Data Processing: A Practical Guide

The steps in the data science pipeline that need to be carried out to answer business questions are:

1. Data Understanding & Formulation of the Questions
2. Data Cleaning
3. Exploratory Data Analysis
4. Feature Engineering
5. Feature Selection
6. Data Modelling

The file data_processing.py compiles the most important tools I use for the steps 2-5, following the 80/20 Pareto principle. Additionally, in the following, some practical guidelines are summarized very schematically.

Notes:

- This guide assumes familiarity with `python`, `numpy`, `pandas`, `matplotlib`, `seaborn`, `sklearn` and `scipy`, among others.
- Additionally, I presume you are acquainted with machine learning and data science concepts.
- Finally, mainly **tabular data** is considered; however, very basic natural text processing (MLP) approaches are introduced in the feature engineering section.

For more information on the motivation of the guide, see my blog post.

Table of Contents

General

- Watch at the returned types:

    - If it is a collection or a container, convert it to a `list()`.
    - If it is an array/tuple with one item, access it with `[0]`.
- Data frames and series can be sorted: `sort_values(by, ascending=False)`.
- Recall we can use handy python data structures:
    - `set()`: sets of unique elements.
    - `Counter()`: dict subclass for counting hashable objects.
- Use `np.log1p()` in case you have `x=0`; `log1p(x) = log(x+1)`.
- Use `df.apply()` extensively!
- Make a copy of the dataset if you drop or change variables: `data = df.copy()`.
- All categorical variables must be enconded as quantitative variables somehow.
- Seaborn plots get `plt.figure(figsize=(10,10))` beforehand; pandas plots get `figsize` as argument.
- `plt.show()` only in scripts!
- Use a seed whenever there is a random number generation to ensure reproducibility!
- Pandas slicing:
    - `df[]` should access only to column names/labels: `df['col_name']`.
    - `df.iloc[]` can access only to row & column index numbers + booleans: `df.iloc[0,'col_name']`, `df.iloc[:,'col_name']`.
    - `df.loc[]` can access only to row & column labels/names + booleans: `df.loc['a','col_name']`, `df.loc[:,'col_name']`.
    - `df[]` can be used for changing entire column values, but `df.loc[]` or `df.iloc[]` should be used for changing sliced row values.
- We can always save any python object as a serialized file using `pickle`; for instance: models or pipelines. But: python versions must be consistent when saving and loading.

Data Cleaning

- Always do general checks: `df.head()`, `df.info()`, `df.describe()`, `df.shape`.
- Always get lists of column types: `select_dtypes()`.
    - Categorical columns, `object`: `unique()`, `nunique()`, `value_counts()`. Can be subdivided in:
        - `str`: text or category levels.
        - `datetime`: encode with `to_datetime()`.
    - Numerical columns, `int`, `float`: `describe()`.
- Detect and remove duplicates: `duplicated()`, `drop_duplicates()`.
- Correct inconsistent text/typos in labels, use clear names: `replace()`, `map()`.
- Beware: many dataframe modifying operations require `inplace=True` flag to change the dataframe.
- Detect and fix missing data:
    - `isnull() == isna()`.
    - Plot the missing amounts: `df.isnull().sum().sort_values(ascending=False)[:10].plot(kind='bar')`.
    - Analyze the effect of missing values on the target: take a feature and compute the target mean & std. for two groups: missing feature, non-missing feature.
        - This could be combined with a T-test.
    - Sometimes the missing field is information:
        - It means there is no object for the field; e.g., `license_povided`: if no string, we understand there is no license.

        - We can create a category level like `'Missing'`
        - We can mask the data: create a category for missing values, in case it leads to insights.
    - `dropna()` rows if several fields missing or missing field is key (e.g., target).
    - `drop()` columns if many (> 20-30%) values are missing.
    - Impute the missing field/column with `fillna()` if few (< 10%) rows missing: `mean()`, `median()`, `mode()`.
    - More advanced:
        - Predict values with a model.
        - Use k-NN to impute values of similar data-points.
- Detect and handle outliers:
    - Linear models are shifted towards the outliers!
    - Keep in mind the empirical rule of 68-95-99.7 (1-2-3 std. dev.).
    - Compute `stats.zscore()` to check how many std. deviations from the mean; often Z > 3 is considered an outlier.
    - Histogram plots: `sns.histplot()`.
    - Box plots: `sns.boxplot()`.
    - Scatterplots: `plt.scatter()`, `plt.plot()`.
    - Residual plots: differences between the real/actual target values and the model predictions.
    - IQR calculation: use `np.percentile()`.
    - Drop outliers? Only if we think they are not representative.
    - Do transformations fix the `skew()`? `np.log()`, `np.sqrt()`, `stats.boxcox()`, `stats.yeojohnson()`.
        - Usually a absolute skewness larger than 0.75 requires a transformation (feature engineering).
- Temporal data / dates or date-time: they need to be converted with `to_datetime()` and the we need to compute the time (in days, months, years) to a reference date (e.g., today).

