# MS4S10 - Machine Learning and Decision Making

## Supplementary Resources

Moizzah Asif, J418, moizzah.asift@southwales.ac.uk

University of South Wales

# Contents


**1 Week 1** **4**
- 1.1 Setting up Python . . . . . . . . . . . . . . . . . . . . . . . . 4
  - 1.1.1 Python Resources . . . . . . . . . . . . . . . . . . . . . . 4
  - 1.1.2 Manual Installation . . . . . . . . . . . . . . . . . . . . 4
  - 1.1.3 Anaconda Installation . . . . . . . . . . . . . . . . . . . 6
- 1.2 Python packages to be used in lecture . . . . . . . . . . . . . . 7
- 1.3 Dataset to be used in lecture . . . . . . . . . . . . . . . . . . . 8
- 1.4 Misc. concepts required . . . . . . . . . . . . . . . . . . . . . . 8
  - 1.4.1 Distance metrics . . . . . . . . . . . . . . . . . . . . . . 8
  - 1.4.2 Variance . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  - 1.4.3 Degenerate distribution . . . . . . . . . . . . . . . . . . 11
- 1.5 Examples for experimental work . . . . . . . . . . . . . . . . . . 12
  - 1.5.1 Categorical Variables . . . . . . . . . . . . . . . . . . . . 12
  - 1.5.2 Binning and Linear models . . . . . . . . . . . . . . . . . 19
- 1.6 Post Lecture reading . . . . . . . . . . . . . . . . . . . . . . . 23
  - 1.6.1 Box-Cox transformations . . . . . . . . . . . . . . . . . . 23
  - 1.6.2 PCA (Principal Component Analysis) . . . . . . . . . . . 23

**2 Week 2** **24**
- 2.1 Some Discrete Mathematics . . . . . . . . . . . . . . . . . . . . . 24
  - 2.1.1 Logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
  - 2.1.2 Propositional logic . . . . . . . . . . . . . . . . . . . . . 24
  - 2.1.3 Composite statements . . . . . . . . . . . . . . . . . . . 24
  - 2.1.4 Negation . . . . . . . . . . . . . . . . . . . . . . . . . . 25
  - 2.1.5 Conjunction . . . . . . . . . . . . . . . . . . . . . . . . . 25
  - 2.1.6 Disjunction . . . . . . . . . . . . . . . . . . . . . . . . . 25
- 2.2 Greedy Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . 26
  - 2.2.1 Is greedy optimal? . . . . . . . . . . . . . . . . . . . . . . 26
- 2.3 Decision Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
  - 2.3.1 ID3 Pseudo code . . . . . . . . . . . . . . . . . . . . . . 27
  - 2.3.2 Creating a decision tree . . . . . . . . . . . . . . . . . . 28
  - 2.3.3 Gini Index . . . . . . . . . . . . . . . . . . . . . . . . . 31
- 2.4 Error Metrics for Regression . . . . . . . . . . . . . . . . . . . . 32

2.4.1 Mean Squared Error MSE . . . . . . . . . . . . . . . . . 33
2.4.2 Root Mean Squared Error . . . . . . . . . . . . . . . . . 33
2.4.3 Mean Absolute Error . . . . . . . . . . . . . . . . . . . 33
2.5 Suggested further reading . . . . . . . . . . . . . . . . . . . . . 33

**3 Week 3** **34**
3.1 Random variable distributions . . . . . . . . . . . . . . . . . . . 34
3.1.1 Bernoulli and binomial distributions . . . . . . . . . . . 35
3.1.2 Multinomial Distribution . . . . . . . . . . . . . . . . . . 45

# Chapter 1

# Week 1

## 1.1 Setting up Python

### 1.1.1 Python Resources

There are three main Python releases: Python 1,2 and 3. We will use **Python 3** as all new features of Python are integrated in this line of release. You may find following link useful `docs.python.org/3/`.

### 1.1.2 Manual Installation

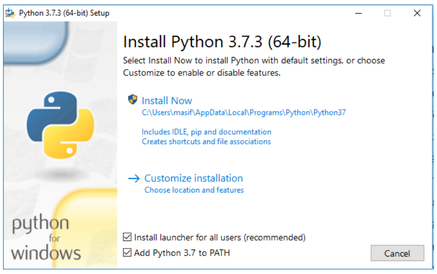



Figure 1.1: Python's default installation window

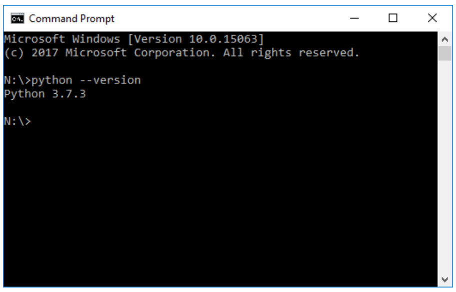



Figure 1.2: Checking python's version on your computer

1. Download the latest stable version (3.7.3 for now), from this link: `https://www.python.org/downloads/windows/`

2. Find a version that suits your computer and operating system, we are assuming 64 bit windows operating system.

3. Run the executable file, and select "install now" as shown in Fig 1.1.

4. Make sure you have checked the second box '**Add Python 3.7 to PATH**' as shown in Fig 1.1

5. Good job! you have just installed Python. Now we need to confirm with our computer that it recognises the newly installed version.

6. Type "cmd" in the search bar, to run command line.

7. Type the following command to check python version your computer is running: *python –version*

8. Your computer's response should look similar to the one shown in Fig 1.2

**setting up IDE**

An IDE is acronym for Integrated Development Environment: a platform to code and execute python's programming scripts. We can run our scripts on the environment which came with Python installation, but lets just not be Old School! Lets learn how to setup Jupyter Notebook to run our python scripts.

1. Run cmd (command line) as administrator.

2. Type the following command to ensure that PIP (a package management and software installation system) is running its late version, *python -m pip install –upgrade pip*

3. Run the following command to install Jupyter notebook, *python -m pip install jupyter*

4. Type the following command on cmd to launch and ensure that Jupyter notebook is installed, *jupyter notebook*

5. Jupyter notebooks open in computer's default browser, and look like as shown in Fig 1.3

6. If you would like to change the working directory, and avoid saving scripts in default folder, type the following command on a new cmd

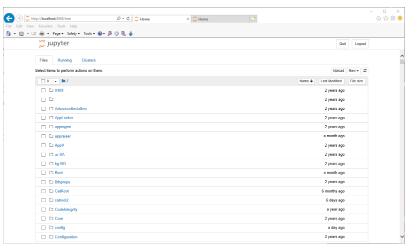


Figure 1.3: Jupyter Notebook in browser

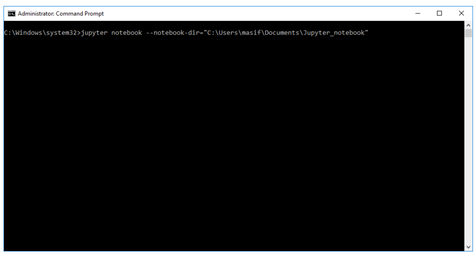



Figure 1.4: Changing Jupyter notebook directory

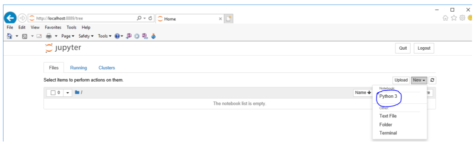


Figure 1.5: Creating new python script

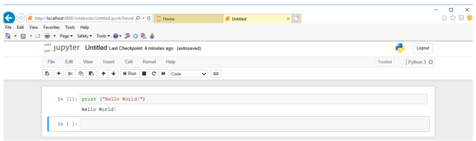


Figure 1.6: First python script: hello world

window. It will set the working directory in your desired folder and all the scripts will be saved here from now on. Please modify and add the folder names as required.
*jupyter notebook --notebook-dir=\ folder1\ folder2*

#### executing first python script in Jupyter Notebook

This is the easiest bit!

1. Open Jupyter notebooks from cmd following the instruction given in 1.1.2.

2. Select new python3 script as shown in Fig 1.5.

3. Type the following piece of code and press RUN from the utility icons bar:
   *print ("Hello World!")*

4. Your output should like the one in Fig 1.6.

### 1.1.3 Anaconda Installation

Anaconda installs IDEs and several important packages like NumPy, Pandas, and so on, and this is a really convenient package which can be downloaded and installed.

1. Download the appropriate anaconda installer from `https://www.anaconda.com/distribution/`

2. Go through the installation procedure with the installer.

3. After the installation is complete, search for Anaconda Navigator in the Start menu. The navigator homepage looks similar to as shown in 1.7

#### Executing first python script with Jupyter Notebook

1. Simply click the Jupyter notebook icon from navigator to open Jupyter notebook.

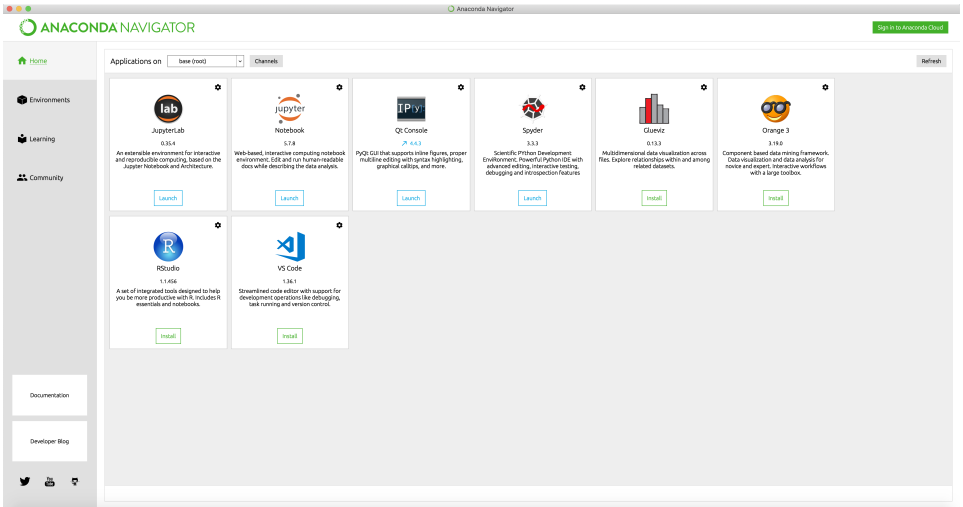



Figure 1.7: Anaconda navigator home

2. follow steps 2 - 4 from 1.1.2 for executing first python script from Jupyter notebook

### Note

Anaconda installation will download all available packages and libraries whereas with manual installation any required libraries and packages would need to be installed manually from command line using pip installer. please refer to `https://packaging.python.org/tutorials/installing-packages/` for further information on how to use pip installer.

