Practical 2

by Chris Emmery

In this practical[1] we will focus on making predictions, ascertaining the correctness of these predictions, and trying to partly improve them.

[1] **Important Practical Note**: If you cannot answer a task during the practicals fully, or feel unsure about your answer (even after the explanation), please ask! It is very important that you develop the correct intuitions for each of the points we discuss here. Sometimes they just don't 'click' by themselves; they require a lot of repeated practice and interpretation, and not every explanation works for everyone. We will be very happy to answer all your questions on the Forum or during office hours!

Preparing for Prediction

From what you have seen up until now, you can probably make a pretty good prediction if someone survived the Titanic. Say that we offer you this table of people as instances and some of their features:

	gender	age	pclass	fare
1	m	51	higher	80
2	f	40	middle	60
3	m	25	lower	10

How would you rank the survivability of these three persons? Now for a bit more complicated one:

	gender	age	pclass	fare
1	m	27	higher	?
2	f	21	middle	?
3	f	35	lower	?

What would you predict is (about) the fair that these three person bought their tickets for?

Data Mining

This is exactly the kind of process we want to automate using Data Mining techniques. Up until now, we've used the Titanic data, which is pretty simple — can be well understood in historical context by humans, and therefore you can at least say something about the likelihood of survival. The fares on the other hand require a bit more of a complex view on the data. You can sort of guess what ballpark the fares would be in based on the `pclass`, but the nuances are harder. These two tasks involve prediction: survived — a discrete feature (`classification`), where we have a limited set of options to choose from, and fare — a continuous feature (`regression`), where we'd like to be as close to the actual number as possible.

Data Mining is in general the combination of several techniques:

1. Managing your data.
2. A thorough understanding of its contents and potential.
3. The ability to manually select and alter the data to create useful insights and visualizations.
4. Understanding and applying predictive models that use the full complexity of the data to create even better insights.
5. Making sure these models are correctly evaluated and being able to judge their usefulness.
6. Communicating these results.

This course will mostly focus on 2-5. The practicals will try to train you in 3-5. However, 2 will require your own effort but is the most important step to the success of actually applying 3-5.

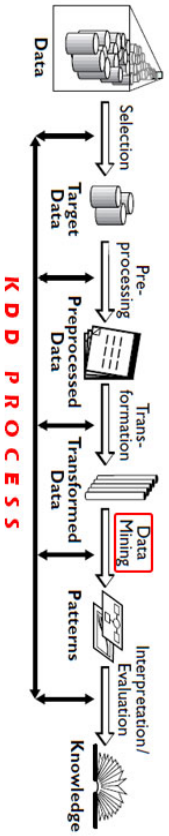

Figure 1: Formal steps of a Data Mining (KDD) workflow.

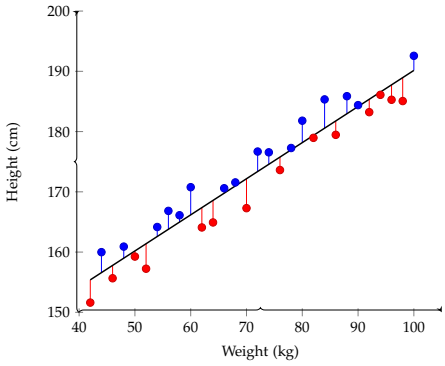

Figure 2: Linear Regression and residuals (error). For 2-d (1 feature, 1 target), the fit of the line is determined by $Y = \beta_0 + \beta_1 \cdot X$ where β_0 is the intercept (or bias coefficient) and β_1 the coefficient for the first feature (determining the slope of the line).

Making Predictions

For this practical, we are first going to take a model-driven approach rather than a data-driven one; we'll try to fit a Linear Regression model (see Figure 2 for a quick visual refresher) that hopefully predicts with low error, and see what kind of information we can get from it. As our data, we'll be looking at the Boston housing dataset from Harrison & Rubenfield — collected by the U.S Census Service concerning housing in the area of Boston Mass. It's a standard 'toy' dataset for regression, given that it only has 506 instances[2].