Exploratory Data Analysis

- Recall we have 3 main ways of plotting:
    - Matplotlib: default: `plt.hist()`.
    - Seaborn: nicer, higher interface: `sns.histplot()`.
    - Pandas built-in: practical: `df.plot(kind='hist')`, `df.hist()`, `df.plot.hist()`; `plt` settings passed as arguments!
- Usually, the exploration is done plotting the independent variables (features) against the target (dependent or predicted variable).
- We need to plot / observe **every** feature or variable:
    - Create automatic lists of variable types: `select_dtypes()`. Usual types:
        - Numerical: `int`, `float`.
        - Strings: `object`. Can contain:
            - `str`: text or category level.
            - `datetime`: encode with `to_datetime()`.
    - Automatically created lists often need to be manually processed, especially `object` types.
    - Loop each type list and apply the plots/tools we require.
- Quantitative/numerical variables: can be uni/multi-variate; most common EDA tools:
    - Histograms: `sns.histplot()`, `plt.hist()`, `df.hist()`.
        - Look at: shape, center, spread, outliers.

    - Numerical summaries: `describe()`.
    - Boxplots: `sns.boxplot()`.
        - Look at outliers.
        - Also, **combine these two**:
            - Boxplot: `sns.catplot()`.
            - Points overlapped: `sns.stripplot()`.
    - Scatteplots: `sns.scatterplot()`, `plt.scatter()`, `sns.regplot()`, `sns.lmplot()`
        - Look at: linear/quadratic relationship, positive/negative relationship, strength: weak/moderate/strong.
        - Beware of the Simpson's Paradox.
    - Correlations: `df.corr()`, `stats.pearsonr()`; see below.
- Categorical variables: ordinal (groups have ranking) / cardinal (no order in groups); most common EDA tools:
    - Count values: `unique()`, `nunique()`, `value_counts().sort_values(ascending=False).plot(kind='bar')`.
    - Frequency tables; see below.
    - Bar charts: `sns.barplot()`, `plt.bar()`, `plt.barh()`; use `sort_values()`.
    - Count plots: `sns.countplot()`.
- If a continuous variable has different behaviors in different ranges, consider stratifying it with `pd.cut()`. Example: `age -> age groups`.
- Frequency tables: stratify if necessary, group by categorical levels, count and compute frequencies.
    - Recipe: `groupby()`, `value_counts()`, normalize with `apply()`.
    - See also: `pd.crosstab()`.
- Correlations:
    - Heatmap for all numerical variables: `sns.heatmap(df.corr())`.
        - `cmap`: Matplotlib colormaps.
        - Seaborn color palettes.
    - Bar chart for correlations wrt. target: `df.corr()['target'].sort_values(ascending=True).plot(kind='bar')`.
    - Pair correlations: `stats.pearsonr(df['x'],df['y'])`.
    - Check if there is multicolinearity (see section on Feature Selection): it's not good.
    - Beware of confounding: two correlated variables can be affected/related by something different.
- If you want to try different or plots:
    - Historgrams: add density, `kde=True` in `sns.histplot()`.
    - Boxplots -> try `sns.swarmplot()`: boxes are replaced by point swarms.
    - Boxplots -> try `sns.violinplot()`: box width changed.
    - Scatterplots -> try `sns.lmplot()` or `sns.reglot()`: linear regression is added to the scatterplot.
    - Scatterplots -> try `sns.jointplot()`: density distribution iso-lines of a pair of quantitative variables.
- We can get a handle of any plot and set properties to is later: `bp = sns.boxplot(); bp.set_xlabel()`.
- Larger group plots:
    - `sns.pairplot()`: scatterplots/histograms of quantitative variables in a matrix; select variables if many.

  - `sns.FacetGrid()`: create a grid according to classes and map plot types and variables.

Feature Engineering

- Always make a copy of the dataset if we change it: `df.copy()`.
- Transformations:
  - General notes:
    - Apply if variables have a `skew()` larger than 0.75; we can also perform normality checks: `scipy.stats.mstats.normaltest`.
    - Try first how well work simple functions: `np.log()`, `np.log1p()`, `np.sqrt()`.
    - If `hist()` / `skew()` / `normalitytest()` don't look good, try power transformations, but remember saving their parameters and make sure we have an easi inverse function:
      - Box-Cox: generalized power transformation which usually requires `x > 0`: `boxcox = (x^lambda + 1)/lambda`.
      - Yeo-Johnson: more sophisticated, piece-wise - better results, but more difficult to invert & interpret.
      - We can encode/transform single columns! Just pass the single column to the encoder/transformer.
  - Target: although it is not necessary for it to be normal, normal targets yield better R2 values.
    - Often the logarithm is applied: `df[col] = df[col].apply(np.log1p)`.
    - That makes undoing the transformation very easy: `np.exp(y_pred)`.
    - However, check if power transformations are better suited (e.g., `boxcox`, `yeojohnson`); if we use them we need to save the params/transformer and make sure we know how to invert the transformation!
  - Predictor / independent variables:
    - `scipy` or `sklearn` can be used for power transformations, e.g., `boxcox`, `yeojohnson`.
    - We can discretize very skewed variables, i.e., we convert then into categorical: we transform the distributions into histograms in which bins are defined as equal width/frequency. That way, each value is assigned the bin number. Try `pd.cut()`. Additionally, see binarization below.
- Extract / create new features, more descriptive:
  - Multiply different features if we suspect there might be an interaction.
  - Divide different features, if the division has a meaning.
  - Create categorical data from continuous if that has a meaning, e.g., daytime.
  - Try polynomial features: `PolynomialFeatures()`:
    - I think it makes most sense using `PolynomialFeatures()` with continuous variables, but I might be wrong.
    - `PolynomialFeatures()` considering dummy variables could make sense to analyze interactions, but I don't see that very clearly.
    - Apply `PolynomialFeatures()` as one of the last steps before scaling to avoid complexity in the data processing.
  - Create deviation factors from the mean of a numeric variable in groups or categories of another categorical variable.
- Measure the cardinality of the categorical variables: how many categories they have.
  - `data[cat_vars].nunique().sort_values(ascending=False).plot.bar(figsize=(12,5))`.
  - Tree-based models overfit if we have