Please refer to the follwing url for further reading and developing an understanding of Jupyter notebooks before the lecture,

## 1.2 Python packages to be used in lecture

1. pandas `https://pandas.pydata.org/`, `https://bit.ly/2Gjup5s`
2. matplotlib `https://matplotlib.org/index.html`
3. numpy
4. SciPy `https://www.scipy.org/`
5. Scikitlearn `https://scikit-learn.org/stable/`

## 1.3 Dataset to be used in lecture

The dataset to be used in today's lectures is originally available at `http://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html`. This dataset appeared in a 1997 paper titled "Sparse Spatial Autoregressions" by Pace, R. Kelley and Ronald Barry, published in the Statistics and Probability Letters journal. They built it using the 1990 California census data. It contains one row per census block group. A block group is the smallest geographical unit for which the U.S. Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people).

We are adapting the slightly tweaked version of this dataset available at `https://github.com/ageron/handson-ml/tree/master/datasets/housing`. The author of the book "Hands on machine learning with scikit learn and tensorflow" has made tweaks as described in the previously provided url for learning purposes.

## 1.4 Misc. concepts required

### 1.4.1 Distance metrics

#### Numerical values

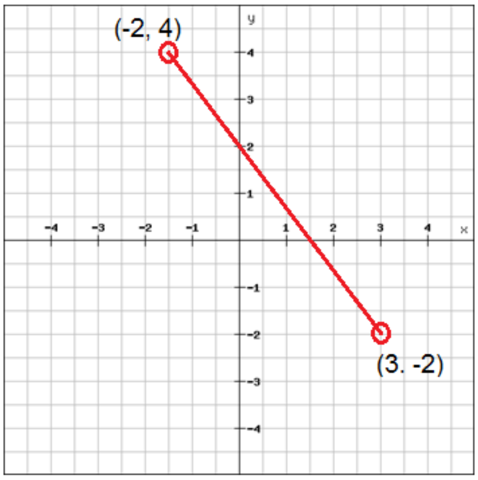



Figure 1.8: line segment PQ, p(-2,4) and q(3,-2)

1. **Euclidean Distance**

   The Euclidean distance between points **p** and **q** is the length of the line segment connecting them, $\overline{\mathbf{pq}}$.

So, if **p** and **q** lie on a two-dimensional Cartesian plane (axis - x and y), and **p** is at points ($x_1$,$y_1$) and **q** is at points ($x_2$, $y_2$), then distance $d_E(p,q)$ can be calculated as follows:

$$d_E(p,q) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

***Example***: using the plot from fig 1.8

$$d_E(p,q) = \sqrt{(3 - (-2))^2 + (-2 - 4)^2}$$
$$d_E(p,q) = 7.81$$

2. **Manhattan Distance** The sum of the lengths of the projections of the line segment between the points onto the coordinate axes. More formally

$$d_M(p,q) = \sum_{i=1}^{n} |p_i - q_i|$$

So, if **p** and **q** lie on a two-dimensional Cartesian plane (axis - x and y), and **p** is at points ($x_1$,$y_1$) and **q** is at points ($x_2$, $y_2$),then distance $d(p,q)$ can be calculated as follows:

$$d_M(p,q) = |x^2 - x^1| + |y^2 - y^1|$$

***Example***: using the plot from fig 1.8

$$d_M(p,q) = |3 - (-2)| + | - 2 - 4|$$
$$d_M(p,q) = 11$$

### Categorical values

1. **Hamming Distance**

This distance is computed by overlaying one string over another and finding the places where the strings vary.
***Example***: consider the two words *'text'* and *'test'*. If you overlay them on each other as follows:

$$text$$
$$test$$

let's underline the places where the strings vary:

$$te\underline{x}t$$
$$te\underline{s}t$$
$$\text{so } d_H(test, text) = 1$$

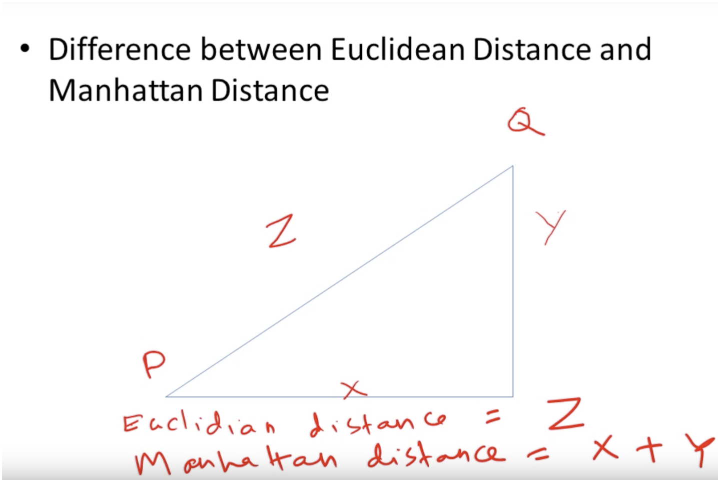



Figure 1.9: Euclidean distance vs Manhattan Distance

Let's take another example:

$$arrow$$
$$arow$$

let's underline the places where the strings vary:

$$ar\underline{row}$$
$$ar\underline{ow\ }$$
so $d_H(arrow, arow) = 3$

note: the blank space is underlined after 'w' in *arow*

2. **Levenshtein Distance**

   This distance is computed by finding the number of edits which will transform one string to another. The transformations allowed are insertion — adding a new character, deletion — deleting a character and substitution — replace one character by another.

   For the *'text'* and *'test'* example, only one substitution/replacement needs to be done, i.e. replacing 's' with 'x'. So, $d_L(test, text) = 1$

   Similarly, for the *'arrow'* and *'arow'* example, only one insertion needs to be done, i.e. adding another 'r' after the first 'r' in *arow*. So, $d_L(arrow, arow) = 1$

### 1.4.2 Variance

If you are not sure what variance is then, please read through the following page to develop an understanding of the statistical measure called Variance.

https://bit.ly/2YSVdTB

### 1.4.3 Degenerate distribution

If you do not have a basic understanding of what degenerate distribution of a random variable refers to, you may as well find this introduction in the following URL useful for the lecture to follow. https://bit.ly/2TgBI68

## 1.5 Examples for experimental work

### 1.5.1 Categorical Variables

Excerpt from Chapter 4 of Introduction to Machine Learning with Python by Andreas C.Muller and Sarah Guido

learning practitioners trying to solve real-world problems. Representing your data in the right way can have a bigger influence on the performance of a supervised model than the exact parameters you choose.

In this chapter, we will first go over the important and very common case of categorical features, and then give some examples of helpful transformations for specific combinations of features and models.

# Categorical Variables

As an example, we will use the dataset of adult incomes in the United States, derived from the 1994 census database. The task of the `adult` dataset is to predict whether a worker has an income of over $50,000 or under $50,000. The features in this dataset include the workers' ages, how they are employed (self employed, private industry employee, government employee, etc.), their education, their gender, their working hours per week, occupation, and more. Table 4-1 shows the first few entries in the dataset.

*Table 4-1. The first few entries in the adult dataset*

| | age | workclass | education | gender | hours-per-week | occupation | income |
|---|---|---|---|---|---|---|---|
| 0 | 39 | State-gov | Bachelors | Male | 40 | Adm-clerical | <=50K |
| 1 | 50 | Self-emp-not-inc | Bachelors | Male | 13 | Exec-managerial | <=50K |
| 2 | 38 | Private | HS-grad | Male | 40 | Handlers-cleaners | <=50K |
| 3 | 53 | Private | 11th | Male | 40 | Handlers-cleaners | <=50K |
| 4 | 28 | Private | Bachelors | Female | 40 | Prof-specialty | <=50K |
| 5 | 37 | Private | Masters | Female | 40 | Exec-managerial | <=50K |
| 6 | 49 | Private | 9th | Female | 16 | Other-service | <=50K |
| 7 | 52 | Self-emp-not-inc | HS-grad | Male | 45 | Exec-managerial | >50K |
| 8 | 31 | Private | Masters | Female | 50 | Prof-specialty | >50K |
| 9 | 42 | Private | Bachelors | Male | 40 | Exec-managerial | >50K |
| 10 | 37 | Private | Some-college | Male | 80 | Exec-managerial | >50K |

The task is phrased as a classification task with the two classes being income `<=50k` and `>50k`. It would also be possible to predict the exact income, and make this a regression task. However, that would be much more difficult, and the 50K division is interesting to understand on its own.

In this dataset, `age` and `hours-per-week` are continuous features, which we know how to treat. The `workclass`, `education`, `sex`, and `occupation` features are categorical, however. All of them come from a fixed list of possible values, as opposed to a range, and denote a qualitative property, as opposed to a quantity.

212 | **Chapter 4: Representing Data and Engineering Features**

Figure 1.10: categorical variable handout 1

As a starting point, let's say we want to learn a logistic regression classifier on this data. We know from Chapter 2 that a logistic regression makes predictions, $\hat{y}$, using the following formula:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + ... + w[p] * x[p] + b > 0$$

where $w[i]$ and $b$ are coefficients learned from the training set and $x[i]$ are the input features. This formula makes sense when $x[i]$ are numbers, but not when $x[2]$ is `"Masters"` or `"Bachelors"`. Clearly we need to represent our data in some different way when applying logistic regression. The next section will explain how we can overcome this problem.

## One-Hot-Encoding (Dummy Variables)

By far the most common way to represent categorical variables is using the *one-hot-encoding* or *one-out-of-N encoding*, also known as *dummy variables*. The idea behind dummy variables is to replace a categorical variable with one or more new features that can have the values 0 and 1. The values 0 and 1 make sense in the formula for linear binary classification (and for all other models in `scikit-learn`), and we can represent any number of categories by introducing one new feature per category, as described here.

Let's say for the `workclass` feature we have possible values of `"Government Employee"`, `"Private Employee"`, `"Self Employed"`, and `"Self Employed Incorporated"`. To encode these four possible values, we create four new features, called `"Government Employee"`, `"Private Employee"`, `"Self Employed"`, and `"Self Employed Incorporated"`. A feature is 1 if `workclass` for this person has the corresponding value and 0 otherwise, so exactly one of the four new features will be 1 for each data point. This is why this is called one-hot or one-out-of-N encoding.

The principle is illustrated in Table 4-2. A single feature is encoded using four new features. When using this data in a machine learning algorithm, we would drop the original `workclass` feature and only keep the 0–1 features.

*Table 4-2. Encoding the workclass feature using one-hot encoding*

| workclass | Government Employee | Private Employee | Self Employed | Self Employed Incorporated |
|---|---|---|---|---|
| Government Employee | 1 | 0 | 0 | 0 |
| Private Employee | 0 | 1 | 0 | 0 |
| Self Employed | 0 | 0 | 1 | 0 |
| Self Employed Incorporated | 0 | 0 | 0 | 1 |



Figure 1.11: categorical variable handout 2

The one-hot encoding we use is quite similar, but not identical, to the dummy encoding used in statistics. For simplicity, we encode each category with a different binary feature. In statistics, it is common to encode a categorical feature with *k* different possible values into *k*–1 features (the last one is represented as all zeros). This is done to simplify the analysis (more technically, this will avoid making the data matrix rank-deficient).

