Practical 4

by Chris Emmery

This practical[1] will mostly focus on discussing the previous one — creating some insights and intuitions about this particular dataset, and hopefully will help you in understanding some of the material you might have had a hard time with in the previous practical. We will dive into how datasets can be misleading, (hopefully you'll agree) in somewhat of light-hearted fashion.

[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 office hours!

Setting up for Predictions

When acquiring data yourself, there's always a clear task or goal. However, when working with pre-made, or pre-collected data—which is commonly the case in industry applications—it is often your goal to discover purpose in data. Moreover, it might also be the case that you already have a particular goal in mind, but are still able to find surprising patterns with some thorough analysis.

As is often mentioned, collection, cleaning, and general preparation of data covers the majority of the effort in Data Science research. Learning about common issues in the mess that is 'in-the-wild' data (and ways to resolve them quickest) takes practice and patience. Even when dealing with pre-made data, as you have seen in the previous practical, the combination of your data and tools can pose quite the obstruction on the way to your final station: doing predictions.

There's a manifold of different issues that arise in the process described above, we will cover the following in this practical:

- Getting your data in the correct data types.
- Plotting to gain extra insight and identifying anomalies in data.
- Preprocessing to improve the information in your features.

Predictions we will leave for later — this practical covers the 'dirty' Data Mining work.

Let's get started.

Interpreting Raw Data

When data is in an unfamiliar format (which is almost always, unless you collected and stored it yourself), a good first step is opening up the files in some plain text editor (e.g. Notepad for Windows, TextEdit for Mac, or whatever your distro provides for Linux). You can also choose to do this via Jupyter / terminal for example:

```
!head --lines=2 /srv/data-mining/data/IMDB/imdb.csv
```

The first line (starting with `color`, ending with `,quality`) are the headers of the `.csv` file, the second line (starting with `Color`, ending with `,good`) is the first instance. Naturally, using a package like Pandas for reading and manipulating such data structures makes the visual interpretation much easier.

```
import pandas as pd

df = pd.DataFrame.from_csv('/srv/data-mining/data/IMDB/imdb.csv', index_col=None)
df.head(n=1)
```

Exploring the Data Types

Pandas does somewhat of a lazy reading of data. When using `from_csv`, or not setting the correct options in `read_csv` — it's going to try to naively interpret the features, and doesn't warn you about for example mixed values[2]. A priori, you don't know anything of this dataset, so finding out the hard way is common. Let's see what happens if we don't handle missing values in our reader:

[2] This for example happens when you have mixed data types, such as numbers and ? as missing value indicator. It will assume ? is just a regular value, and you end up with an `object` type column.

```
df['gross'].head()
```

Pandas sees this as an `object`, and not an integer, or float, as we would prefer. This is why we'd generally want to use `pd.read_csv` for this, as we can indicate the *encoding* of the missing values. As such:

```
df = pd.read_csv('/srv/data-mining/data/IMDB/imdb.csv', index_col=None, na_values='?')
df['gross'].head()
```

However, notice that we could think of this fix only because we either encountered the `?` issue before, or did some manual analysis of the data. There might be other values still messing up our data types. Let's check to be completely sure:

```
df.dtypes
```

Task 1

- Given the intuitions that you have about the features — are the datatypes correct?
- What is the difference between `float64` and `int64`? Can you find what causes this for features that likely aren't a floating point number?

Plotting

Now that all our features are in place, let's start plotting some things:

```
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')
```

Plots are a great way to gain extra insight in data. However, in order to produce them, usually some data aggregation / filtering methods need to be employed. While there are numerous ways to visualize this small dataset to bits, we'll look at a few different ones and how it allows us to critically analyse a particular dataset that we're not familiar with. After all, while the features are somewhat self-explanatory, we don't know how they were collected, and if we can trust them. First, we'll take a look at the 'most popular first actors'.

```
p_df = df.drop_duplicates('actor_1_name')
p_df = p_df.sort_values('actor_1_facebook_likes', ascending=False)
p_df = p_df.head(50)
p_df.plot(x='actor_1_name', y='actor_1_facebook_likes',
                  kind='bar', figsize=(20, 5))
```

Code Explained:

- Actors can play in multiple movies, but their facebook likes will be the same. Therefore, we drop duplicate entries of `actor_1_name` (note that this only makes sense for this particular plot, that's why we're subsetting).
- Sort these values based on `actor_1_facebook_likes`, in a descending fashion (i.e. from large to small).
- To not overflow the plot, we only take the first 50.
- Finally we plot our sorted, subsetted data.