        - many categories,
        - rare labels.
    - Replace categorical levels with few counts (rare) with `'other'`.
- Categorical feature encoding:
    - Note: we can encode single columns! Just pass the single column to the encoder.
    - One-hot encoding / dummy variables: `get_dummies()`; use `drop_first=True` to avoid multi-colinearity issues! (see Feature Selection Section)
        - Alternative: `sklearn.preprocessing.OneHotEncoder`.
        - In general, `sklearn` encoders are objects that can be saved and have attributes and methods: `classes_`, `transform()`, `inverse_transform()`, etc.
        - Use `pd.Categorical()` if we want to dummify integers; for strings or `np.object` this should not be necessary.
    - Binarization: manually with `apply()`, `np.where()` or `sklearn.preprocessing.LabelBinarizer`.
        - Usually very skewed variables are binarized.
            - We can check the predictive strength of binarized variables with bar plots and T tests: we binarize and compute the mean & std. of the target according to the binary groups.
        - `LabelBinarizer` one-hot encodes classes defined as integers: `fit([1, 2, 6, 4, 2]) -> transform([1, 6]): [[1, 0, 0, 0], [0, 0, 0, 1]]`.
        - Sometimes it is necessary to apply this kind of multi-class binarization to the **target**.
    - Also useful for the **target**, `LabelEncoder`: converts class strings into integers, necessary for target values in multi-class classification.
        - `fit(["paris", "paris", "tokyo", "amsterdam"]) -> transform(["tokyo", "paris"]): [2, 1]`.
        - It is common to use `LabelEncoder` first and then `LabelBinarizer`.
    - Ordinal encoding: convert ordinal categories to `0,1,2,...`; but be careful: we're assuming the distance from one level to the next is the same -- Is it really so? If not, consider applying one-hot encoding?
        - `sklearn` tools: `OrdinalEncoder`, `DictVectorizer`.
        - `OrdinalEncoder` works with strings and integers, but don't mix them.
        - In all these encoders in which a category is mapped to a value, we need to consider the possibility of having new categories in the test split; to avoid problems, use `handle_unknown`: `OrdinalEncoder(handle_unknown='ignore')`
        - If the ordinal sequence order is not clear, provide it: `categories = ['high school', 'uni']; OrdinalEncoder(categories=categories)`
    - If a distribution of a numerical variable is very skewed or irregular, we can discretize it with `pd.cut()`.
    - There are many more tools: [Preprocessing categorical features]
    - Weight of evidence: If the target is binary and we want to encode categorical features, we can store the target ratios associated to each feature category level.
- Train/test split: perform it before scaling the variables: `train_test_split()`. ALWAYS use the seed for reproducibility!
    - If we have a classification problem and we want to maintain the class ratios in the splits, we can use the parameter `stratify=y`; `y` needs to be `LabelEncoded`.
    - `ShuffleSplit`: if several train-test splits are required, not just one.

- `StratifiedShuffleSplit`: same as before, but when we have imbalanced datasets and we'd like to maintain the label/class ratios in each split to avoid introducing bias; this a more advanced way than using the `stratify` parameter.
- Feature scaling: apply it if data-point distances are used in the model or parameter sizes matter (regularization); **apply it always as the last feature engineering step and fit the scaler only with the train split!**
  - `StandardScaler()`: subtract the mean and divide by the standard deviation; features are converted to standard normal variables (e.g., if normally distributed, 99% of the values will be in a range of `[-3,3]`). Note that if dummy variables `[0,1]` passed, they are scaled, too. That should not be an issue, but the interpretation is not as intuitive later on. Alternatives: use `MinMaxScaler()` or do not pass dummies to the scaler.
  - `MinMaxScaler()`: a mapping with which the minimum value becomes 0, max becomes 1. This is sensitive to outliers! However, if we remove the outliers and the scaling is applied to the whole dataset including dummy variables, it can be a good option.
  - `RobustScaler()`: IQR range is mapped to `[0,1]`, i.e., percentiles `25%, 75%`; thus, the scaled values go out from the `[0,1]` range.

Scikit-Learn Transformers

List of the most important Scikit-Learn Transformers:

- Missing data imputation
  - `sklearn.impute.SimpleImputer`: we define what is a missing value and specify a strategy for imputing it, e.g., replace with the mean.
  - `sklearn.impute.IterativeImputer`: features with missing values are modeled with the other features, e.g., a regression model is built to predict the missing values.
- Categorical Variable Encoding
  - `sklearn.preprocessing.OneHotEncoder`: dummy variables of all levels (except one) in a categorical variable are created, i.e., a binary variable for each category-level. **The advantage of this again the pandas `get_dummies()` is that OneHotEncoder returns a sparse matrix, which is much more memory efficient in datasets with many features.**
  - `sklearn.preprocessing.OrdinalEncoder`: string variables are converted into ordered integers; however, be careful, because these cannot be used in scikit-learn if they do not really represent continuous variables... if that is not the case, try the `OneHotEncoder` instead.
- Scalers
  - `sklearn.preprocessing.MinMaxScaler`: data mapped to the min-max range
  - `sklearn.preprocessing.StandardScaler`: subtract mean and divide by standard deviation
  - `sklearn.preprocessing.RobustScaler`: scaling with the IQR done
  - ...
- Discretisation
  - `sklearn.preprocessing.KBinsDiscretizer`: quantization, partition of continuous variables into discrete values; different strategies available: constant-width bins (uniform), according to quantiles, etc.
- Variable Transformation
  - `sklearn.preprocessing.PowerTransformer`: Yeo-Johnson, Box-Cox