There are two ways to convert your data to a one-hot encoding of categorical variables, using either `pandas` or `scikit-learn`. At the time of writing, using `pandas` is slightly easier, so let's go this route. First we load the data using `pandas` from a comma-separated values (CSV) file:

**In[2]:**

```
import pandas as pd
# The file has no headers naming the columns, so we pass header=None
# and provide the column names explicitly in "names"
data = pd.read_csv(
    "/home/andy/datasets/adult.data", header=None, index_col=False,
    names=['age', 'workclass', 'fnlwgt', 'education',  'education-num',
           'marital-status', 'occupation', 'relationship', 'race', 'gender',
           'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',
           'income'])
# For illustration purposes, we only select some of the columns
data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week',
             'occupation', 'income']]
# IPython.display allows nice output formatting within the Jupyter notebook
display(data.head())
```

Table 4-3 shows the result.

*Table 4-3. The first five rows of the adult dataset*

| | age | workclass | education | gender | hours-per-week | occupation | income |
|---|---|---|---|---|---|---|---|
| 0 | 39 | State-gov | Bachelors | Male | 40 | Adm-clerical | <=50K |
| 1 | 50 | Self-emp-not-inc | Bachelors | Male | 13 | Exec-managerial | <=50K |
| 2 | 38 | Private | HS-grad | Male | 40 | Handlers-cleaners | <=50K |
| 3 | 53 | Private | 11th | Male | 40 | Handlers-cleaners | <=50K |
| 4 | 28 | Private | Bachelors | Female | 40 | Prof-specialty | <=50K |

#### Checking string-encoded categorical data

After reading a dataset like this, it is often good to first check if a column actually contains meaningful categorical data. When working with data that was input by humans (say, users on a website), there might not be a fixed set of categories, and differences in spelling and capitalization might require preprocessing. For example, it might be that some people specified gender as "male" and some as "man," and we



Figure 1.12: categorical variable handout 3

might want to represent these two inputs using the same category. A good way to check the contents of a column is using the `value_counts` function of a `pandas` `Series` (the type of a single column in a `DataFrame`), to show us what the unique values are and how often they appear:

**In[3]:**

```
print(data.gender.value_counts())
```

**Out[3]:**

```
 Male      21790
 Female    10771
Name: gender, dtype: int64
```

We can see that there are exactly two values for gender in this dataset, `Male` and `Female`, meaning the data is already in a good format to be represented using one-hot-encoding. In a real application, you should look at all columns and check their values. We will skip this here for brevity's sake.

There is a very simple way to encode the data in `pandas`, using the `get_dummies` function. The `get_dummies` function automatically transforms all columns that have object type (like strings) or are categorical (which is a special `pandas` concept that we haven't talked about yet):

**In[4]:**

```
print("Original features:\n", list(data.columns), "\n")
data_dummies = pd.get_dummies(data)
print("Features after get_dummies:\n", list(data_dummies.columns))
```

**Out[4]:**

```
Original features:
 ['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation',
  'income']

Features after get_dummies:
 ['age', 'hours-per-week', 'workclass_ ?', 'workclass_ Federal-gov',
  'workclass_ Local-gov', 'workclass_ Never-worked', 'workclass_ Private',
  'workclass_ Self-emp-inc', 'workclass_ Self-emp-not-inc',
  'workclass_ State-gov', 'workclass_ Without-pay', 'education_ 10th',
  'education_ 11th', 'education_ 12th', 'education_ 1st-4th',
   ...
  'education_ Preschool', 'education_ Prof-school', 'education_ Some-college',
  'gender_ Female', 'gender_ Male', 'occupation_ ?',
  'occupation_ Adm-clerical', 'occupation_ Armed-Forces',
  'occupation_ Craft-repair', 'occupation_ Exec-managerial',
  'occupation_ Farming-fishing', 'occupation_ Handlers-cleaners',
  ...
  'occupation_ Tech-support', 'occupation_ Transport-moving',
  'income_ <=50K', 'income_ >50K']
```



Figure 1.13: categorical variable handout 4

You can see that the continuous features age and hours-per-week were not touched, while the categorical features were expanded into one new feature for each possible value:

**In[5]:**

```
data_dummies.head()
```

**Out[5]:**

| | age | hours-per-week | workclass_ ? | workclass_ Federal-gov | workclass_ Local-gov | ... | occupation_ Tech-support | occupation_ Transport-moving | income_ <=50K | income_ >50K |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 39 | 40 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |
| 1 | 50 | 13 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |
| 2 | 38 | 40 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |
| 3 | 53 | 40 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |
| 4 | 28 | 40 | 0.0 | 0.0 | 0.0 | ... | 0.0 | 0.0 | 1.0 | 0.0 |

```
5 rows × 46 columns
```

We can now use the values attribute to convert the data_dummies DataFrame into a NumPy array, and then train a machine learning model on it. Be careful to separate the target variable (which is now encoded in two income columns) from the data before training a model. Including the output variable, or some derived property of the output variable, into the feature representation is a very common mistake in building supervised machine learning models.

Be careful: column indexing in pandas includes the end of the range, so 'age':'occupation_ Transport-moving' is inclusive of occupation_ Transport-moving. This is different from slicing a NumPy array, where the end of a range is not included: for example, np.arange(11)[0:10] doesn't include the entry with index 10.

In this case, we extract only the columns containing features—that is, all columns from age to occupation_ Transport-moving. This range contains all the features but not the target:

**In[6]:**

```
features = data_dummies.ix[:, 'age':'occupation_ Transport-moving']
# Extract NumPy arrays
X = features.values
y = data_dummies['income_ >50K'].values
print("X.shape: {}  y.shape: {}".format(X.shape, y.shape))
```

216 | **Chapter 4: Representing Data and Engineering Features**

Figure 1.14: categorical variable handout 5

**Out[6]:**

```
X.shape: (32561, 44)  y.shape: (32561,)
```

Now the data is represented in a way that `scikit-learn` can work with, and we can proceed as usual:

**In[7]:**

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Test score: {:.2f}".format(logreg.score(X_test, y_test)))
```

**Out[7]:**

```
Test score: 0.81
```

In this example, we called `get_dummies` on a `DataFrame` containing both the training and the test data. This is important to ensure categorical values are represented in the same way in the training set and the test set.

Imagine we have the training and test sets in two different `DataFrames`. If the `"Private Employee"` value for the `workclass` feature does not appear in the test set, `pandas` will assume there are only three possible values for this feature and will create only three new dummy features. Now our training and test sets have different numbers of features, and we can't apply the model we learned on the training set to the test set anymore. Even worse, imagine the `workclass` feature has the values `"Government Employee"` and `"Private Employee"` in the training set, and `"Self Employed"` and `"Self Employed Incorporated"` in the test set. In both cases, `pandas` will create two new dummy features, so the encoded `DataFrames` will have the same number of features. However, the two dummy features have entirely different meanings in the training and test sets. The column that means `"Government Employee"` for the training set would encode `"Self Employed"` for the test set.

If we built a machine learning model on this data it would work very badly, because it would assume the columns mean the same things (because they are in the same position) when in fact they mean very different things. To fix this, either call `get_dummies` on a `DataFrame` that contains both the training and the test data points, or make sure that the column names are the same for the training and test sets after calling `get_dummies`, to ensure they have the same semantics.

Categorical Variables | 217

Figure 1.15: categorical variable handout 6

## Numbers Can Encode Categoricals

In the example of the `adult` dataset, the categorical variables were encoded as strings. On the one hand, that opens up the possibility of spelling errors, but on the other hand, it clearly marks a variable as categorical. Often, whether for ease of storage or because of the way the data is collected, categorical variables are encoded as integers. For example, imagine the census data in the `adult` dataset was collected using a questionnaire, and the answers for `workclass` were recorded as 0 (first box ticked), 1 (second box ticked), 2 (third box ticked), and so on. Now the column will contain numbers from 0 to 8, instead of strings like `"Private"`, and it won't be immediately obvious to someone looking at the table representing the dataset whether they should treat this variable as continuous or categorical. Knowing that the numbers indicate employment status, however, it is clear that these are very distinct states and should not be modeled by a single continuous variable.

> Categorical features are often encoded using integers. That they are numbers doesn't mean that they should necessarily be treated as continuous features. It is not always clear whether an integer feature should be treated as continuous or discrete (and one-hot-encoded). If there is no ordering between the semantics that are encoded (like in the `workclass` example), the feature must be treated as discrete. For other cases, like five-star ratings, the better encoding depends on the particular task and data and which machine learning algorithm is used.

The `get_dummies` function in `pandas` treats all numbers as continuous and will not create dummy variables for them. To get around this, you can either use `scikit-learn`'s `OneHotEncoder`, for which you can specify which variables are continuous and which are discrete, or convert numeric columns in the `DataFrame` to strings. To illustrate, let's create a `DataFrame` object with two columns, one containing strings and one containing integers:

**In[8]:**

```
# create a DataFrame with an integer feature and a categorical string feature
demo_df = pd.DataFrame({'Integer Feature': [0, 1, 2, 1],
                        'Categorical Feature': ['socks', 'fox', 'socks', 'box']})
display(demo_df)
```

Table 4-4 shows the result.



Figure 1.16: categorical variable handout 7

*Table 4-4. DataFrame containing categorical string features and integer features*

| | Categorical Feature | Integer Feature |
|---|---|---|
| 0 | socks | 0 |
| 1 | fox | 1 |
| 2 | socks | 2 |
| 3 | box | 1 |

Using `get_dummies` will only encode the string feature and will not change the integer feature, as you can see in Table 4-5:

**In[9]:**

```
pd.get_dummies(demo_df)
```

*Table 4-5. One-hot-encoded version of the data from Table 4-4, leaving the integer feature unchanged*

| | Integer Feature | Categorical Feature_box | Categorical Feature_fox | Categorical Feature_socks |
|---|---|---|---|---|
| 0 | 0 | 0.0 | 0.0 | 1.0 |
| 1 | 1 | 0.0 | 1.0 | 0.0 |
| 2 | 2 | 0.0 | 0.0 | 1.0 |
| 3 | 1 | 1.0 | 0.0 | 0.0 |

If you want dummy variables to be created for the "Integer Feature" column, you can explicitly list the columns you want to encode using the `columns` parameter. Then, both features will be treated as categorical (see Table 4-6):

**In[10]:**

