MS4S10 - Machine Learning and Decision Making

Supplementary Resources

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

University of South Wales

Contents

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/`.

You can install python manually or via anaconda. We'll see both types of installations in the following subsections.

1.1.2 Manual Installation

Figure 1.1: Python's default installation window - (the image will be updated later)

Figure 1.2: Checking python's version on your computer - (the image will be updated later)

1. Download the latest stable version (3.9.0 for now), or later 3.8.x from the following link: `https://www.python.org/downloads/`
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 3.9/3.8 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 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.
2. follow steps 2 - 4 from 1.1.2 for executing first python script from Jupyter notebook

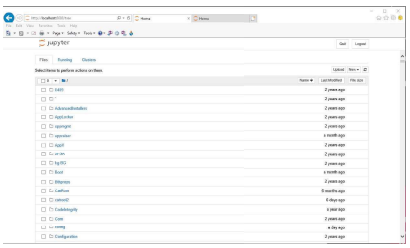

Figure 1.3: Jupyter Notebook in browser

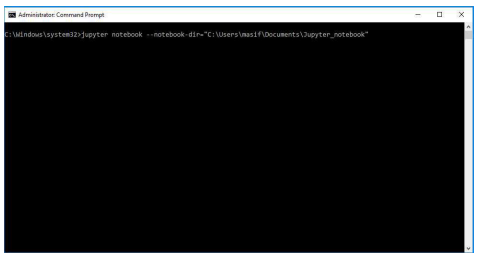

Figure 1.4: Changing Jupyter notebook directory

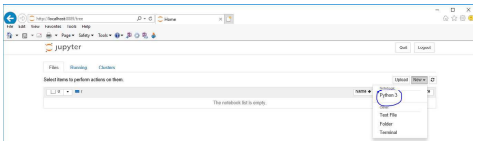

Figure 1.5: Creating new python script

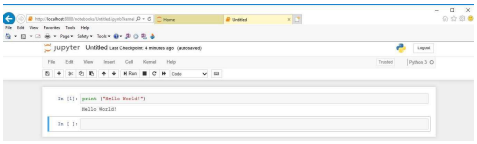

Figure 1.6: First python script: hello world

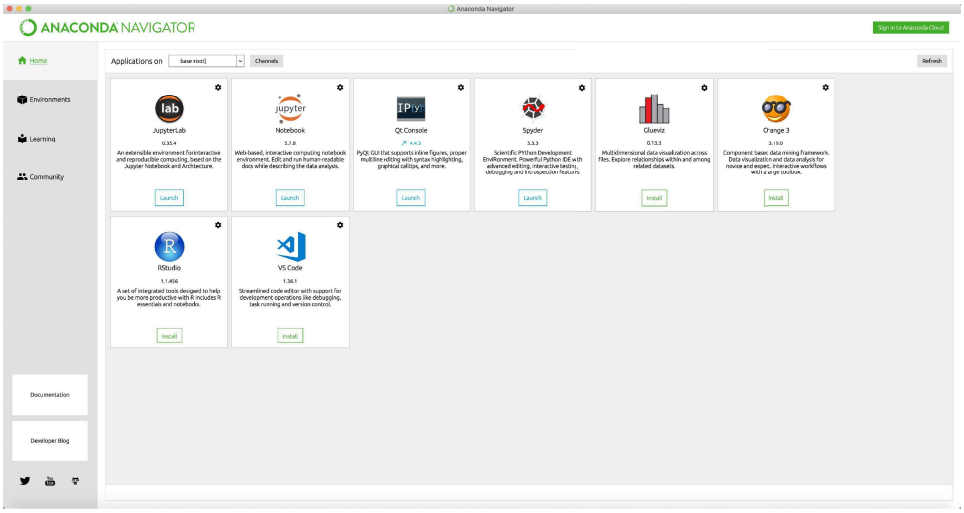

Figure 1.7: Anaconda navigator home

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

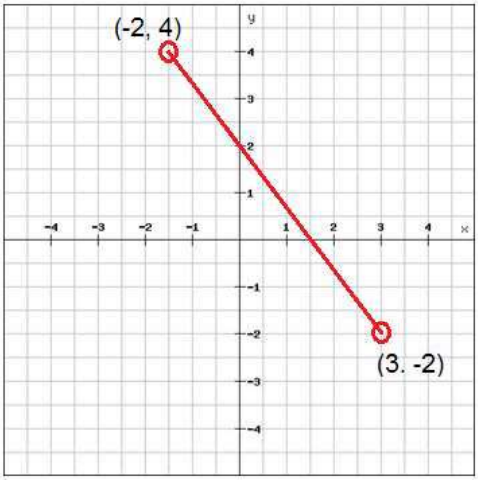

Figure 1.8: line segment PQ, p(-2,4) and q(3,-2)