    - `sklearn.preprocessing.FunctionTransformer`: It constructs a transformer from an arbitrary callable function! That's very useful! An example is shown in my repository ml_pipeline_rental_prices/`train_random_forest/run.py`.
    - ...
- Variable Combination
    - `sklearn.preprocessing.PolynomialFeatures`: given a degree d, obtain polynomial features up to the degree: x_1, x_2, d=2 -> x_1, x_2, x_1*x_2, x_1^2, x_2^2
- Text Vectorization
    - `sklearn.feature_extraction.text.CountVectorizer`: create a vocabulary of the corpus and populate the document-term matrix `document x word` with count values.
    - `sklearn.feature_extraction.text.TfidfTransformer`: create the document-term matrix by scaling with in-document an in-corpus frequencies.

Creation of Transformer Classes

Manual definition:

```python
# Parent class: its methods & attributes are inherited
class TransformerMixin:
    def fit_transform(self, X, y=None):
        X = self.fit(X, y).transform(X)
        return X

# Child class
# Same class definition as before
# BUT now we inherit the methods and attributes
# from TransformerMixin
class MeanImputer(TransformerMixin):
    def __init__(self, variables):
        self.variables = variables
    def fit(self, X, y=None):
        self.imputer_dict_ = X[self.variables].mean().to_dict()
        return self
    def transform(self, X):
        for v in self.variables:
            X[v] = X[v].fillna(self.imputer_dict[v])
        return X

# Usage
my_imputer = MeanImputer(variables=['age','fare'])
my_imputer.fit(X) # means computed and saved as a dictionary
X_transformed = my_imputer(X) # We get the transformed X: mean imputed in
NA cells
X_transformed = my_imputer.fit_transform(X)
my_imputer.variables # ['age','fare']
my_imputer.imputer_dict_ # {'age': 39, 'fare': 100}
```

Classes derived from `sklearn`; they have the advantage that they can be stacked in a `Pipeline`:

```python
import numpy as np
import pandas as pd

from sklearn.base import BaseEstimator, TransformerMixin

class MeanImputer(BaseEstimator, TransformerMixin):
    """Numerical missing value imputer."""

    def __init__(self, variables):
        # Check that the variables are of type list
        if not isinstance(variables, list):
            raise ValueError('variables should be a list')
        self.variables = variables

    def fit(self, X, y=None):
        # Learn and persist mean values in a dictionary
        self.imputer_dict_ = X[self.variables].mean().to_dict()
        return self

    def transform(self, X):
        # Note that we copy X to avoid changing the original dataset
        X = X.copy()
        for feature in self.variables:
            X[feature].fillna(self.imputer_dict_[feature], inplace=True)
        return X
```

Natural Languange Processing (NLP): Extracting Text Features with Bags of Words

Natural Language Processing is a completely separate topic, as are Image Processing or Computer Vision. If we'd like to model natural language texts as sequences of words, the most effective approaches are (1) Recurrent Neural Networks (RNN) or (2) Transformers. My following repositories show examples about how we can use Recurrent Neural Networks with text:

- text_sentiment
- text_generator

However, in simple cases in which we'd like to vectorize short texts that are embedded in larger tabular datasets, we can use **bags of words**. Let's say we have a corpus of many documents of a similar type (e.g., articles, reviews, etc.); each document is a text. Then, we do the following:

- We tokenize (and maybe stem/lemmatize) all the words in the corpus. A more detailed description of these steps is given here.
- We create a vocabulary with all the unique words.
- We create a **document-term matrix (DTM)**, with
  - rows: documents
  - columns: tokens from vocabulary
  - cell content: presence/count/frequency of word in document

That DTM is our `X` and it can be used to perform supervised (e.g., classification, if we have document labels) or unsupervised learning (e.g., topic discovery).

The cell contents can be

1. word presence: whether a word appears (1) or not (0) in the document: `CountVectorizer`.
2. word count: how many times a word appears in the document: `CountVectorizer`.
3. word frequency: **term frequency inverse document frequency (TF-IDF)** values: `TfidfVectorizer`.

The **term frequency inverse document frequency (TF-IDF)** consists in multiplying the count of that term in the document by the how rare that term is throughout all the documents we are looking at. That way, common but meaningless words (e.g., 'the', 'of', etc.) have a lower value.

Example

Two documents: *"We like dogs and cats"*; *"We like cars and planes"*

`CountVectorizer()` yields:

doc	We	like	and	dogs	cats	cars	planes
0	1	1	1	1	1	0	0
1	1	1	1	0	0	1	1

`TfidfVectorizer()` would yield:

doc	We	like	and	dogs	cats	cars	planes
0	1	1	1	1.6931	1.6931	0	0
1	1	1	1	0	0	1.6931	1.6931

The TF-IDF matrix would contain the following values for each document `d` and term `t` cell:

`idf(d,t) = ln((N/|d in D in which t in d|) + 1)` `tfidf(d,t) = C(d,t) * idf(d,t)`

With:

- `N`: total number of documents in the corpus, `|D|`
- `|d in D in which t in d|`: number of documents in which the term `t` appears
- `C(d,t)`: how many times the term `t` appears in document `d`

However, note that `TfidfVectorizer()` additionally normalizes each row to have length 1.