```
demo_df['Integer Feature'] = demo_df['Integer Feature'].astype(str)
pd.get_dummies(demo_df, columns=['Integer Feature', 'Categorical Feature'])
```

*Table 4-6. One-hot encoding of the data shown in Table 4-4, encoding the integer and string features*

| | Integer Feature_0 | Integer Feature_1 | Integer Feature_2 | Categorical Feature_box | Categorical Feature_fox | Categorical Feature_socks |
|---|---|---|---|---|---|---|
| 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| 1 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 2 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 |
| 3 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 |

Categorical Variables | 219

Figure 1.17: categorical variable handout 8

### 1.5.2 Binning and Linear models

Excerpt from Chapter 4 of Introduction to Machine Learning with Python by Andreas C.Muller and Sarah Guido

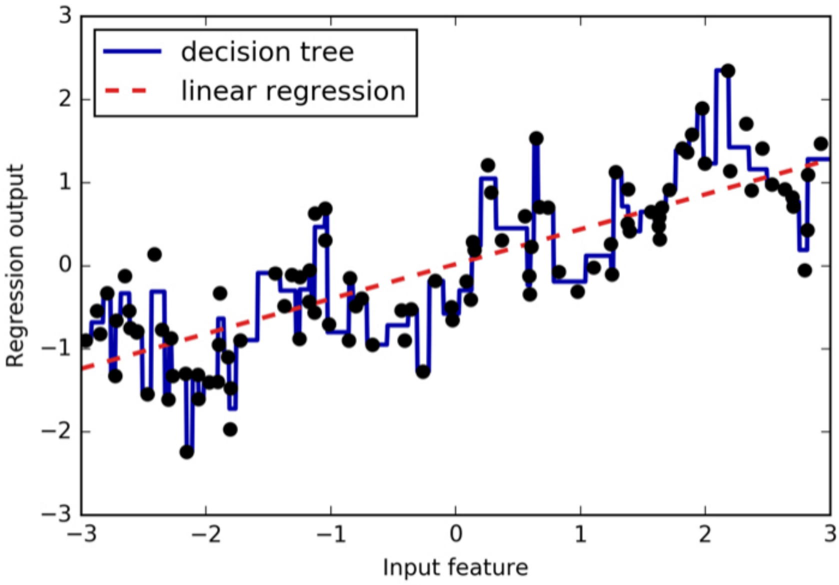



*Figure 4-1. Comparing linear regression and a decision tree on the wave dataset*

We imagine a partition of the input range for the feature (in this case, the numbers from –3 to 3) into a fixed number of *bins*—say, 10. A data point will then be represented by which bin it falls into. To determine this, we first have to define the bins. In this case, we'll define 10 bins equally spaced between –3 and 3. We use the `np.linspace` function for this, creating 11 entries, which will create 10 bins—they are the spaces in between two consecutive boundaries:

**In[12]:**

```
bins = np.linspace(-3, 3, 11)
print("bins: {}".format(bins))
```

**Out[12]:**

```
bins: [-3.  -2.4 -1.8 -1.2 -0.6  0.   0.6  1.2  1.8  2.4  3. ]
```

Here, the first bin contains all data points with feature values –3 to –2.68, the second bin contains all points with feature values from –2.68 to –2.37, and so on.

Next, we record for each data point which bin it falls into. This can be easily computed using the `np.digitize` function:



Figure 1.18: binning handout 1

**In[13]:**

```
which_bin = np.digitize(X, bins=bins)
print("\nData points:\n", X[:5])
print("\nBin membership for data points:\n", which_bin[:5])
```

**Out[13]:**

```
Data points:
 [[-0.753]
  [ 2.704]
  [ 1.392]
  [ 0.592]
 [-2.064]]

Bin membership for data points:
 [[ 4]
  [10]
  [ 8]
  [ 6]
 [ 2]]
```

What we did here is transform the single continuous input feature in the `wave` dataset into a categorical feature that encodes which bin a data point is in. To use a `scikit-learn` model on this data, we transform this discrete feature to a one-hot encoding using the `OneHotEncoder` from the `preprocessing` module. The `OneHotEncoder` does the same encoding as `pandas.get_dummies`, though it currently only works on categorical variables that are integers:

**In[14]:**

```
from sklearn.preprocessing import OneHotEncoder
# transform using the OneHotEncoder
encoder = OneHotEncoder(sparse=False)
# encoder.fit finds the unique values that appear in which_bin
encoder.fit(which_bin)
# transform creates the one-hot encoding
X_binned = encoder.transform(which_bin)
print(X_binned[:5])
```

**Out[14]:**

```
[[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

Because we specified 10 bins, the transformed dataset `X_binned` now is made up of 10 features:



Figure 1.19: binning handout 2

**In[15]:**

```
print("X_binned.shape: {}".format(X_binned.shape))
```

**Out[15]:**

```
X_binned.shape: (100, 10)
```

Now we build a new linear regression model and a new decision tree model on the one-hot-encoded data. The result is visualized in Figure 4-2, together with the bin boundaries, shown as dotted black lines:

**In[16]:**

```
line_binned = encoder.transform(np.digitize(line, bins=bins))

reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='linear regression binned')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='decision tree binned')
plt.plot(X[:, 0], y, 'o', c='k')
plt.vlines(bins, -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")
plt.ylabel("Regression output")
plt.xlabel("Input feature")
```

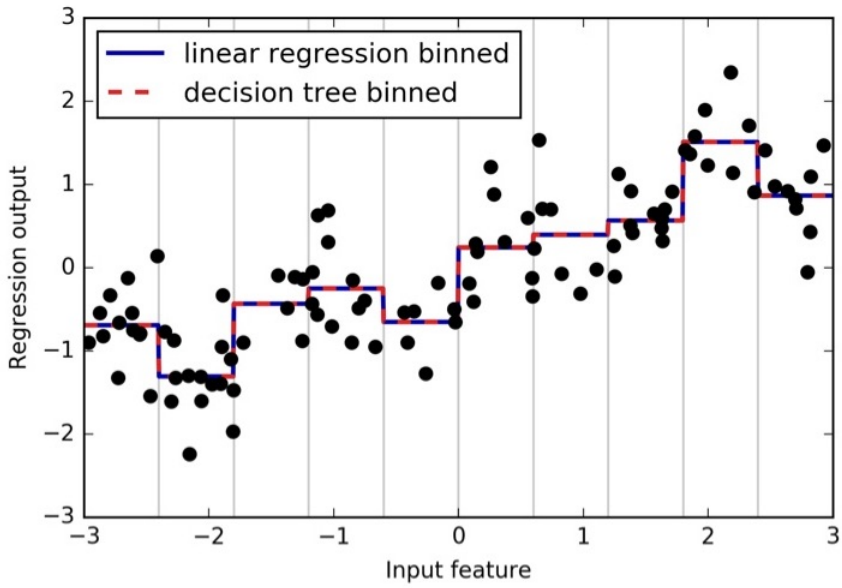
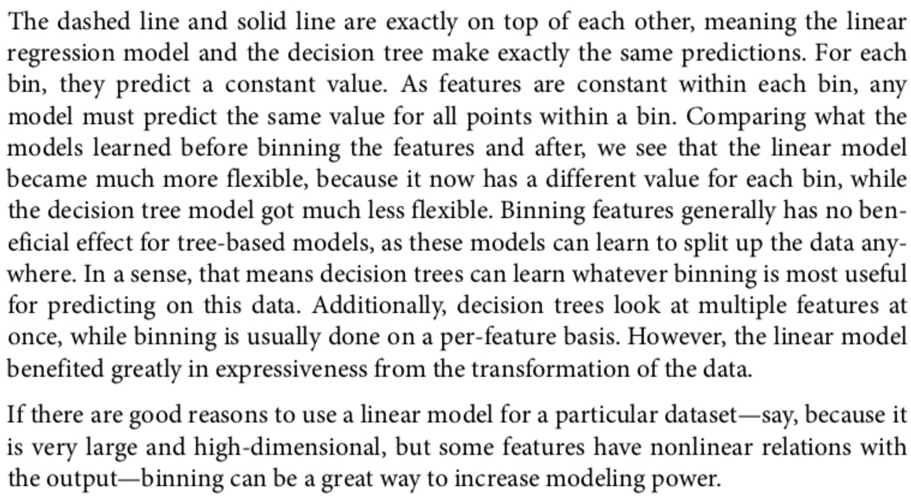



*Figure 4-2. Comparing linear regression and decision tree regression on binned features*



Figure 1.20: binning handout 3



The dashed line and solid line are exactly on top of each other, meaning the linear regression model and the decision tree make exactly the same predictions. For each bin, they predict a constant value. As features are constant within each bin, any model must predict the same value for all points within a bin. Comparing what the models learned before binning the features and after, we see that the linear model became much more flexible, because it now has a different value for each bin, while the decision tree model got much less flexible. Binning features generally has no beneficial effect for tree-based models, as these models can learn to split up the data anywhere. In a sense, that means decision trees can learn whatever binning is most useful for predicting on this data. Additionally, decision trees look at multiple features at once, while binning is usually done on a per-feature basis. However, the linear model benefited greatly in expressiveness from the transformation of the data.

If there are good reasons to use a linear model for a particular dataset—say, because it is very large and high-dimensional, but some features have nonlinear relations with the output—binning can be a great way to increase modeling power.

Figure 1.21: binning handout 4

## 1.6 Post Lecture reading

### 1.6.1 Box-Cox transformations

Here is a link to further reading on Box-Cox transformations if you would like to know a bit more about them mathematically. `https://bit.ly/289Hatf`

### 1.6.2 PCA (Principal Component Analysis)

If you want to read further and strengthen your concepts on PCA, the following URL is suggested as the starting point: `https://bit.ly/2nd1XeP` and then this URL to go slightly further `https://bit.ly/2rGNlXn`

# Chapter 2

# Week 2

## 2.1 Some Discrete Mathematics

### 2.1.1 Logic

Defines a formal language for representing knowledge and for making logical inferences, and supports construction of valid arguments.

**Logic defines**:

1. syntax of statements
2. meaning of statements
3. rules of logical inference (manipulation)

### 2.1.2 Propositional logic

Proposition is a statement which can be either 'True' or 'False'. For example:

***P1***: Pontypridd is in South Wales - TRUE
***P2***: $3 + 5 = 35$ - FALSE

### 2.1.3 Composite statements

More complex statements can be build from elementary statements using logical connectives. For example:

***P3***: It rains outside
***P4***: We will go to cinema
***P3-4***: If it rains outside then we will see a movie

More complex propositional statements can be built from elementary statements using logical connectives. Some of these include

1. Negation
2. Conjunction
3. Disjunction

### 2.1.4 Negation

Let $\boldsymbol{p}$ be a proposition. The statement "It is not the case that p." is another proposition, called the negation of $\boldsymbol{p}$. The negation of $\boldsymbol{p}$ is denoted by $\sim \boldsymbol{p}$ and read as "not $\boldsymbol{p}$". For example:

$\boldsymbol{P5}$: It is raining today
$\sim\boldsymbol{P5}$: It is not raining today

### 2.1.5 Conjunction

Let  and $\boldsymbol{q}$ be propositions. The proposition "$\boldsymbol{p}$ and $\boldsymbol{q}$" denoted by $\boldsymbol{p} \wedge \boldsymbol{q}$, is true when both $\boldsymbol{p}$ and $\boldsymbol{q}$ are true and is false otherwise. The proposition $\boldsymbol{p} \wedge \boldsymbol{q}$ is called the conjunction of $\boldsymbol{p}$ and $\boldsymbol{q}$.
For example:

Let, $\boldsymbol{P5}$ be TRUE
$\boldsymbol{P6}$: 2 is a prime number, is TRUE
$\boldsymbol{P7}$: 24 is a prime number, is FALSE

then:

It is raining today **and** 2 is a prime number - TRUE
It is raining today **and** 24 is a prime number - FALSE

The last two statements can also be written as follows:
$\boldsymbol{P5} \wedge \boldsymbol{P6}$ is TRUE
$\boldsymbol{P5} \wedge \boldsymbol{P7}$ is FALSE

### 2.1.6 Disjunction

Let  and $\boldsymbol{q}$ be propositions. The proposition "$\boldsymbol{p}$ or $\boldsymbol{q}$" denoted by $\boldsymbol{p} \vee \boldsymbol{q}$, is false only when both $\boldsymbol{p}$ and $\boldsymbol{q}$ are false and is true otherwise. The proposition $\boldsymbol{p} \vee \boldsymbol{q}$ is called the disjunction of $\boldsymbol{p}$ and $\boldsymbol{q}$.
For example:

It is raining today **or** 2 is a prime number - TRUE It is raining today **and** 24 is a prime number - TRUE It is not raining today **and** 24 is a prime number - FALSE

The last two statements can also be written as follows:
***P5*** $\vee$ ***P6*** is TRUE
***P5*** $\vee$ ***P7*** is TRUE
$\sim$***P5*** $\vee$ ***P7*** is FALSE

## 2.2 Greedy Algorithms

A greedy algorithm has one fundamental strategy to make decisions/choices, i.e.
At that exact moment in time, what is the optimal choice to make?