Mysterious Popularity

What can we get from this plot? I personally found it interesting that Darcy Donavan has so many facebook likes (full disclosure, mostly because I didn't know who she is).

Task 2

- Look up Darcy Donavan and Matthew Ziff on IMDB. Do you think their roles would make them the most popular actors on facebook?

- Look them up on facebook. Do the likes of the plot match that of Facebook?

Now, let's inspect the movies they were supposedly starring in as first actor:

```
df[df['actor_1_name'] == 'Darcy-Donavan'][
        ['movie_title', 'actor_1_name',
     'actor_2_name', 'actor_1_facebook_likes']]
```

Code Explained: the first part subsets the entries to only those where 'actor_1_name' is 'Darcy-Donavan', and then the second part selects the rows.

Task 3

- Look up the cast of Anchorman on IMDB. Where should Will Ferell be according to the IMDB ordering?
- Are the actors perhaps ordered alphabetically?
- Are the likes for Will Ferell correct?

If you've seen the video lecture `Working with Text Data: Part II`, you know frequencies in human behaviour often follow a Zipfian (or Power Law) distribution. The facebook pages that are very popular should, assuming this law holds, have exponentially more likes than the ones that are a little less popular. So much so, that on a double logarithmic scale, they look almost linear. Run the following code:

```
l_df = df['actor_1_facebook_likes'].sort_values(ascending=False)
l_df = l_df.reset_index()
l_df.plot(y='actor_1_facebook_likes', kind='line', loglog=True)
```

Task 4

- Do likes follow a Zipfian distribution?
- Why / why not?

Inspecting Correlations

Anyway, enough about the facebook likes, let's try to find out some correlations. We can try to plot all of them at once with the `scatter_matrix` function, but it does take quite a while to render. Try if you like:

```
pd.scatter_matrix(df, alpha=0.2, figsize=(50, 50), diagonal='kde')
```

Some of the more interesting ones (in my opinion) would be these:

Are movies getting worse?

```
df.plot(kind='scatter', x='title_year', y='imdb_score')

# --- new cell ---

from scipy.stats import pearsonr
corr = pearsonr(df['title_year'].fillna(0), df['imdb_score'].fillna(0))
print("r={0}, p={1}".format(*corr))
```

Are people more inclined to review good movies?

```
df.plot(kind='scatter', x='num_user_for_reviews', y='imdb_score')

# --- new cell ---

from scipy.stats import pearsonr
corr = pearsonr(df['num_user_for_reviews'].fillna(0), df['imdb_score'].fillna(0))
print("r={0}, p={1}".format(*corr))
```

Does money generate more money?

```
df.plot(kind='scatter', x='gross', y='budget')

# --- new cell ---

from scipy.stats import pearsonr
corr = pearsonr(df['duration'].fillna(0), df['gross'].fillna(0))
print("r={0}, p={1}".format(*corr))
```

Now, hopefully we can agree these questions above are way overstated. There's no way a sample of just 5000 movies is going to show us anything definitive enough to answer these questions. As you can see from the distributions like the one below, the set is very biased towards more recent movies. At the very least, it shows you how to use scatter plots informatively. Let's for consider the final `gross*budget` plot in particular. There are a few enormous outliers both in budget as well as gross that beg for further inspection.

```
df['title_year'].plot(kind='kde')
```

We Want More Money!

So, what about these outliers. If we just read from the plot, we know exactly in what range we need to find the highest `gross`ing movies:

```
df[df['gross'] > 6 * 1e8][['movie_title', 'gross', 'budget']
        ].sort_values('gross', ascending=False)
```

Code Explained: the first part subsets the entries to only those where `gross` is higher than 600000000, and then the second part selects the rows. Lastly, they are sorted by gross (descending).