Feature Selection

- Less features prevent overfitting and are easier to explain.
- Remove features which almost always (99% of the time) have the same value.
- There are three major approaches for feature selection:
    1. The effect of each variable is analyzed on the target using ANOVA or similar.
    2. Greedy approaches: all possible feature combinations are tested (very expensive).
    3. Lasso regularization: L1 regularization forces coefficients of less important variables to become 0; thus, we can remove them. This is the method I have normally used.

- Multi-colinearity: detect and avoid it! In models such as linear regression the coefficients or parameters denote the effect of increasing one unit value of the associated feature while the *rest of the parameters is fixed*. If there are strong correlations (i.e., multi-colinearity), that is not true, since the parameter values are related, they move together. Additionally, in regression models X^TX might become non-invertible, because the variables are not independent enough.
  - We can detect multi-colinearity with heatmaps of correlations `sns.heatmap(df.corr())`; if we have too many variables to visualize, take the upper triangle of the correlation matrix with `np.tril_indices_from()` and `stack()` them into a multi-indexed dataframe. Then, `query()` the pairs with high absolute correlation values.
  - A more formal way of detecting multi-colinearity consists in computing the R^2 of each predictor fitted against the rest of the predictors; if any variable can be predicted with R^2 > 0.8, it is introducing multi-colinearity. This is related to the Variable Inflation Factor: VIF = 1 / (1-R^2); if VIF > 5, i.e., R^2 > 0.8, we have multi-colinearity.
  - Consider removing variables that are strongly correlated with other variables.
- We can measure sparsity of information with `PCA()`; if less variables explain most of the variance, we could drop some.
- Typical method: Select variables with L1 regularized regression (lasso): `SelectFromModel(Lasso())`.
- Use `sns.pairplot()` or similar to check multi-colinearity; correlated features are not good.
- If the model is overfitting, consider dropping features.
  - Ovefitting: when performance metric is considerably better with train split than in cross-validation/test split.

Hypothesis Tests

- Always define `H0`, `Ha` and `alpha` beforehand and keep in mind the errors:
  - Type I error: you're healthy but the test says you're sick: False positive.
  - Type II error: your're sick, but the test says you're healthy: False negative.
  - P(Type I error) = `alpha`, significance level, typically 0.05.
  - P(Type II error) = `beta`.
  - Power of a test = 1 - `beta`: depends on `alpha`, sample size, effect size.
    - We can estimate it with a **power analysis**.
- Most common hypothesis tests
  - Two independent proportions: Z Test (Standard Distribution).
  - Two independent means: T Test (Stundent's T Distribution).
  - One factor with L>2 levels creating L means: One-way ANOVA (Fisher's F Distribution).
  - One factor with L>2 levels creating L proportions/frequencies: Chi Square Test with contingency table.
    - Example contingency table: `[age_group, smoke] = (18-30, 31-50, 51-70, 71-100) x (yes, no)`.
    - Use `crosstab.
- Take into account all the assumptions made by each test and the details!
  - Independent vs. paired groups (repeated measures; within-studies).
  - One-sided (H: >, <) vs 2-sided tests (H: !=).
  - Normality assumption: check histograms, QQ plots, run normality tests if necessary.
  - Equal variances assumption.
  - If assumptions broken, consider equivalent parametric tests.

- Post-hoc tests when >2 levels (e.g., after ANOVA): apply Bonferroni correction if T tests used: `alpha <- alpha / num_tests`.

Data Modeling and Evaluation (Supervised Learning)

Data modeling is out of the scope of this guide, because the goal is to focus on the data processing and analysis part prior to creating models. However, some basic modeling steps are compiled, since they often provide feedback for new iterations in the data processing.

- Most common approaches to start with tabular data (supervised learning):
  - Regression (focusing on interpretability): `Ridge` (homogeneous coeffs.), `Lasso` (feature selection), , `ElasticNet` (`Ridge + Lasso`), `RandomForestRegressor`.
  - Classification: **note: in `sklearn` classification models have a regression class, too!**
    - `LogisticRegression` (use `penalty` for regularization): not only for binary classification in Scikit-Learn; nice log-odds interpretation of coefficients.
    - `KNearestNeighbors`
      - Easy interpretation of similar points, but complete dataset is used for inference, thus small and scaled datasets should be used (e.g., `10000 x 50`).
      - Use the elbow method to deduce the optimum `K`: try different `K` values in a loop and register the error metric with the test/validation split; then take the `K` of the best metric.
    - Support Vector Machines (SVMs): `sklearn.svm.SVC`, `LinearSVC`
      - Usually non-linear SVMs are used with a kernel for medium datasets, e.g., with Gaussian kernels (`rbf`)
        - Example: `rbfSVC = SVC(kernel='rbf', gamma=1.0, C=10.0)`
        - `gamma` and `C` need to be chosen; the smaller they are, the larger the regularization (less complex model)
        - `gamma ~ 1/sigma^2`
        - `C = 1 / lambda`
      - However, non-linear SVMs are time-intensive as the number of features and data-points increase, because kernels are used based on the similarities of a point to all the landmarks in the dataset. Alternative: use approximate kernels with sampling and linear SVMs
      - Approximate kernels such as `Nystroem` and `RBFSampler` are used to transform the `X` with sampled approximative kernels; then, we use a `LinearSVC` or `SGDClassifier` with the transformed dataset; that is much faster for large datasets with many features.
      - Use `GridSearchCV` to detect the optimum hyperparameters.
    - `DecisionTreeClassifier`
      - Decision trees are nice to interpret when we plot the trees
      - However, they overfit the dataset if no constraints are set to the tree; therefore, the usual approach is:
        - Create a tree with overfitting and extract its parameters: `max_depth`, etc.
        - Run a `GridSearchCV` with the parameter ranges given by the maximum values of the overfit tree to perform hyperparmeter tuning.
        - We can perform regressions with `DecisionTreeRegressor`: a mean value for each leaf node is computed