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:

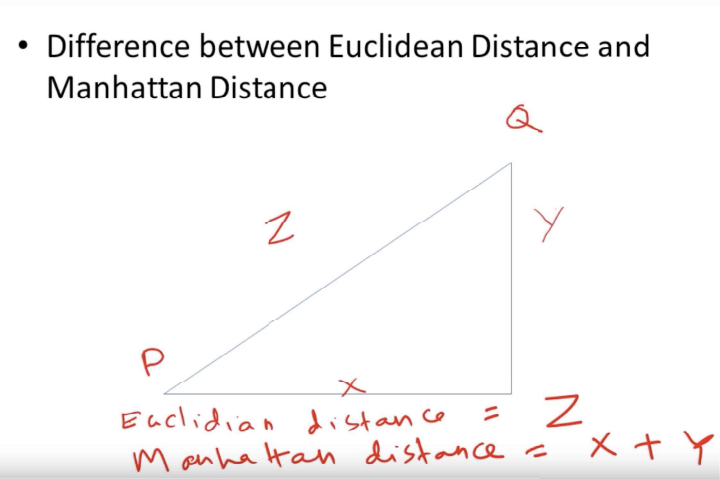

Figure 1.9: Euclidean distance vs Manhattan Distance

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

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.

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))
```


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.

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

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

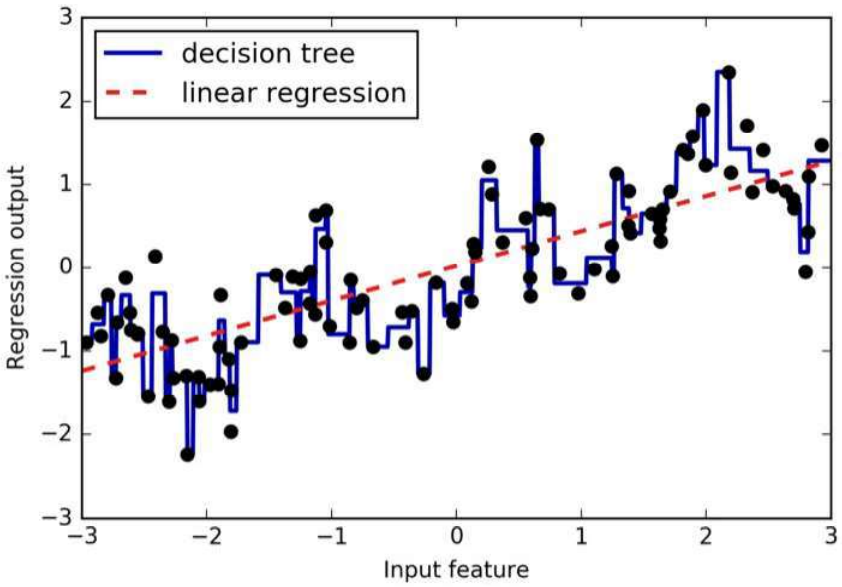

Figure 4-1. Comparing linear regression and a decision tree on the wave dataset

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")
```

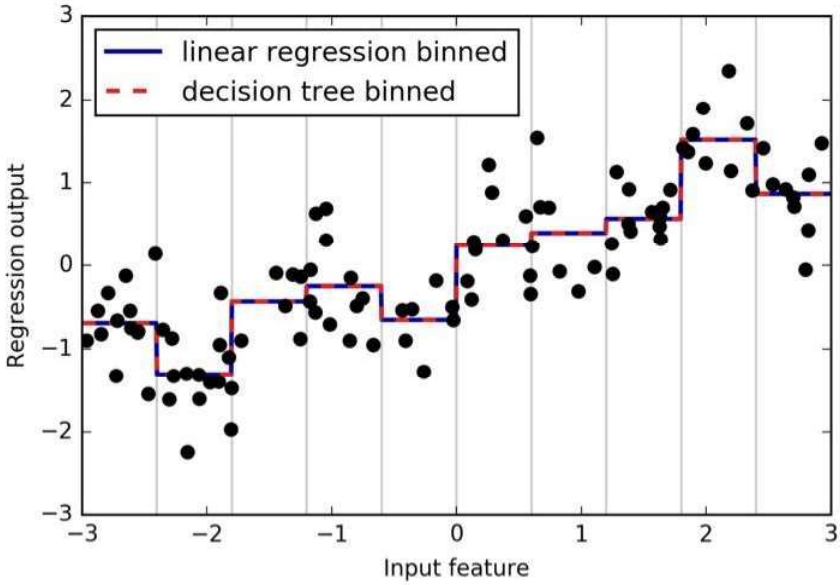

Figure 4-2. Comparing linear regression and decision tree regression on binned features

