:  1160
```

K-Fold Cross Validation

- The single validation set (validation set approach) is still widely used, especially when resource constraints prohibit alternatives that require resampling (like cross-validation). But it is not perfect. The obvious issue is that the estimate of the test error can be highly variable depending on which particular observations are included in the training set and which are included in the validation set. That is, how do we know that the 30% we selected is the best way to split the data? What if we had used a different split instead? Another issue is that this approach tends to overestimate the test error for models fit on the entire dataset. This is because more training data usually means better accuracy, but the validation set approach reserves a decent-sized chunk of data for validation and testing (and not training).
- Cross-validation is a way to use more of the data for training while also simultaneously evaluating the performance across all the variance in the dataset. Rather than worrying about which split of data to use for training versus validation, all of them are used in turn.



K-Fold Cross Validation

- When the # of records in our sample is small
  (then data partitioning is not advisable as each partition will contain too few records for model building and performance evaluation)
- Repeated partitioning = cross-validation ("cv")
- k-fold cross validation, e.g. k=5 (often)
  - For each fold, set aside ⅕ of data as validation
  - Use full remainder as training
  - The validation folds are non-overlapping (subsamples)
- In Python
  - `cross_val_score()`
  - cross_val_score(a classifier, Data, Target, cv=5)
  - Argument `cv` determines the number of folds
  - More general `cross_validate` (retrieves more info)

Example – Linear Regression West Roxbury Housing Data

TABLE 2.1 DESCRIPTION OF VARIABLES IN WEST ROXBURY (BOSTON) HOME VALUE DATASET

TOTAL VALUE	Total assessed value for property, in thousands of USD
TAX	Tax bill amount based on total assessed value multiplied by the tax rate, in USD
LOT SQ FT	Total lot size of parcel in square feet
YR BUILT	Year the property was built
GROSS AREA	Gross floor area
LIVING AREA	Total living area for residential properties (ft^2)
FLOORS	Number of floors
ROOMS	Total number of rooms
BEDROOMS	Total number of bedrooms
FULL BATH	Total number of full baths
HALF BATH	Total number of half baths
KITCHEN	Total number of kitchens
FIREPLACE	Total number of fireplaces
REMODEL	When the house was remodeled (Recent/Old/None)

Data Preparation

```
# data loading and preprocessing
housing_df = pd.read_csv('WestRoxbury.csv')
housing_df.columns = [s.strip().replace(' ', '_')
   for s in housing_df.columns]
housing_df = pd.get_dummies(housing_df,
   prefix_sep='_', drop_first=True)

# create list of predictors and outcome
excludeColumns = ('TOTAL_VALUE', 'TAX')
predictors = [s for s in housing_df.columns if s
   not in excludeColumns]
outcome = 'TOTAL_VALUE'
```

"TAX" variable is excluded, why? Circularity, similar to the "TOTAL VALUE"

Data Partitioning

```
# partition data
X = housing_df[predictors]
y = housing_df[outcome]
train_X, valid_X, train_y, valid_y =
    train_test_split(X, y, test_size=0.4,
    random_state=1)
```

Fit Model and Make Predictions (Training Data)

```
model = LinearRegression()
model.fit(train_X, train_y)

train_pred = model.predict(train_X)
train_results = pd.DataFrame({
    'TOTAL_VALUE': train_y,
    'predicted': train_pred,
    'residual': train_y - train_pred
})

# show sample of predictions
train_results.head()
```

Sample Output

	TOTAL_VALUE	predicted	residual
2024	392.0	387.726258	4.273742
5140	476.3	430.785540	45.514460
5259	367.4	384.042952	-16.642952
421	350.3	369.005551	-18.705551
1401	348.1	314.725722	33.374278

Access the model`s parameters in LinearRegression:

- model.coef_
- model.intercept_

Scoring the validation data

```
valid_pred = model.predict(valid_X)
valid_results = pd.DataFrame({
    'TOTAL_VALUE': valid_y,
    'predicted': valid_pred,
    'residual': valid_y - valid_pred
})
valid_results.head()
```

	TOTAL_VALUE	predicted	residual
1822	462.0	406.946377	55.053623
1998	370.4	362.888928	7.511072
5126	407.4	390.287208	17.112792
808	316.1	382.470203	-66.370203
4034	393.2	434.334998	-41.134998

Assess Accuracy

```
# import the utility function regressionSummary
# from dmba import regressionSummary


# training set
regressionSummary(train_results.TOTAL_VALUE,
    train_results.predicted)

# validation set
regressionSummary(valid_results.TOTAL_VALUE,
valid_results.predicted)
```

Assess Accuracy - Output

Regression statistics (training)

```
                    Mean Error (ME) : -0.0000
         Root Mean Squared Error (RMSE) : 43.0306
              Mean Absolute Error (MAE) : 32.6042
            Mean Percentage Error (MPE) : -1.1116
Mean Absolute Percentage Error (MAPE) : 8.4886
```

Regression statistics (validation)

```
                    Mean Error (ME) : -0.1463
         Root Mean Squared Error (RMSE) : 42.7292
              Mean Absolute Error (MAE) : 31.9663
            Mean Percentage Error (MPE) : -1.0884
Mean Absolute Percentage Error (MAPE) : 8.3283
```

Error metrics

Error = actual – predicted

ME = Low (Unbiased prediction)

RMSE = Error magnitude (Sensitive to outliers)

MAE = Average magnitude or size of errors made by the model (robust to outliers)

MPE = Tendency to overestimate (positive values) or underestimate (negative values)

MAPE = Absolute tendency to overestimate (positive values) or underestimate (negative values)