        - Ensemble Methods: they beat usually any other method with tabular data.
            - Most common ensemble methods:
                - Bagging = bootstrapped aggregating: several independent trees are fitted using samples with replacement; since independent, parallelized.
                    - `BaggingClassifier`: overfitting can occur because trees are correlated (due to sampling with replacement).
                    - `RandomForestClassifier`: max number of features selected randomly, which decreases correlation; thus, no overfitting.
                - Boosting: complete dataset used in successive weak or simple base learners which improve by penalizing residuals (miss-classified points). Since we're improving the previous models, we risk overfitting, thus we need to do grid search.
                    - `AdaBoostClassifier`: we can select our weak learner model; the loss is exponential.
                    - `GradientBoostingClassifier`: weak learners are trees; the loss is not as steep and it performs better with outliers.
                - Voting / Stacking: we combine several models and the final classification is a hard or soft average (i.e., majority of classes or average probability). Use them only if they are significantly better than the single models, because they introduce complexity, i.e., they are more difficult to handle and they overfit. So, if we use them, apply grid search!
                    - `VotingClassifier`: voting is done.
                    - `StackingClassifier`: a classifier is appended to give the final result.
            - `AdaBoostClassifier` can take different base learners, not only trees.
            - `RandomForestClassifier` forests do not overfit with more learners/trees, the performance plateaus; they are fast, because the trees are independently trained.
            - `GradientBoostingClassifier` is better than `AdaBoostClassifier`, but it takes longer to train it; try the `xgboost` library instead of `sklearn`.
            - Boosting overfits, thus, perform always a grid search!
            - **Advice: stick to `RandomForestClassifier` or `GradientBoostingClassifier`with grid search; also, try `xgboost`.**
- Always evaluate with a test split that never was exposed to the model
    - Regression: R2, RMSE.
    - Classification: confusion matrix, accuracy, precision, recall, F1, ROC curve (AUC), Precision-Recall curve (for unbalanced classes).
        - Confusion matrix: `Real (Positive, Negative) x Predicted (Positive, Negative)`
        - Accuracy (bad metric, unless equal number of class instances): diagonal / all 4 cells = `(TP + TN) / (TP + FP + FN + TN)`
        - **Precision** (of Predicted Positives) of each class = `TP / (TP + FP)`
            - Interpretation: do we wan to assure that our predicted positives are correct? **How good are we at minimizing Type I error?**
        - **Recall** or Sensitivity (wrt. Real Positives) of each class = `TP / (TP + FN)`
            - Interpretation: do we want to capture all the true positives? **How good are we at minimizing Type II error?**

        - It is the most important metric in disease or fraud detection.
      - Specificity of each class: Precision for Negatives = `TN / (FP + TN)`
      - F1: harmonic mean between precision and recall; it is a nice trade-off between precision and recall, thus, a metric which is recommend by default: `F1 = 2 * (Precision*Recall)/(Precision+Recall)`
      - Matthews Correlation Coefficient (MCC)
      - ROC curve (binary classifications)
        - True positive rate = Sensitivity = Recall
        - False positive rate = 1 - Specificity
        - AUC: Area Under the Curve: 0.5 random guess (triangle) - 1 perfect model (square)
      - Alternative to ROC: Precision-Recall curve; usually descending, better suited than ROC for unbalanced datasets.
- Classification problems / models:
  - Always maintain class ratios in different splits with stratified groupings: `stratify`, `StratifiedShuffleSplit`
  - Usually all classification models can be used as regression models!
  - Binary classifiers are already generalized to be multi-class classifiers. Depending on the solver, we have a *one-vs-rest* approach or a *multinomial* approach.
  - A nice example of how to stack the results of several multi-class problems: 03_Classification_IBM.md `/ 1.9 Python Lab: Human Activity`.
  - Always check which classes get mixed in the confusion matrix: Why are they similar? How could we differentiate them? Do we need more data?
  - For unbalanced datasets, the Precision-Recall curve is a good metric.
  - Decision boundaries can be plotted; look at `plot_decision_boundary()` in `data_processing.py`.
- Unbalanced datasets (specially important for classification problems):
  - Accuracy is often not a good metric; in particular, accuracy is a bad metric for unbalanced datasets.
  - Precision measures how bad the Type I error is.
  - Recall measures how bad the Type II error is.
  - In each business case (e.g., illness detection, fraud detection), we need to choose the cost of each type of error: Type I or II; then, we select the right metric.
  - Techniques to deal with unbalanced datasets; first, measure the class distribution (`value_counts()`) and the metric to improve, e.g. recall. Then, apply:
    - Weights: use weights which are inverse to the ratio of the class; however, we can try several weight values, i.e., we can treat them as hyperparameters to be tuned. Weights are used in the loss computation.
      - Weights can be passed in model instantiation or
      - in the `fit` method
    - Resampling:
      - Oversampling minority class; e.g. SMOTE: create points in between a minority point and its nearest neighbors
      - Undersampling majority class; e.g., randomly remove majority points.
  - **Important note: there is no single best approach for all cases!** We need to:
    - Choose metric to improve.