For example:

Someone gives a vending machine £1 coin to buy a drink worth £0.70p. A greedy algorithm starts from the highest denomination change the vending machine has and works backwards.

It start with £1, checks whether £1 is more than £30p; as it is more than 30p, the algorithm compares the return value with the next denomination coin i.e. 50p. It reaches £0p and as 20p is less than 30p, it takes one 20p coin.

Now the algorithm needs to return 10p. It check with 20p again, but it is greater than 10p so it moves on to the next coin. The next coin is 10p, which is the exact match. The greedy algorithm stop here as it has made all the decisions to return 30p change.

### 2.2.1 Is greedy optimal?

It is optimal locally, but sometimes it isn't optimal globally. In the change giving algorithm, we can force a point at which it isn't optimal globally.

For example:
Pick 3 denominations of coins. 1p, $x$, and less than $2x$ but more than $x$.

pick 1, 15, 25. (imagine there is coin for each of these three denomination)

Ask for change of 2 * second denomination (15)

ask for change of 30. Now, let's see what our Greedy algorithm does.

It chooses 1x 25p, and 5x 1p. The optimal solution is 2x 15p. As it gives out the same amount in less number of coins.

This Greedy algorithm failed because it didn't look at 15p. It looked at 25p and thought "yup, that fits. Let's take it."

It then looked at 15p and thought "that doesn't fit, let's move on".

This is an example of where Greedy Algorithms fail.

you can find out more about greedy algorithms from the following url as a starting point: `https://www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/`.

## 2.3 Decision Trees

### 2.3.1 ID3 Pseudo code

#### The ID3 Algorithm

**The ID3 Algorithm**

- If all examples have the same label:
  - return a leaf with that label
- Else if there are no features left to test:
  - return a leaf with the most common label
- Else:
  - choose the feature $\hat{F}$ that maximises the information gain of $S$ to be the next node using **Equation (12.2)**
  - add a branch from the node for each possible value $f$ in $\hat{F}$
  - for each branch:
    - calculate $S_f$ by removing $\hat{F}$ from the set of features
    - recursively call the algorithm with $S_f$, to compute the gain relative to the current set of examples

Figure 2.1: ID3 Algorithm Pseudo Code

### 2.3.2 Creating a decision tree

Here is an excerpt from 'Machine Learning' by Stephen Marsland about creating a simple decision tree.

#### 12.4 CLASSIFICATION EXAMPLE

We'll work through an example using ID3 in this section. The data that we'll use will be a continuation of the one we started the chapter with, about what to do in the evening.

When we want to construct the decision tree to decide what to do in the evening, we start by listing everything that we've done for the past few days to get a suitable dataset (here, the last ten days):

| Deadline? | Is there a party? | Lazy? | Activity |
|---|---|---|---|
| Urgent | Yes | Yes | Party |
| Urgent | No | Yes | Study |
| Near | Yes | Yes | Party |
| None | Yes | No | Party |
| None | No | Yes | Pub |
| None | Yes | No | Party |
| Near | No | No | Study |
| Near | No | Yes | TV |
| Near | Yes | Yes | Party |
| Urgent | No | No | Study |

261

Figure 2.2: Decision Tree handout 1

To produce a decision tree for this problem, the first thing that we need to do is work out which feature to use as the root node. We start by computing the entropy of $S$:

$$
\begin{aligned}
\text{Entropy}(S) &= -p_{\text{party}} \log_2 p_{\text{party}} - p_{\text{study}} \log_2 p_{\text{study}} \\
&- p_{\text{pub}} \log_2 p_{\text{pub}} - p_{\text{TV}} \log_2 p_{\text{TV}} \\
&= -\frac{5}{10} \log_2 \frac{5}{10} - \frac{3}{10} \log_2 \frac{3}{10} - \frac{1}{10} \log_2 \frac{1}{10} - \frac{1}{10} \log_2 \frac{1}{10} \\
&= 0.5 + 0.5211 + 0.3322 + 0.3322 = 1.6855 \qquad (12.11)
\end{aligned}
$$

and then find which feature has the maximal information gain:

$$
\begin{aligned}
\text{Gain}(S, \text{Deadline}) &= 1.6855 - \frac{|S_{\text{urgent}}|}{10} \text{Entropy}(S_{\text{urgent}}) \\
&- \frac{|S_{\text{near}}|}{10} \text{Entropy}(S_{\text{near}}) - \frac{|S_{\text{none}}|}{10} \text{Entropy}(S_{\text{none}}) \\
&= 1.6855 - \frac{3}{10} \left( -\frac{2}{3} \log_2 \frac{2}{3} - \frac{1}{3} \log_2 \frac{1}{3} \right) \\
&- \frac{4}{10} \left( -\frac{2}{4} \log_2 \frac{2}{4} - \frac{1}{4} \log_2 \frac{1}{4} - \frac{1}{4} \log_2 \frac{1}{4} \right) \\
&- \frac{3}{10} \left( -\frac{1}{3} \log_2 \frac{1}{3} - \frac{2}{3} \log_2 \frac{2}{3} \right) \\
&= 1.6855 - 0.2755 - 0.6 - 0.2755 \\
&= 0.5345 \qquad (12.12)
\end{aligned}
$$

$$
\begin{aligned}
\text{Gain}(S, \text{Party}) &= 1.6855 - \frac{5}{10} \left( -\frac{5}{5} \log_2 \frac{5}{5} \right) \\
&- \frac{5}{10} \left( -\frac{3}{5} \log_2 \frac{3}{5} - \frac{1}{5} \log_2 \frac{1}{5} - \frac{1}{5} \log_2 \frac{1}{5} \right) \\
&= 1.6855 - 0 - 0.6855 \\
&= 1.0 \qquad (12.13)
\end{aligned}
$$

$$
\begin{aligned}
\text{Gain}(S, \text{Lazy}) &= 1.6855 - \frac{6}{10} \left( -\frac{3}{6} \log_2 \frac{3}{6} - \frac{1}{6} \log_2 \frac{1}{6} - \frac{1}{6} \log_2 \frac{1}{6} - \frac{1}{6} \log_2 \frac{1}{6} \right) \\
&- \frac{4}{10} \left( -\frac{2}{4} \log_2 \frac{2}{4} - \frac{2}{4} \log_2 \frac{2}{4} \right) \\
&= 1.6855 - 1.0755 - 0.4 \\
&= 0.21 \qquad (12.14)
\end{aligned}
$$

Therefore, the root node will be the party feature, which has two feature values ('yes' and 'no'), so it will have two branches coming out of it (see **Figure 12.6**). When we look at the 'yes' branch, we see that in all five cases where there was a party we went to it, so we just put a leaf node there, saying 'party'. For the 'no' branch, out of the five cases there are three different outcomes, so now we need to choose another feature. The five cases we are looking at are:

Figure 2.3: Decision tree handout 2

263

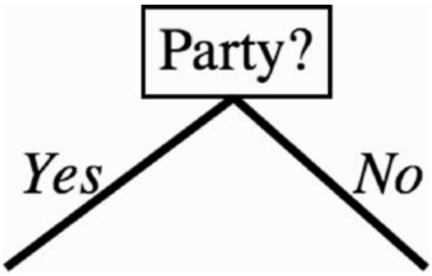



**FIGURE 12.6 The decision tree after one step of the algorithm.**

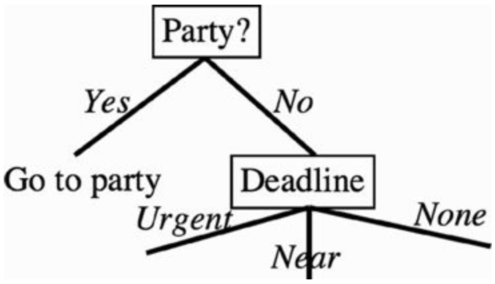



**FIGURE 12.7 The tree after another step.**

| Deadline? | Is there a party? | Lazy? | Activity |
|---|---|---|---|
| Urgent | No | Yes | Study |
| None | No | Yes | Pub |
| Near | No | No | Study |
| Near | No | Yes | TV |
| Urgent | No | Yes | Study |

We've used the party feature, so we just need to calculate the information gain of the other two over these five examples:

Figure 2.4: Decision tree handout 3

We've used the party feature, so we just need to calculate the information gain of the other two over these five examples:

$$
\begin{aligned}
\text{Gain}(S, \text{Deadline}) &= 1.371 - \frac{2}{5}\left(-\frac{2}{2}\log_2\frac{2}{2}\right) \\
&- \frac{2}{5}\left(-\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2}\right) - \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 1.371 - 0 - 0.4 - 0 \\
&= 0.971 \qquad (12.15)
\end{aligned}
$$

$$
\begin{aligned}
\text{Gain}(S, \text{Lazy}) &= 1.371 - \frac{4}{5}\left(-\frac{2}{4}\log_2\frac{2}{4} - \frac{1}{4}\log_2\frac{1}{4} - \frac{1}{4}\log_2\frac{1}{4}\right) \\
&- \frac{1}{5}\left(-\frac{1}{1}\log_2\frac{1}{1}\right) \\
&= 1.371 - 1.2 - 0 \\
&= 0.1710 \qquad (12.16)
\end{aligned}
$$

This leads to the tree shown in **Figure 12.7**. From this point it is relatively simple to complete the tree, leading to the one that was shown in **Figure 12.1**.

Figure 2.5: Decision tree handout 4

### 2.3.3 Gini Index

Here is an excerpt from Stephen Marsland's "Machine Leaning, An Algorithmic Perspective", second edition to introduce you to Gini impurity.