Now, Avatar does rank amongst the most expensive films ever, but should also be the highest-grossing ones. So let's confirm if this is the case.

```
df[df['budget'] > 0.2*1e10][['movie_title', 'gross', 'budget']
        ].sort_values('budget', ascending=False)
```

Task 5

- Look up the movies on IMDB, do you notice something about their movie posters?
- Why are these movies generating outliers?
- How would you tackle this?

Popular Genres

Now we're going to use some boxplots to see if there are certain genres that are particularly popular.

```
df['genres'] = df['genres'].apply(lambda x: x.split('|')[0])
df.boxplot(column=['imdb_score'], by='genres', figsize=(20, 4))
```

This gives us information regarding which genre has the most outliers in terms of good (or bad score), which ones have the highest variance, overall best scores, etc. For example, Romance seems to get very consistent scores between 5-7 whereas 'Family' seems to have many different scores across the board. Popularity of a genre also comes into play here though:

```
df.boxplot(column=['num_user_for_reviews'], by='genres', figsize=(20, 10))
```

Task 6

- Which genre overall gets best reviews?
- Look at the second plot. For this particular genre, what can you deduce from the plot?

Finally, just to show what pivot tables are[3] and how you can colour them. If you have multiple categories that you want to compare certain data on (just more than just raw count), you can use the `pivot_table()` method. You can specify a categorical index, column(s), and the values you want to show in this table. For convenient sorting of quality, I replaced them by numbers. Note that due to the mixed currencies, the values aren't particularly informative — but there are some funny inferences to be made, such as that in the UK more budget seems to yield lower scores, and in the USA it's the other way around (scroll all the way to the right).

See:

[3] Excel people are big fans of these.

```
df.replace({"quality": {"very-bad": 1, "bad": 2, "okay": 3,
                                        "good": 4, "very-good": 5}}, inplace=True)
df.pivot_table(index='quality', columns='country',
                           values='budget').style.background_gradient(cmap='Blues')
```

Preprocessing

Now that we've looked at some interesting characteristics of this particular dataset, I'll discuss a few methods of how to enrich, and fix some of the errors that we found. Now again, please keep in mind that there's many more things that can be done — I'm just showing a few. For most of the 'actual' preprocessing, you need a bit more scripting knowledge. An example of that could be identifying the conversion rate of the 'gross' values based on the country (you'd need some API that does the conversion to dollars).

Dealing with Highly Unique Features

As we've discussed in the Text-Mining-related video lectures, language is often represented as text vectors. However, what happens when we're dealing with actor names? These are not actually embedded in a piece of text, and we can't split them into separate tokens (chances of the same name being an important feature are slim). But, it does contain some information, intuitively at least. When you see names such as e.g. Ryan Gosling, Dave Bautista, Robin Wright, Jared Leto, and Harrison Ford on one poster (totally random composition, honest), you could make an educated guess that there needs to be something very wrong with a movie for it to get a low IMDB score. So, we associate some quality with certain names.

A cheap way to convey this kind of world knowledge is replacing the actor names with the average of the imdb scores for the movies they have starred in. This is actually quite a straight-forward operation in pandas:

```
df.groupby('actor_1_name')['imdb_score'].mean().sort_values(
        ascending=False).head(10)
```

However, mean score might not be enough. We would also want to introduce some reliability score. The way I solved it here is definitely not the most elegant, but at least it accounts for the fact that (a) popularity is important (e.g. starting in a lot of movies), and (b) the scores of those movies are important. Now note that this metric can be offset by just being old (see most of the listed actors below) AND popular. Though, by using the sum over the z-scores for movies scores an actor starred in, we're at least provided with some indication how consistently 'good' an actor is.

```
from scipy.stats.mstats import zscore

df['imdb_z'] = zscore(df['imdb_score'])

pd.Series(df.groupby('actor_1_name')['imdb_z'].sum()
         ).sort_values(ascending=False).head(10)
```

Now, to me this at least looks like a fair list. Note that when we don't take the amount of movies into consideration, we get some very different results:

```
pd.Series(df.groupby('actor_1_name')['imdb_z'].mean()
         ).sort_values(ascending=False).head(10)
```

Dealing with Text Data

So, what do we do when we *do* have text data? While we could set up a prediction task for the movie score based on the plot alone, this dataset doesn't provide us with the rich textual representation required for a proper computational linguistics approach. Instead, we'll take the keywords from the plot:

```
df['plot_keywords'].head(5)
```