        - Compute metric with original dataset.
        - Try different techniques (resampling, weights) and check metric.
        - Select the best technique.
- Bias-Variance trade-off: always consider it!
    - Model error = bias + variance + irreducible randomness.
    - Bias error: more simplistic model (e.g., less features), more relaxed model - **underfitting**.
    - Variance error: more complex model (e.g., more features), model more tightly adjusted to the data-points, thus less generalizable - **overfitting**.
    - We need to find the sweet spot: since more bias means less viariance, and vice-versa, the sweet spot is the minimum sum of both:
        - Perform cross-validation: see how the validation split error changes as we change the model hyperparameters.
        - If train/validation/test error high: we're probably underfitting: add features, e.g., `PolynomialFeatures`.
        - If validation error high: we are probably overfitting; solution: regularization (`Ridge`, `Lasso`, etc.), drop features / reduce dimensions, etc.
- **Cross-validation** and **hyperparameter tuning**:
    - Perform a `train_test_split`, take `X_train` for cross-validation, `X_test` for final model performance
    - Define a k-fold split: non-overlapping validation splits within the `train` subset
        - Regression: `KFold`
        - Classification: `StratifiedKFold` to keep class ratios and avoid bias
    - Define a `Pipeline` with the models from which parameters need to be found/optimized (see `Pipelines` below).
    - Instantiate and fit `GridSearchCV` with the parameters to be optimized.
        - Define correct parameter grid.
        - Define `cv = k`; `k`: how many different exclusive validation splits.
        - Define correct scoring; beware that not all scores are for multi-class; use one-versus-rest for mult-class, e.g., `roc_auc_ovr`.
    - **Alternative**: use models with built-in cross-validation: `RidgeCV`, `LassoCV`, `ElasticNetCV`.
- Model interpretation: see 03_Classification_IBM.md
    - If the model is interpretable (linear, trees, KNN), plot model parameters to understand what's going on; often it's better than the predictions:
        - `importance = pd.DataFrame(model.coef_.ravel())` and `sort_values()`.
        - `model.feature_importances_`
        - However, note that in addition to coefficient magnitude (related to importance), we should have a look at the significances, i.e., p-values. Scikit-Learn doesn't have that, but `statsmodels` does.
    - If the model is not that interpretable (SVM, NNs, Random forests, Gradient Boosted Trees), we can try:
        - Global surrogate models: build twin but interpretable models that achieve similar enough errors and analyze their parameters.
        - Local surrogate models: pick one *local* row to analyze, create a synthetic dataset with similarity weights and train a surrogate model with it which can be interpretable.
        - Model Agnostic Explanations:

        - Permutation feature importance: selected feature values are shuffled and difference between predicted targets is evaluated
        - Partial Dependency Plots (PDP): all values of a feature are ranged from its minimum to its maximum and the average outcome is computed and plotted.
- If text reports need to be saved, convert them to figures: `plt.text()`.
- In production, avoid leaving default parameters to models, because defaults can change. Instead, save parameters in YAML files and load them as dicts; we can pass them to models at instantiation! Of course, dict key-values must be as defined in the model class.

Dataset Structure: Unsupervised Learning

Unsupervised learning is out of the scope of this guide, because the goal is to focus on the data processing and analysis part. However, some basic techniques related to dataset structure identification are compiled, since they can be very helpful while performing EDA or feature engineering. All in all, unsupervised learning techniques for tabular data can be classified as (1) **clustering** techniques and (2) **dimensionality reduction** techniques for either improving modeling or visualizing the dataset. In both cases, **distance metrics** are fundamental.

- Distance metrics
    - Euclidean distance, L2: `d(x,y) = srqt(sum((x-y)^2))`
    - Mahattan distance, L1: `d(x,y) = sum(abs(x-y))`; often used in cases with very high dimensionality, because distance values between point pairs become more unique than L2.
    - Cosine: `d = 1 - cos(x,y)`
        - It measures the angle between vectors.
        - Always check formula in documentation, because it is distinguished between *cosine similarity* and *cosine distance*.
        - Better for data that can contain many similar vectors, e.g., text: one text can be the summary of another so the vector end-points are relatively far from each other, but they are aligned! In other words, the location/occurrence/coordinate of the end-point is not important.
    - Jaccard: another distance measure used also with text or, in general, with sets of sequences: it measures the intersection over union of unique words/items (i.e., sets) that appear in two texts/sequences.
        - Ideal for **categorical features**!
        - Check formula, since there are differences between *Jaccard similarity* and *Jaccard distance*.
    - Average vs. pairwise distances:
        - Average distance: given 2 X & Y datasets with rows of feature vectors, the distances between each x in X to each y in Y are computed, and then, the average.
        - Pairwise distance: given 2 X & Y datasets with rows of feature vectors, the distance between paired rows is computed, and then, the average.
- Clustering: find `k` centroids `C_k` and assign each `x_i` to one `C_k`
    - Evaluation Metrics
        - `inertia_k = sum(i = 1:n; (x_i - C_k)^2)`
        - `distortion_k = (1/n) * inertia_k = (1/n) sum(i = 1:n; (x_i - C_k)^2)`
        - If we want clusters with similar numbers of points, use distortion.