### 12.3.1 Gini Impurity

The entropy that was used in ID3 as the information measure is not the only way to pick features. Another possibility is something known as the `Gini impurity`. The 'impurity' in the name suggests that the aim of the decision tree is to have each leaf node represent a set of datapoints that are in the same class, so that there are no mismatches. This is known as purity. If a leaf is pure then all of the training data within it have just one class. In which case, if we count the number of datapoints at the node (or better, the fraction of the number of datapoints) that belong to a class $i$ (call it $N(i)$), then it should be 0 for all except one value of $i$. So suppose that you want to decide on which feature to choose for a split. The algorithm loops over the different features and checks how many points belong to each class. If the node is pure, then $N(i) = 0$ for all values of $i$ except one particular one. So for any particular feature $k$ you can compute:

$$G_k = \sum_{i=1}^{c} \sum_{j \neq i} N(i)N(j), \qquad (12.8)$$

where $c$ is the number of classes. In fact, you can reduce the algorithmic effort required by noticing that $\sum_i N(i) = 1$ (since there has to be some output class) and so $\sum_{j \neq i} N(j) = 1 - N(i)$. Then Equation (12.8) is equivalent to:

$$G_k = 1 - \sum_{i=1}^{c} N(i)^2. \qquad (12.9)$$

Either way, the Gini impurity is equivalent to computing the expected error rate if the classification was picked according to the class distribution. The information gain can then be measured in the same way, subtracting each value $G_i$ from the total Gini impurity. 260

The information measure can be changed in another way, which is to add a weight to the misclassifications. The idea is to consider the cost of misclassifying an instance of class $i$ as class $j$ (which we will call the `risk` in Section 2.3.1) and add a weight that says how important each datapoint is. It is typically labelled as $\lambda_{ij}$ and is presented as a matrix, with element $\lambda_{ij}$ representing the cost of misclassifying $i$ as $j$. Using it is simple, modifying the Gini impurity (Equation (12.8)) to be: 261

$$G_i = \sum_{j \neq i} \lambda_{ij} N(i)N(j). \qquad (12.10)$$

We will see in Section 13.1 that there is another benefit to using these weights, which is to successively improve the classification ability by putting higher weight on datapoints that the algorithm is getting wrong.

Figure 2.6: Gini Index

## 2.4 Error Metrics for Regression

There is a range of error metrics that can be utilised to calculate the error of a Learning Algorithm which performs regression. A few of them are discussed here as follows.

### 2.4.1 Mean Squared Error MSE

One of the simplest and common metric for regression evaluation. MSE measures the averagge squared error of the predictions. The higher the value the more the error.

One bad predition (a prediction with a large error) can make the metric's over all value worse than it actually is. A false sense of poor model performance is reflected in such scenarios.

Similary, if the errors at all test prediction instances are small, let's say less than one, then an opposite effect is reflected in this metric i.e, it makes the observer overestimate the model's performance.

$$MSE = \frac{1}{N} \sum_{i=0}^{N} (y_i - y_i')^2$$

$y_i$ is the actual output and $y_i'$ is the model's predicted output.

### 2.4.2 Root Mean Squared Error

It is the sqaure root of the MSE's value.

### 2.4.3 Mean Absolute Error

## 2.5 Suggested further reading

**For decision trees**

Here is a simple research paper which discusses some of the popular decision trees in a very comprehensive manner.
`https://saiconference.com/Downloads/SpecialIssueNo10/Paper_3-A_comparative_study_of_decision_tree_ID3_and_C4.5.pdf`

If you would like to delve further into decision trees, "C4.5: Programs for Machine Learning" by J. Ross Quinlan is a popular and a recommended read in this area. Quinlan is responsible for introducing us to classic decision tree algorithms like ID3 and C4.5.

# Chapter 3

# Week 3

## 3.1 Random variable distributions

Here are a couple of excerpts from the classic book on probability called "A first course in probability", $7^th$ edition by Sheldon Ross. They are provided to enhance your understanding of a few random variable distributions. The idea is to comprehend the possible usage of these distributions when discussed in the lecture with respect to some learning algorithms.

It will be helpful if, you could quickly go through your MS4S08 lecture notes to revisit Normal, Multinomial, Binomial and Bernoulli distributions.

### 3.1.1 Bernoulli and binomial distributions

150 Chapter 4 Random Variables

**EXAMPLE 5a**

Calculate $\mathrm{Var}(X)$ if $X$ represents the outcome when a fair die is rolled.

***Solution.*** It was shown in Example 3a that $E[X] = \frac{7}{2}$. Also,

$$
\begin{aligned}
E[X^2] &= 1^2\left(\tfrac{1}{6}\right) + 2^2\left(\tfrac{1}{6}\right) + 3^2\left(\tfrac{1}{6}\right) + 4^2\left(\tfrac{1}{6}\right) + 5^2\left(\tfrac{1}{6}\right) + 6^2\left(\tfrac{1}{6}\right) \\
&= \left(\tfrac{1}{6}\right)(91)
\end{aligned}
$$

Hence

$$\mathrm{Var}(X) = \frac{91}{6} - \left(\frac{7}{2}\right)^2 = \frac{35}{12} \qquad \blacksquare$$

A useful identity is that for any constants $a$ and $b$.

$$\mathrm{Var}(aX + b) = a^2\mathrm{Var}(X)$$

To prove the preceding, let $\mu = E[X]$ and note that from Corollary 4.1, $E[aX + b] = a\mu + b$. Hence

$$
\begin{aligned}
\mathrm{Var}(aX + b) &= E[(aX + b - a\mu - b)^2] \\
&= E[a^2(X - \mu)^2] \\
&= a^2E[(X - \mu)^2] \\
&= a^2\mathrm{Var}(X)
\end{aligned}
$$

**Remarks**. (a) Analogous to the mean being the center of gravity of a distribution of mass, the variance represents, in the terminology of mechanics, the moment of inertia.

(b) The square root of the $\mathrm{Var}(X)$ is called the *standard deviation* of $X$, and we denote it by $\mathrm{SD}(X)$. That is,

$$\mathrm{SD}(X) = \sqrt{\mathrm{Var}(X)}$$

Discrete random variables are often classified according to their probability mass function. In the next few sections we consider some of the more common types.

**4.6 THE BERNOULLI AND BINOMIAL RANDOM VARIABLES**

Suppose that a trial, or an experiment, whose outcome can be classified as either a *success* or a *failure* is performed. If we let $X = 1$ when the outcome is a success and $X = 0$ when it is a failure, then the probability mass function of $X$ is given by

$$
\begin{aligned}
p(0) &= P\{X = 0\} = 1 - p \\
p(1) &= P\{X = 1\} = p
\end{aligned}
\tag{6.1}
$$

where $p$, $0 \le p \le 1$, is the probability that the trial is a success.

Figure 3.1: Bernoulli Distribution and Binomial Distribution 1

A random variable $X$ is said to be a Bernoulli random variable (after the Swiss mathematician James Bernoulli) if its probability mass function is given by Equations (6.1) for some $p \in (0, 1)$.

Suppose now that $n$ independent trials, each of which results in a success with probability $p$ and in a failure with probability $1 - p$, are to be performed. If $X$ represents the number of successes that occur in the $n$ trials, then $X$ is said to be a *binomial* random variable with parameters $(n, p)$. Thus a Bernoulli random variable is just a binomial random variable with parameters $(1, p)$.

The probability mass function of a binomial random variable having parameters $(n, p)$ is given by

$$p(i) = \binom{n}{i} p^i (1 - p)^{n-i} \qquad i = 0, 1, \ldots, n \tag{6.2}$$

The validity of Equation (6.2) may be verified by first noting that the probability of any particular sequence of $n$ outcomes containing $i$ successes and $n - i$ failures is, by the assumed independence of trials, $p^i(1 - p)^{n-i}$. Equation (6.2) then follows, since there are $\binom{n}{i}$ different sequences of the $n$ outcomes leading to $i$ successes and $n - i$ failures. This perhaps can most easily be seen by noting that there are $\binom{n}{i}$ different choices of the $i$ trials that result in successes. For instance, if $n = 4, i = 2$, then there are $\binom{4}{2} = 6$ ways in which the four trials can result in two successes, namely, any of the outcomes $(s, s, f, f)$, $(s, f, s, f)$, $(s, f, f, s)$, $(f, s, s, f)$, $(f, s, f, s)$, or $(f, f, s, s)$, where the outcome $(s, s, f, f)$ means, for instance, that the first two trials are successes and the last two failures. Since each of these outcomes has probability $p^2(1 - p)^2$ of occurring, the desired probability of two successes in the four trials is thus $\binom{4}{2} p^2(1 - p)^2$.

Note that by the binomial theorem, the probabilities sum to 1; that is,

$$\sum_{i=0}^{\infty} p(i) = \sum_{i=0}^{n} (ni)\, p^i (1 - p)^{n-i} = [p + (1 - p)]^n = 1$$

**EXAMPLE 6a**

Five fair coins are flipped. If the outcomes are assumed independent, find the probability mass function of the number of heads obtained.

***Solution.*** If we let $X$ equal the number of heads (successes) that appear, then $X$ is a binomial random variable with parameters $\left(n = 5,\ p = \frac{1}{2}\right)$. Hence, by Equation (6.2),

$$P\{X = 0\} = \binom{5}{0} \left(\frac{1}{2}\right)^0 \left(\frac{1}{2}\right)^5 = \frac{1}{32}$$

Figure 3.2: Bernoulli Distribution and Binomial Distribution 2

$$P\{X = 1\} = \binom{5}{1}\left(\frac{1}{2}\right)^1\left(\frac{1}{2}\right)^4 = \frac{5}{32}$$

$$P\{X = 2\} = \binom{5}{2}\left(\frac{1}{2}\right)^2\left(\frac{1}{2}\right)^3 = \frac{10}{32}$$

$$P\{X = 3\} = \binom{5}{3}\left(\frac{1}{2}\right)^3\left(\frac{1}{2}\right)^2 = \frac{10}{32}$$

$$P\{X = 4\} = \binom{5}{4}\left(\frac{1}{2}\right)^4\left(\frac{1}{2}\right)^1 = \frac{5}{32}$$

$$P\{X = 5\} = \binom{5}{5}\left(\frac{1}{2}\right)^5\left(\frac{1}{2}\right)^0 = \frac{1}{32}$$

■

**EXAMPLE 6b**

It is known that screws produced by a certain company will be defective with probability .01 independently of each other. The company sells the screws in packages of 10 and offers a money-back guarantee that at most 1 of the 10 screws is defective. What proportion of packages sold must the company replace?

***Solution.*** If $X$ is the number of defective screws in a package, then $X$ is a binomial random variable with parameters (10, .01). Hence the probability that a package will have to be replaced is

$$1 - P\{X = 0\} - P\{X = 1\} = 1 - \binom{10}{0}(.01)^0(.99)^{10} - \binom{10}{1}(.01)^1(.99)^9$$
$$\approx .004$$

Hence only .4 percent of the packages will have to be replaced. ■

**EXAMPLE 6c**

The following gambling game, known as the wheel of fortune (or chuck-a-luck), is quite popular at many carnivals and gambling casinos: A player bets on one of the numbers 1 through 6. Three dice are then rolled, and if the number bet by the player appears $i$ times, $i = 1, 2, 3$, then the player wins $i$ units; on the other hand, if the number bet by the player does not appear on any of the dice, then the player loses 1 unit. Is this game fair to the player? (Actually, the game is played by spinning a wheel that comes to rest on a slot labeled by three of the numbers 1 through 6, but it is mathematically equivalent to the dice version.)