We'll have to do two things here: split the different keywords, and then convert them into word counts. Scikit-learn offers a `CountVectorizer` to do just this. It even detects `|` out of the box as token boundary! We only have to make sure that the NaNs are replaced by an empty string:

```
df['plot_keywords'] = df['plot_keywords'].fillna(' ')
```

Note that the transformation returns a `sparse matrix`, if we after want to view it in a DataFrame format again, it needs to be a dense matrix (or we have to do some special pandas loading procedure for sparse matrices). To restrict the words to frequent ones only (and our matrix not to blow up), we can use `max_features`. After, you can join this word matrix to your original DataFrame if you'd prefer, or even do something fancier like using the `TfidfVectorizer`. However, to keep further interpretation in this practical straight-forward, we'll not be doing this.

```
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer(max_features=500)

pd.DataFrame(cv.fit_transform(df['plot_keywords']).todense(),
             columns=sorted(cv.vocabulary_))
```

Dealing with Missing Values

We've discussed them a few times, and have so far just replaced them with some standard value (i.e. 0 or `' '`), but how do we correctly deal with them?

```
df['gross'].head(5)
```

First it's important to make a few considerations:

- Does it make sense to fill the missing values? In the case of text, what are you going to fill it with? It's easier to just fix the problem or ignore the document altogether.
- Are the missing values random? If it's just some small error, it's not going to influence your analysis that much to just get rid of the entries that have missing values. That is, given the fact that there's enough data.
- If the missing values are not random, you can choose a few strategies to fix them without hurting the overall vector representation too much:
  - `Impute`: replaces the missing values with some value inferred from the data (e.g. the mean/median feature value, or the majority category).
  - `Estimate`: replace the missing values with some value learned from the data (e.g. by using Singular Value Decomposition, K-NN or Naive Bayes).

Estimation often proves effective when doing predictions afterwards, even with up to half of values missing. However, using parametrized models also introduces more complexity in your pipeline, so it's worthwhile to consider if time is a factor, and if there's enough resources (in terms of data).

Imputing

Doing this column-wise in pandas requires a bit of effort for making compatible with scikit-learn's `Imputer`. First, we need to specify `axis=1` (to make it column-wise). Moreover, scikit-learn expects a matrix, so if we only want to impute one column, we need to wrap it between brackets, and after only select the 0th element. Like so:

```
from sklearn.preprocessing import Imputer
imp = Imputer(strategy='mean', axis=1)

df['gross'] = imp.fit_transform([df['gross']])[0]
```

As we can see, it's done:

```
df['gross'].head(5)
```

A less flexible, pure pandas implementation would be as follows:

```
df['gross'] = df['gross'].fillna(df['gross'].mean())
```

Dealing with Discrete (Categorical) Data

Sadly, there's no straight-forward way to deal with categorical data. If we want to use any scikit-learn model, we need the whole dataframe to be numeric, and have to remove all missing values. In turn, if we want to impute on most frequent categories, we have to count their occurrences, sort them by most frequent, take the top entry, and get its index value. As such:

```
df['genres'].value_counts()

# --- new cell ---

df['genres'].value_counts()[:1].index[0]
```

Now, to both impute numerical *and* categorical values, we need a small piece of code rather than a one-liner. Pandas has a 'category' type, that allows for easy conversion to numbers. We'll use that in combination with the two other `fillna` lines that we discussed before to fill the missing values of all features:

```
for column, dtype in df.dtypes.to_dict().items():  # for each column, dtype pair in the dataframe

    if dtype == 'object':  # if the column is an object (thus discrete)
        df[column] = df[column].fillna(df[column].value_counts()[:1].index[0])  # fill with most common
        cats = df[column].astype('category')  # convert to category
        df[column] = cats.cat.codes  # use the category indices to convert to numeric

    else:  # if the column is something else (thus numeric)
        df[column] = df[column].fillna(df[column].mean())  # take the mean
```

Please note that this is quite an advanced application — don't worry if it doesn't make sense code-wise.