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:

P5: It is raining today
$\sim$***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, ***P5*** be TRUE
P6: 2 is a prime number, is TRUE
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:
P5 $\wedge$ ***P6*** is TRUE
P5 $\wedge$ ***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:
$\boldsymbol{P5} \vee \boldsymbol{P6}$ is TRUE
$\boldsymbol{P5} \vee \boldsymbol{P7}$ is TRUE
$\sim\boldsymbol{P5} \vee \boldsymbol{P7}$ 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

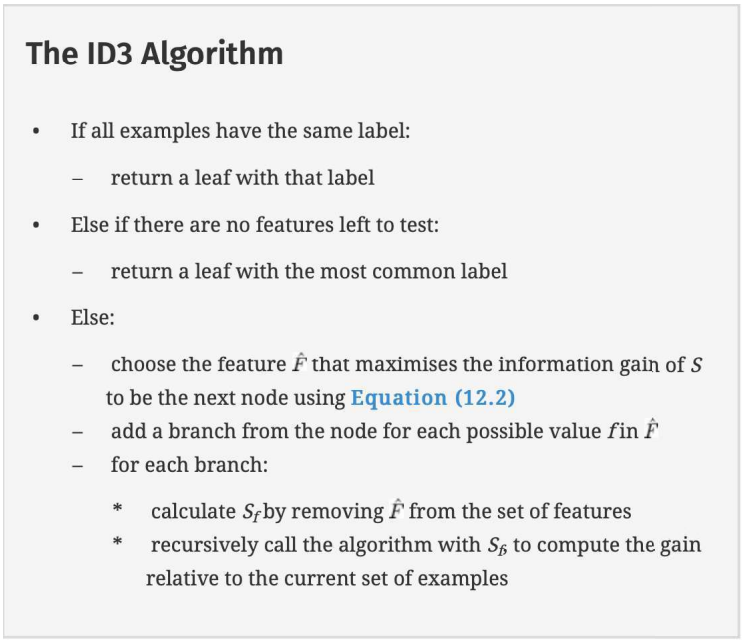

Figure 2.1: ID3 Algorithm Pseudo Code

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

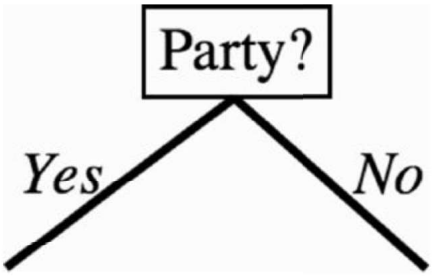

FIGURE 12.6 The decision tree after one step of the algorithm.

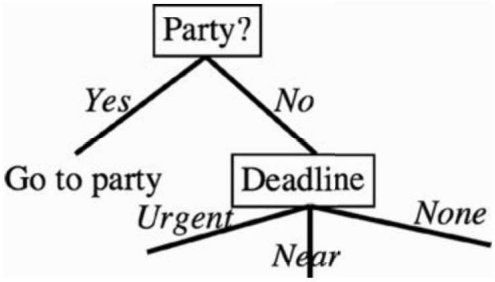

FIGURE 12.7 The tree after another step.

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:

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

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

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:

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

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:

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

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.

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 λ_{ij} and is presented as a matrix, with element λ_{ij} representing the cost of misclassifying i as j. Using it is simple, modifying the Gini impurity (Equation (12.8)) to be:

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

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 further reading for regression metric

Kindly have a look at the following link `https://bit.ly/35Bfelx` to see the formulae for some other important regression error metrics. It is not recommended that you agree with the author's stance through out this blog.

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.

Visualising decision trees - post lecture

Due to some technical difficulties that we have been facing over the past couple of weeks, an important aspect of decision tree algorithms could not be demonstrated in the lecture. It is quite easy to graphically visualise decision trees rule (if-then): the conjunction of disjunctions.

It is **highly** suggested that students familiarise themselves with Decision tree visualtion options provided by python on the following link:`https://scikit-learn.org/stable/modules/tree.html`