[2]

Information	task	data
Data	raw	github

If you're still unsure how prediction and error assessment for this type of model works, try the following given bias and coefficients:

```
 -0.1080 * crime-rate +
  0.0464 * zoned +
  0.0206 * industry +
  2.6867 * charles +
-17.7666 * nitric-oxide +
  3.8099 * rooms +
  0.0007 * age +
 -1.4756 * employment-center +
  0.3061 * radial-highways +
 -0.0123 * property-tax +
 -0.9527 * pupil-teach-ratio +
  0.0093 * proportion-black-families +
 -0.5247 * poor-people +
 36.4595
```

You are provided with the following test data:

crime-rate	zoned	industry	charles	nitric-oxide	rooms	age	employment-center	radial-highways	property-tax	pupil-teach-ratio	proportion-black-families	poor-people
0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98

Task 1

- Use the given coefficients and bias to predict $\hat{y}$: the median-value (by hand) for the feature vectors in the table above (X_{test}). See below for instructions.

Given the true median values[3] (so the actual price of the houses) Y, use the predicted median-value $\hat{Y}$ to calculate the Root Mean Squared Error (RMSE) for the median-value in Footnote 3. You do this by (for each of the feature vectors) subtracting the actual (y_i) from the predicted ($\hat{y}_i$) value, and squaring them. Here, i is the index of any instance y in Y (and thus X). After, you take the sum over all these values (should be $i = 1$ value, because 1 instance), divide it by the amount of predictions ($n = 1$), and take the root[4].

So, that was a tedious exercise you probably don't want to do again. Luckily, we can automate the fitting of the regression line, making predictions, and calculating RMSE with `scikit-learn`.

[3]
$$Y = \langle 24 \rangle \tag{1}$$

[4]
$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{n}} \tag{2}$$

Linear Regression with Scikit-Learn

In the following piece of code we are going to recode the column names[5] with more interpretable ones. For an explanation of what's going on, see sidenote[6].

```
new_columns = ["crime-rate", "zoned", "industry", "charles",
               "nitric-oxide", "rooms", "age", "employment-center",
               "radial-highways", "property-tax", "pupil-teach-ratio",
               "proportion-black-families", "poor-people", "median-value"]

# using from_csv
df = pd.DataFrame.from_csv('/srv/data-mining/data/Housing/housing.csv',
                                                    index_col=None)
df.columns = new_columns  # replace the initial columns with new names

# using smarter read_csv
df = pd.read_csv('/srv/data-mining/data/Housing/housing.csv',
                                 index_col=None, names=new_columns, header=0)

df
```

[5] Which immediately reveal this an old dataset with little regard for political correctness.

[6] To boil it down, `new_columns` is a list; an ordered collection of strings (our column names between quotes). Because we can look up what the actual column names represent, and we know the order in which they occur in the data (check Table **??**), we can manually recode them in the order that they occur in. After, we can either overwrite the names (`from_csv()` method), or pass the names directly (`read_csv()` method). Either way, we now have human-readable feature names.

To set up for predicting anything, we first have to split our data into `features` and a `target`. Hence, we want to put `median-value` into its own variable[7] Y — and everything else into X. For the label, we can just simply select the column from the dataframe and put it in `y`, like so:

```
y = df["median-value"]
```

[7] Please be aware of the mixed use of the word `variable` between the Python variable (a name reference to a particular object, like an integer, list, or dataframe) and that used in data mining (where it is explicitly a feature vector).

Now comes a bit more complicated syntax for selecting everything **but** the `median-value` from the dataframe. For this we can use the `loc()` method[8]. As such:

```
X = df.loc[:, df.columns != "median-value"]
```

[8] This 'locates' (thus subsets) columns that are according to a certain condition. Our specific condition is that we want all but `median value`. We could manually type out all the column names, but `df.columns != "median-value"` gives us all the indices that are not the median value. If you try the command by itself you'll see that it gives a list of `True` and `False` when something meets the condition (is not `median-value`). The `loc` function has two parts for subsetting: by rows (first part between `[ , ]`) and by columns (second part). Given that we want all the rows, we type `:`, and provide the indices in the second part.

Now, we want to create a train and test set. We can do this pretty easily with the following piece of code:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=0.2, random_state=42)
```

Here, `train_test_split()` returns multiple objects, so that's why there's more variable names before the = sign. The splitting does multiple things under the hood:

- It makes sure the training set (`X_train`) is `split` into 80% of the total data, and the test set (`X_test`) is 20% (`test_size=0.2`.
- It `stratifies` the data, so that the distribution of the `target` is the same in the training set as it is in the set. As such, there's a

proportionally equal amount of houses in both sets (i.e. if the total data had 10, test now has 2, and train 8).

- It randomly `shuffles` the data, so that the model can't infer anything out of the order in which the dataset was composed. As a result, if we want to be able to reproduce our exact results, we need to provide a seed to this (`random_state=42`) so that the 'random' order is the same if we were to repeat the experiment.

For regression, we first initialize the model (this is where you'd want to set the parameters). After, we can call the `fit()` method on our `training set` (`X_train`), and labels (`y_train`)[9].

[9] `scikit-learn` has very comprehensible documentation. Look up the parameters for the model and its methods!