***Solution.*** If we assume that the dice are fair and act independently of each other, then the number of times that the number bet appears is a binomial random variable with parameters $\left(3, \frac{1}{6}\right)$. Hence, letting $X$ denote the player's winnings in the game, we have

$$P\{X = -1\} = \binom{3}{0}\left(\frac{1}{6}\right)^0\left(\frac{5}{6}\right)^3 = \frac{125}{216}$$

Figure 3.3: Bernoulli Distribution and Binomial Distribution 3

$$P\{X = 1\} = \binom{3}{1}\left(\frac{1}{6}\right)^1\left(\frac{5}{6}\right)^2 = \frac{75}{216}$$

$$P\{X = 2\} = \binom{3}{2}\left(\frac{1}{6}\right)^2\left(\frac{5}{6}\right)^1 = \frac{15}{216}$$

$$P\{X = 3\} = \binom{3}{3}\left(\frac{1}{6}\right)^3\left(\frac{5}{6}\right)^0 = \frac{1}{216}$$

In order to determine whether or not this is a fair game for the player, let us determine $E[X]$. From the preceding probabilities we obtain

$$\begin{aligned} E[X] &= \frac{-125 + 75 + 30 + 3}{216} \\ &= \frac{-17}{216} \end{aligned}$$

Hence, in the long run, the player will lose 17 units per every 216 games he plays. ■

In the next example we consider the simplest form of the theory of inheritance as developed by G. Mendel (1822–1884).

**EXAMPLE 6d**

Suppose that a particular trait (such as eye color or left handedness) of a person is classified on the basis of one pair of genes and suppose that $d$ represents a dominant gene and $r$ a recessive gene. Thus a person with $dd$ genes is pure dominance, one with $rr$ is pure recessive, and one with $rd$ is hybrid. The pure dominance and the hybrid are alike in appearance. Children receive 1 gene from each parent. If, with respect to a particular trait, 2 hybrid parents have a total of 4 children, what is the probability that 3 of the 4 children have the outward appearance of the dominant gene?

***Solution.*** If we assume that each child is equally likely to inherit either of 2 genes from each parent, the probabilities that the child of 2 hybrid parents will have $dd$, $rr$, or $rd$ pairs of genes are, respectively, $\frac{1}{4}$, $\frac{1}{4}$, $\frac{1}{2}$. Hence, as an offspring will have the outward appearance of the dominant gene if its gene pair is either $dd$ or $rd$, it follows that the number of such children is binomially distributed with parameters $\left(4, \frac{3}{4}\right)$. Thus the desired probability is

$$\binom{4}{3}\left(\frac{3}{4}\right)^3\left(\frac{1}{4}\right)^1 = \frac{27}{64}$$

■

**EXAMPLE 6e**

Consider a jury trial in which it takes 8 of the 12 jurors to convict; that is, in order for the defendant to be convicted, at least 8 of the jurors must vote him guilty. If we assume that jurors act independently and each makes the right decision with probability $\theta$, what is the probability that the jury renders a correct decision?

Figure 3.4: Bernoulli Distribution and Binomial Distribution 4

***Solution.*** The problem, as stated, is incapable of solution, for there is not yet enough information. For instance, if the defendant is innocent, the probability of the jury's rendering a correct decision is

$$\sum_{i=5}^{12} \binom{12}{i} \theta^i (1-\theta)^{12-i}$$

whereas, if he is guilty, the probability of a correct decision is

$$\sum_{i=8}^{12} \binom{12}{i} \theta^i (1-\theta)^{12-i}$$

Therefore, if $\alpha$ represents the probability that the defendant is guilty, then, by conditioning on whether or not he is guilty, we obtain that the probability that the jury renders a correct decision is

$$\alpha \sum_{i=8}^{12} \binom{12}{i} \theta^i (1-\theta)^{12-i} + (1-\alpha) \sum_{i=5}^{12} \binom{12}{i} \theta^i (1-\theta)^{12-i} \qquad \blacksquare$$

**EXAMPLE 6f**

A communication system consists of $n$ components, each of which will, independently, function with probability $p$. The total system will be able to operate effectively if at least one-half of its components function.

**(a)** For what values of $p$ is a 5-component system more likely to operate effectively than a 3-component system?

**(b)** In general, when is a $(2k+1)$-component system better than a $(2k-1)$-component system?

***Solution.*** (a) As the number of functioning components is a binomial random variable with parameters $(n, p)$, it follows that the probability that a 5-component system will be effective is

$$\binom{5}{3} p^3(1-p)^2 + \binom{5}{4} p^4(1-p) + p^5$$

whereas the corresponding probability for a 3-component system is

$$\binom{3}{2} p^2(1-p) + p^3$$

Hence, the 5-component system is better if

$$10p^3(1-p)^2 + 5p^4(1-p) + p^5 > 3p^2(1-p) + p^3$$

which reduces to

$$3(p-1)^2(2p-1) > 0$$

Figure 3.5: Bernoulli Distribution and Binomial Distribution 5

or

$$p > \tfrac{1}{2}$$

(b) In general, a system with $2k + 1$ components will be better than one with $2k - 1$ components if (and only if) $p > \frac{1}{2}$. To prove this, consider a system of $2k + 1$ components and let $X$ denote the number of the first $2k - 1$ that function. Then

$$\begin{aligned} P_{2k+1}(\text{effective}) &= P\{X \geq k+1\} + P\{X = k\}(1 - (1-p)^2) \\ &\quad + P\{X = k-1\}p^2 \end{aligned}$$

which follows since the $(2k + 1)$-component system will be effective if either

**(i)** $X \geq k + 1$;
**(ii)** $X = k$ and at least one of the remaining 2 components function; or
**(iii)** $X = k - 1$ and both of the next 2 function.

As

$$\begin{aligned} P_{2k-1}(\text{effective}) &= P\{X \geq k\} \\ &= P\{X = k\} + P\{X \geq k+1\} \end{aligned}$$

we obtain

$$\begin{aligned} &P_{2k+1}(\text{effective}) - P_{2k-1}(\text{effective}) \\ &\quad = P\{X = k-1\}p^2 - (1-p)^2 P\{X = k\} \\ &\quad = \binom{2k-1}{k-1} p^{k-1}(1-p)^k p^2 - (1-p)^2 \binom{2k-1}{k} p^k (1-p)^{k-1} \\ &\quad = \binom{2k-1}{k} p^k (1-p)^k [p - (1-p)] \quad \text{since } \binom{2k-1}{k-1} = \binom{2k-1}{k} \\ &\quad > 0 \Leftrightarrow p > \tfrac{1}{2} \end{aligned}$$

■

### 4.6.1 Properties of Binomial Random Variables

We will now examine the properties of a binomial random variable with parameters $n$ and $p$. To begin, let us compute its expected value and variance. Now,

$$\begin{aligned} E[X^k] &= \sum_{i=0}^{n} i^k \binom{n}{i} p^i (1-p)^{n-i} \\ &= \sum_{i=1}^{n} i^k \binom{n}{i} p^i (1-p)^{n-i} \end{aligned}$$

Using the identity

$$i \binom{n}{i} = n \binom{n-1}{i-1}$$

Figure 3.6: Bernoulli Distribution and Binomial Distribution 6

gives that

$$
\begin{aligned}
E[X^k] &= np \sum_{i=1}^{n} i^{k-1} \binom{n-1}{i-1} p^{i-1}(1-p)^{n-i} \\
&= np \sum_{j=0}^{n-1} (j+1)^{k-1} \binom{n-1}{j} p^{j}(1-p)^{n-1-j} \qquad \text{by letting } j = i-1 \\
&= npE[(Y+1)^{k-1}]
\end{aligned}
$$

where $Y$ is a binomial random variable with parameters $n-1$, $p$. Setting $k=1$ in the preceding equation yields

$$E[X] = np$$

That is, the expected number of successes that occur in $n$ independent trials when each is a success with probability $p$ is equal to $np$. Setting $k=2$ in the preceding equation, and using the preceding formula for the expected value of a binomial random variable, gives that

$$
\begin{aligned}
E[X^2] &= npE[Y+1] \\
&= np[(n-1)p+1]
\end{aligned}
$$

Since $E[X] = np$ we obtain

$$
\begin{aligned}
\text{Var}(X) &= E[X^2] - (E[X])^2 \\
&= np[(n-1)p+1] - (np)^2 \\
&= np(1-p)
\end{aligned}
$$

Summing up, we have shown the following:

If $X$ is a binomial random variable with parameters $n$ and $p$, then

$$
\begin{aligned}
E[X] &= np \\
\text{Var}(X) &= np(1-p)
\end{aligned}
$$

The following proposition details how the binomial probability mass function first increases and then decreases.

***Proposition 6.1***
If $X$ is a binomial random variable with parameters $(n, p)$, where $0 < p < 1$, then as $k$ goes from 0 to $n$, $P\{X = k\}$ first increases monotonically and then decreases monotonically, reaching its largest value when $k$ is the largest integer less than or equal to $(n+1)p$.

Figure 3.7: Bernoulli Distribution and Binomial Distribution 7

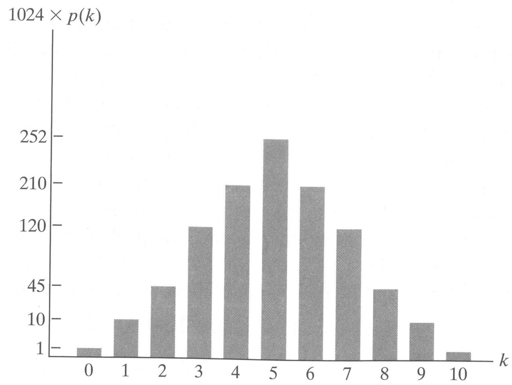



**Figure 4.5.** Graph of $p(k) = \binom{10}{k}\left(\frac{1}{2}\right)^{10}$

**Proof:** We prove the proposition by considering $P\{X = k\}/P\{X = k - 1\}$ and determining for what values of $k$ it is greater or less than 1. Now,

$$\frac{P\{X = k\}}{P\{X = k-1\}} = \frac{\dfrac{n!}{(n-k)!k!}p^k(1-p)^{n-k}}{\dfrac{n!}{(n-k+1)!(k-1)!}p^{k-1}(1-p)^{n-k+1}}$$
$$= \frac{(n-k+1)p}{k(1-p)}$$

Hence $P\{X = k\} \geq P\{X = k - 1\}$ if and only if

$$(n - k + 1)p \geq k(1 - p)$$

or, equivalently, if and only if

$$k \leq (n + 1)p$$

and the proposition is proved. ✦

As an illustration of Proposition 6.1 consider Figure 4.5, the graph of the probability mass function of a binomial random variable with parameters $(10, \frac{1}{2})$.