Scaling

With all the missing values out of the way, we need to standardize our feature space. Most models benefit greatly from a Gaussian distribution, and therefore we can use scikit-learn's `StandardScaler` to achieve this. And while we're at it, let's immediately dump the whole thing in `X` and set our `y`:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
y = df.pop('quality')
X = scaler.fit_transform(df)
```

So are we finally done for predictions now? Not quite yet.

Evaluation Set-up

Now our DataFrame is ok, and we can apply our model to it. However, we first need to set up baselines and inspect their behaviour. Let's start by at least making our train / test set-up:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y)
```

Notice that because we're not doing any parameter tuning, we don't require a validation set yet. Simple classification models such as Naive Bayes or Logistic Regression (without polynomial features, kernel transformations, or parameter tuning) make good first baselines. And, of course, we'd also need a 'dumb' majority baseline. The latter we can construct like so:

```
ŷ_baseline = [y.value_counts()[:1].index[0]] * len(y)
```

Notice that this is the same 'most common class' as we used for the imputation of categories. We then create a list of the length of the initial label, filled with this value. That's our majority baseline. We can quickly evaluate its performance like so:

```
from sklearn.metrics import classification_report

print(classification_report(y, ŷ_baseline))
```

Now for the 'regular' baseline, let's use LogisticRegression:

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(X_train, y_train)

ŷ_test = lr.predict(X_test)
```

And evaluate:

```
from sklearn.metrics import classification_report

print(classification_report(y_test, ŷ_test))
```

Well, aren't we doing amazing? This is the point where some loud alarm bells need to start ringing. Why?

- We determined that most of the features are actually pretty useless; they don't accurately reflect the 'truth' or contain mixed information.
- The amount of information we have available is quite limited, and would intuitively not be enough to guess movie quality this well (although, of course this is speculation at this point).

- We ran a vanilla model, no tuning, no nothing, and we're doing almost perfect on the test set.

  Let's see what the model is paying attention to:

```
list(zip(df.columns[:-1], lr.coef_[2]))
```

LogisticRegression has coefficients per class, so this would be for class '3'.

Well, well, apparently `imdb_score` is an amazing predictor for quality. Let's see why:

```
pd.concat([df['imdb_score'], y], axis=1).plot(kind='scatter', x='imdb_score', y='quality')
```

Task 7

- What is happening here?
- Try removing the polluting feature with `del df['featurename']` and run the experiments again.

Solutions

Task 1:

- Not all. Some are floating point numbers (decimal numbers), whilst they can't have 'halves' like facenumber in poster.
- Try `df['facenumber_in_poster'].value_counts().plot(kind='bar')`.

Task 2:

- No, definitely not for mr. Ziff, who is a stuntman.
- This seems to be the Darcy Donavan facebook page, worth about 3.2M likes. That's not nearly the >600M that were suggested here. Matthew who? only has 54K likes on facebook.

Task 3:

- Anchorman should have had Will Ferrell as first actor. Darcy Donavan is actually last on the initial Cast List page.
- Can't be Alphabetically determined either, because then Christina Applegate would have definitely been higher.
- Looking at the likes, the IMDB sorting might also not be trusted, as Will Ferrell only has 11M likes, and not > 600M. So I guess it remains a mystery how these names have been determined.

Task 4:

- Not really, no.
- Due to this small sample, and the data mostly being biased towards more popular actors, there's no 'heavy tail' with many low frequencies.

Task 5:

- The "'most expensive movies"' are South Korean, Hungarian and Japanese movies.
- As it turns out, the `budget` feature for IMDB lists *local currency*. As a result, using this as a feature will probably be quite noisy (lot of inaccurate information).

- You could look up the country name and match some currency database to convert to dollars, but you'd have to have historic information.

Task 6:

- Documentaries (at least visually) look like the best bets for decent reviews.
- Seems like that a very particular crowd reviews documentaries between a particular range. Which—if you think about people who watch documentaries, and in which circumstances—it makes some sense.

Task 7:

- Hopefully, with this you will see that quality is actually based on `imdb_score`: 1-3 = 1, 3-5 = 2, 5-7 = 3, 7-8 = 4, 8-10 = 5. The model will see this, and use this feature. Given that it's just a derivation from `imdb_score`, we don't want to use it, as it is a case of feature contamination / pollution.
- You should get much lower scores, but the coefficients should make more sense.