```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(X_train, y_train)
```

The classifier is now `fitted`, or trained. We can first try to interpret the `coefficients`. This can be done by calling the `coef_` attribute[10], and then zipping (concatenating)[11] it with the column names, like below:

[10] See the Attributes section in the documentation.

[11] No need to really understand what's going on here. Basically, the coefficient values are ordered according to our features, so here we just link the column names (minus our target) with those values. Regardless, just remember it as a way to look at the coefficients.

```
list(zip(df.columns[:-1], lr.coef_))
```

Now that the model is trained, we can use it to perform prediction on our (unseen) `test set` (`X_test`), and then evaluate how far the predicted values are from the *actual* true values (`y_test`). We do this by providing one of the `sklearn` metrics (`mean_squared_error()`) with these two variables. Sadly, `scikit-learn` doesn't provide the Root part (to bring the units of error back to that of our actual target), but we can off-ship that to `NumPy` (using `np.sqrt`).

```
from sklearn.metrics import mean_squared_error
import numpy as np

y_pred = lr.predict(X_test)
rmse_model = np.sqrt(mean_squared_error(y_test, y_pred))
rmse_model
```

Interpret all the output we have generated up until now. Then finally, solve:

Task 2

- Fit Linear Regression on the Boston Housing dataset, interpret the coefficients. What would these mean?
- How do we asses if the model has a) actually learned something, and b) will generalize well?

Further Analysis

With our model-driven results we have some empirical indication of how well we can predict a house price in Boston given the features (and data) we were provided with. Note that even if very successful, this shouldn't be presented as a general model of predicting house prizes. The final scores we are looking at are constrained by:

- Our `data` (amount and sample) — we have Boston housing data, and arguably a very small sample. One could question how well that translates to other cities, different varieties of houses, etc. Can we predict rural areas? Residential areas with many expensive houses?
- Our `model` (algorithm and parameters) — we chose for the simplest implementation of regression; Linear Regression. We did not tune the algorithm to perform better, nor have we thought about choosing other models that might be better suited for the type of data we are dealing with.
- Our `evaluation` (scheme and scores) — while RMSE gives a fair indication of how well a particular model approximates the actual pattern underlying the data, we do need to realize how much data we are using for training and testing. We might also have to look at some other evaluation metrics to get a better indication (like MAE). Moreover, if we will tune any of our parameters, we need to consider doing that on a different `data split` altogether. The latter we're not going into for this particular practical.

Anyway, let's get into some analysis. If we want a nicer visualization of the coefficients, we can simply dump them in a dataframe and get a bar plot going. Like so:

```
cdf = pd.DataFrame([list(lr.coef_)], columns=df.columns[:-1])
plt = cdf.plot(figsize=(15,5), kind='bar', legend=True)
plt.xaxis.set_visible(False)
```

We can control the size to see the bars a bit better using `figsize`, where the first part of `figsize=( , )` is the width, and the second the height. Because there are no values on the x-axis of the plot, we can just turn that off. For this we have to store the plot in a variable first (`plt`), after which we can manipulate the visibility (last line). Mostly you will see how much (visual) impact the `nitric-oxide` feature has in the model that we fitted. Rooms also seems to have quite an impact — both should be somewhat in accordance with your intuition about what controls a house price.

Baseline

Up until now we've only looked at the test scores; we don't actually know if our model actually learned something (i.e. if it's not

making some stupid predictions). An easy way to get a baseline score for a regression model is to take an `mean baseline`. For this, we can simply take:

```
df["median-value"].mean()
```

We'll have to make this prediction as many times as we have test items though, so we can take the length (`len`) of `y_test`, and fill a list up with as many times the mean. This looks like:

```
bl = [df["median-value"].mean()] * len(y_test)
bl
```

Now we have 502 entries with the exact same baseline prediction, great. We can compare it with the true `y_test` and see how far we are off:

```
rmse_bl = np.sqrt(mean_squared_error(y_test, bl))
rmse_bl
```

Task 3

- Interpret the baseline score and compare it to our previous RMSE score. How would you say the model is performing?
- Looking at the error, and `median-value`, determine if the model is useful in practice.

Post-hoc Feature Analysis

The regression coefficients have given us some indication of feature importance. We know which one affect the price positively, and which do negatively — and to what intensity. But how much does that correspond to what we would expect from a general correlation between that specific feature, and our target? Let's find out:

```
from scipy.stats import pearsonr
df.plot(x='nitric-oxide', y='median-value', kind='scatter')
pearsonr(df['nitric-oxide'], df['median-value'])
```

Previously, you could have observed `nitric-oxide` to have a strong negative effect. According to our model, weighing even heavier than the amount of `rooms`. Let's inspect that one as well:

```
df.plot(x='rooms', y='median-value', kind='scatter')
pearsonr(df['rooms'], df['median-value'])
```

Task 4

- Interpreting the plots (and more importantly the correlations), do these correspond to the regression coefficients?
- What is the reason behind this?

More Plots

As we're looking at plots anways, let's try to see if we can get a bit more insight into the Housing dataset. For example, it's cheap to live in polluted areas:

```
df.plot(x='nitric-oxide', y='median-value', c='property-tax',
                colormap='Oranges', kind='scatter')
```

Older buildings are in less crowded areas:

```
df.plot(x='zoned', y='pupil-teach-ratio', c='age',
                colormap='Blues', kind='scatter')
```

Why Normalize?

As we arrive at boxplots, hopefully you'll see some oddities and issues with the following plot[12]:

```
df.boxplot(figsize=(20,5), rot=90)
```

[12] Note that `rot` is used to rotate the labels on the x-axis by 90 degrees to fit the names.

Most of the plots are barely visible, because the ranges of the feature values differ tremendously. There's quite a simple fix for that, which is normalizing by z-score.

```
df_norm = (df - df.mean()) / (df.max() - df.min())
df_norm.boxplot(figsize=(20,5), rot=90)
```

In addition to fixing our boxplot, standardizing or normalizing also potentially improves algorithms (as the space they need to fit a line in is greatly reduced). Let's see how it affects ours, start with:

```
X = df_norm.loc[:, df_norm.columns != "median-value"]
```

Task 5

- From here on out you are on your own. Use the new normalized X[13] to get a `rmse_norm` and compare it to the baseline and `rmse_model`. How much did we improve?

[13] Repeat all steps we did before from there on out.

Solutions

Task 1:

- The predicted value should be around 30. This makes the RMSE around 6.

Task 2:

- Positive means it increases the house price by some factor, negative coefficients mean it decreases the price by this factor. Note that they are on different scales though!
- a) we would need some 'stupid' baseline that we can use to predict without learning anything (like the mean of all the values that we saw in our training dataset). b) we have a test set, so we know the performance on unseen data. Actual generalizability isn't easy to comment on with small datasets. The more instances we have (especially across time, these are housing market dynamics mind you), the more confident we can be that it still holds.

Task 3:

- Baseline error should be around 8.6, whilst our model error is around 4.9. From this, we can at least conclude that is quite a bit better than baseline — the model is at least learning some pattern in the data. We have to further interpret the error to make an actual quality estimate, however. See below.
- Let's see what this `median-value` actually represents. The documentation gives the following info: *"Median value of owner-occupied homes in $ 1000's"*. This means that per house we are generally about $5000 (plus or minus) off. Looking at `df["median-value"].mean()`, it can be observed that the average house costs about $ 22500 with a standard deviation (`df["median-value"].std()`) of about $ 9200. I'm not in the house selling business, but if I would have a reasonable chance of losing about $ 5000 (when a lower price is predicted) — I'd not expect to run much profit given the standard deviation. You can further inspect `y_pred` and `y_test` (using e.g. `np.mean`, or just outputting the values) to see how far the model is off in some specific cases (spoiler: on average it undershoots the prices).

Task 4:

- No, they don't. There's a way stronger correlation between `rooms` and `median-value` than the negative for `nitric-oxide`.
- Linear Regression models *multiple* dimensions simultaneously when adding **all** features. As such, some features might be weighted lower (e.g. if their values have a smaller range). You can try the boxplot again *after* normalizing to get a 'better' representation.

Task 5:

- RMSE for the normalized model should be exactly the same. Note that this applies to standard features; when later we go into interaction features, the importance will become more evident.