### EXAMPLE 6g

In a U.S. presidential election the candidate who gains the maximum number of votes in a state is awarded the total number of electoral college votes allocated to that state. The number of electoral college votes of a given state is roughly proportional to the population of that state—that is, a state of population size $n$ has roughly $nc$ electoral votes. (Actually, it is closer to $nc + 2$ as a state is given an electoral vote for each

Figure 3.8: Bernoulli Distribution and Binomial Distribution 8

member of the House of Representatives, the number of such representatives being roughly proportional to its population, and one electoral college vote for each of its two senators.) Let us determine the average power in a close presidential election of a citizen in a state of size $n$, where by *average power* in a close election we mean the following: A voter in a state of size $n = 2k+1$ will be decisive if the other $n-1$ voters split their votes evenly between the two candidates. (We are assuming here that $n$ is odd, but the case where $n$ is even is quite similar.) As the election is close, we shall suppose that each of the other $n - 1 = 2k$ voters acts independently and is equally likely to vote for either candidate. Hence the probability that a voter in a state of size $n = 2k + 1$ will make a difference to the outcome is the same as the probability that $2k$ tosses of a fair coin lands heads and tails an equal number of times. That is,

$$\begin{aligned} &P\{\text{voter in state of size } 2k+1 \text{ makes a difference}\} \\ &\quad = \binom{2k}{k}\left(\frac{1}{2}\right)^k\left(\frac{1}{2}\right)^k \\ &\quad = \frac{(2k)!}{k!k!2^{2k}} \end{aligned}$$

To approximate the preceding, we make use of Stirling's approximation, which says that for $k$ large,

$$k! \sim k^{k+1/2}e^{-k}\sqrt{2\pi}$$

where we say that $a_k \sim b_k$ when the ratio $a_k/b_k$ approaches 1 as $k$ approaches $\infty$. Hence we see that

$$\begin{aligned} &P\{\text{voter in state of size } 2k+1 \text{ makes a difference}\} \\ &\quad \sim \frac{(2k)^{2k+1/2}e^{-2k}\sqrt{2\pi}}{k^{2k+1}e^{-2k}(2\pi)2^{2k}} = \frac{1}{\sqrt{k\pi}} \end{aligned}$$

As such a voter will, if he or she makes a difference, affect $nc$ electoral votes, we see that the expected number of electoral votes a voter in a state of size $n$ will affect—or the voter's average power—is given by

$$\begin{aligned} \text{average power} &= ncP\{\text{makes a difference}\} \\ &\sim \frac{nc}{\sqrt{n\pi/2}} \\ &= c\sqrt{2n/\pi} \end{aligned}$$

Hence the average power of a voter in a state of size $n$ is proportional to the square root of $n$, thus showing that in presidential elections, voters in large states have more power than do those in smaller states. ■

### 4.6.2 Computing the Binomial Distribution Function

Suppose that $X$ is binomial with parameters $(n, p)$. The key to computing its distribution function

$$P\{X \le i\} = \sum_{k=0}^{i}\binom{n}{k}p^k(1-p)^{n-k} \qquad i = 0, 1, \ldots, n$$

Figure 3.9: Bernoulli Distribution and Binomial Distribution 9

is to utilize the following relationship between $P\{X = k + 1\}$ and $P\{X = k\}$, which was established in the proof of Proposition 6.1:

$$P\{X = k + 1\} = \frac{p}{1 - p}\frac{n - k}{k + 1}P\{X = k\} \tag{6.3}$$

**EXAMPLE 6h**

Let $X$ be a binomial random variable with parameters $n = 6$, $p = .4$. Then, starting with $P\{X = 0\} = (.6)^6$ and recursively employing Equation (6.3), we obtain

$$\begin{aligned}
P\{X = 0\} &= (.6)^6 \approx .0467 \\
P\{X = 1\} &= \tfrac{4}{6}\tfrac{6}{1}P\{X = 0\} \approx .1866 \\
P\{X = 2\} &= \tfrac{4}{6}\tfrac{5}{2}P\{X = 1\} \approx .3110 \\
P\{X = 3\} &= \tfrac{4}{6}\tfrac{4}{3}P\{X = 2\} \approx .2765 \\
P\{X = 4\} &= \tfrac{4}{6}\tfrac{3}{4}P\{X = 3\} \approx .1382 \\
P\{X = 5\} &= \tfrac{4}{6}\tfrac{2}{5}P\{X = 4\} \approx .0369 \\
P\{X = 6\} &= \tfrac{4}{6}\tfrac{1}{6}P\{X = 5\} \approx .0041
\end{aligned}$$

■

A computer program that utilizes the recursion (6.3) to compute the binomial distribution function is easily written. To compute $P\{X \leq i\}$ the program should compute first $P\{X = i\}$ and then use the recursion to compute successively $P\{X = i - 1\}$, $P\{X = i - 2\}$, and so on. Such a program is on the website. In utilizing it, one enters the binomial parameters $n$ and $p$ and a value $i$ and the program computes the probabilities that a binomial $(n, p)$ random variable is equal to and is less than or equal to $i$.

## Historical Note

Independent trials having a common success probability $p$ were first studied by the Swiss mathematician Jacques Bernoulli (1654–1705). In his book *Ars Conjectandi* (the Art of Conjecturing), published by his nephew Nicholas eight years after his death in 1713, Bernoulli showed that if the number of such trials were large, then the proportion of them that were successes would be close to $p$ with a probability near 1.

Jacques Bernoulli was from the first generation of the most famous mathematical family of all time. Altogether there were between eight and twelve Bernoullis, spread over three generations, who made fundamental contributions to probability, statistics, and mathematics. One difficulty in knowing their exact number is the fact that several had the same name. (For example, two of the sons of Jacques's brother Jean were named Jacques and Jean.) Another difficulty is that several of the Bernoullis were known by different names in different

Figure 3.10: Bernoulli Distribution and Binomial Distribution 10

### 3.1.2 Multinomial Distribution

266 Chapter 6 Jointly Distributed Random Variables

***Solution.*** We start by computing the distribution function of $X/Y$. For $a > 0$,

$$
\begin{aligned}
F_{X/Y}(a) &= P\left\{\frac{X}{Y} \le a\right\} \\
&= \iint\limits_{x/y \le a} e^{-(x+y)}\,dx\,dy \\
&= \int_0^\infty \int_0^{ay} e^{-(x+y)}\,dx\,dy \\
&= \int_0^\infty (1 - e^{-ay})e^{-y}\,dy \\
&= \left\{-e^{-y} + \frac{e^{-(a+1)y}}{a+1}\right\}\Bigg|_0^\infty \\
&= 1 - \frac{1}{a+1}
\end{aligned}
$$

Differentiation yields that the density function of $X/Y$ is given by $f_{X/Y}(a) = 1/(a + 1)^2, 0 < a < \infty$. ■

We can also define joint probability distributions for $n$ random variables in exactly the same manner as we did for $n = 2$. For instance, the joint cumulative probability distribution function $F(a_1, a_2, \ldots, a_n)$ of the $n$ random variables $X_1, X_2, \ldots, X_n$ is defined by

$$F(a_1, a_2, \ldots, a_n) = P\{X_1 \le a_1, X_2 \le a_2, \ldots, X_n \le a_n\}$$

Further, the $n$ random variables are said to be jointly continuous if there exists a function $f(x_1, x_2, \ldots, x_n)$, called the joint probability density function, such that for any set $C$ in $n$-space

$$P\{(X_1, X_2, \ldots, X_n) \in C\} = \iint\limits_{(x_1,\ldots,x_n)\in C} \cdots \int f(x_1, \ldots, x_n)dx_1dx_2\cdots dx_n$$

In particular, for any $n$ sets of real numbers $A_1, A_2, \ldots, A_n$,

$$
\begin{aligned}
&P\{X_1 \in A_1, X_2, \in A_2, \ldots, X_n \in A_n\} \\
&\quad = \int_{A_n}\int_{A_{n-1}} \cdots \int_{A_1} f(x_1, \ldots, x_n)\,dx_1dx_2\cdots dx_n
\end{aligned}
$$

**EXAMPLE 1f The multinomial distribution**

One of the most important joint distributions is the multinomial, which arises when a sequence of $n$ independent and identical experiments is performed. Suppose that each experiment can result in any one of $r$ possible outcomes, with respective probabilities

Figure 3.11: Multinomial Distribution 1

$p_1, p_2, \ldots, p_r, \sum_{i=1}^{r} p_i = 1$. If we let $X_i$ denote the number of the $n$ experiments that result in outcome number $i$, then

$$P\{X_1 = n_1, X_2 = n_2, \ldots, X_r = n_r\} = \frac{n!}{n_1! n_2! \cdots n_r!} p_1^{n_1} p_2^{n_2} \cdots p_r^{n_r} \tag{1.5}$$

whenever $\sum_{i=1}^{r} n_i = n$.

Equation (1.5) is verified by noting that any sequence of outcomes for the $n$ experiments that leads to outcome $i$ occurring $n_i$ times for $i = 1, 2, \ldots, r$, will, by the assumed independence of experiments, have probability $p_1^{n_1} p_2^{n_2} \ldots p_r^{n_r}$ of occurring. As there are $n!/(n_1! n_2! \ldots n_r!)$ such sequences of outcomes (there are $n!/n_1! \ldots n_r!$ different permutations of $n$ things of which $n_1$ are alike, $n_2$ are alike, $\ldots$, $n_r$ are alike). Equation (1.5) is established. The joint distribution whose joint probability mass function is specified by Equation (1.5) is called the multinomial distribution. The reader should note that when $r = 2$, the multinomial reduces to the binomial distribution.

As an application of the multinomial, suppose that a fair die is rolled 9 times. The probability that 1 appears three times, 2 and 3 twice each, 4 and 5 once each, and 6 not at all is

$$\frac{9!}{3!2!2!1!1!0!}\left(\frac{1}{6}\right)^3\left(\frac{1}{6}\right)^2\left(\frac{1}{6}\right)^2\left(\frac{1}{6}\right)^1\left(\frac{1}{6}\right)^1\left(\frac{1}{6}\right)^0 = \frac{9!}{3!2!2!}\left(\frac{1}{6}\right)^9 \qquad \blacksquare$$

## 6.2 INDEPENDENT RANDOM VARIABLES

The random variables $X$ and $Y$ are said to be *independent* if for any two sets of real numbers $A$ and $B$,

$$P\{X \in A, Y \in B\} = P\{X \in A\}P\{Y \in B\} \tag{2.1}$$

In other words, $X$ and $Y$ are independent if, for all $A$ and $B$, the events $E_A = \{X \in A\}$ and $F_B = \{Y \in B\}$ are independent.

It can be shown by using the three axioms of probability that Equation (2.1) will follow if and only if for all $a, b$,

$$P\{X \le a, Y \le b\} = P\{X \le a\}P\{Y \le b\}$$

Hence, in terms of the joint distribution function $F$ of $X$ and $Y$, we have that $X$ and $Y$ are independent if

$$F(a, b) = F_X(a)F_Y(b) \qquad \text{for all } a, b$$

When $X$ and $Y$ are discrete random variables, the condition of independence (2.1) is equivalent to

$$p(x, y) = p_X(x)p_Y(y) \qquad \text{for all } x, y \tag{2.2}$$

The equivalence follows because, if (2.1) is satisfied, then we obtain Equation (2.2) by letting $A$ and $B$ be, respectively, the one point sets $A = \{x\}$, $B = \{y\}$. Furthermore, if

Figure 3.12: Multinomial Distribution 2