- `KMeans`:
    - Main parameter: number of clusters, `k`
    - Use `kmeans++` for better initialization: initial centroids are chosen far from each other.
    - Elbow method: fit clusterings with different `k` and take the one from which metric (e.g., inertia) doesn't improve significantly
- Gaussian Mixtures Model, `GaussianMixture`: Gaussian blobs are placed on detected clusters
    - K-means is a *hard* clustering algorithm: it says whether a point belongs to a cluster or not.
        - GMM is a *soft* clustering algorithm: it tells the probability of a point of belonging to different clusters.
        - It is more informative to have a probability, since we can decide to take different clusters for a point depending on the business case.
        - Another application: **Anomaly Detection**; since we get probabilities of cluster belonging, a data point can be classified as outlier if all its probabilities are below a threshold!
    - Hierarchical Agglomerative Clustering, `AgglomerativeClustering`: iteratively find the two closest items in our dataset and cluster them together; an item can be (i) a data point (ii) or a cluster.
        - We need to define:
            1. A good distance metric.
            2. A type of linkage for computing distances between point-cluster or cluster-cluster.
        - A stop criterium is required, else all ends up being one big cluster.
    - `DBSCAN`: Density-Based Spatial Clustering of Applications with Noise
        - It truly finds clusters of data, i.e., we do not partition the data; points can be left without cluster or in cluster *noise*: **outliers**! It can be used for **anomaly / outlier detection**.
        - It finds *core points* in high density regions and expands clusters from them, adding points that are at least at a given distance.
        - Points end up being *core*, *density-reachable* or *noise*, depending on their local density / connectivity to neighbors.
        - It works with namy dataset shapes, but it doesn't do well with clusters of different densities.
    - `MeanShift`
        - Similar to k-means: cluster centroid is shifted to the point with highest local density iteratively.
        - No assumption on number of clusters.
        - A window/bandwidth is defined centered in each point (standard deviation); then, the weighted mean (e.g., with kernel RBF) of the points within that window is measured to compute the new window center. The window wanders until a convergence point, called the mode = centroid. The process is repeated for every point in the dataset.
        - Robust to outliers: outliers have their own clusters.
        - Slow: complexity is `m*n^2`, with `m` iterations and `n` points.
    - **Which clustering algorithm should we use?**

      - Often, it depends on the shape of the dataset, the homogeneity of the density and the number of points.
      - DBSCAN seems to work well in many toy datasets, but density needs to be homogeneous.
      - Check: Comparing different clustering algorithms on toy datasets.
- Dimensionality Reduction
  - Curse of dimensionality: having more features is often counter-productive:
    - With more dimensions, the number of data-points we need to cover the complete feature space increases exponentially.
    - The probability of having correlations between features increases.
    - The risk of having outliers increases.
    - Solution:
      - Feature selection
      - Dimensionality reduction
  - Principal Component Analysis: `PCA`, `KernelPCA`
    - The axes/directions (= components) which account to the maximum variance in the feature space are discovered using the **Singular Value Decomposition (SVD)**. Those axes form a new base: all directions are perpendicular to each other and they have the least correlation with each other.
    - The number of default components, `n_components`, is the number of features `n` if we have more samples `m` than features, and these components are ordered according to their explained variance (singular value); however, the set is often truncated to reach a cumulative explained variance. Thus, if we take `n_components = k < n` components:
      - `X_(mxn) = U_(mxm) * S_(mxn) * V^T_(nxn)`
      - `X_hat(mxn) = U_(mxk) * S_(kxk) * V^T_(kxn)`
    - Note that not always `n < m`, i.e., there are sometimes more features than samples (e.g., with images). In the general case, the maximum value of `n_components` is `min(n_features=n, n_samples=m)`.
    - **Scaling is fundamental** for PCA.
    - **The accuracy of a model can increase if a truncated PCA-ed dataset is used!** That's because of the curse of dimensionality.
    - If we think the dataset has non-linearities, we can apply `KernelPCA`, which is equivalent to `KernelSVM`.
    - `GridSearchCV` can be applied with a custom `score()` function to find the best parameters of `PCA` / `KernelPCA`: `score = -mse(X_hat, X)`
  - Manifold learning: we reduce the dimensionality of the dataset, but maintaining the distances between the datapoint pairs in the lower dimensional space.
    - Multidimensional Scaling, `MDS`:
      - A mapping `x -> z` is achieved so that we minimize the *stress* function, which is the sum of all squared differences in distance values in each space: `min sum((d(x_i,x_j) - d(z_i,z_j))^2)`.
      - We can pass to `MDS` either
        - the dataset of points and the pairwise Euclidean distances are computed
        - or the matrix of distances between points (`dissimilarity='precomputed'`); we can use our choice of distance metric.

        - T-SNE: similar to PCA/MDS, often used to visualize datasets
    - Non-negative Matrix Factorization: `NMF`
        - We decompose our original dataset matrix as a multiplication of other two lower rank matrices. Important characteristic: all matrices need to have only positive values.
        - Examples:
            - Recommender System:
                - `V (m x u)`: movies x users, estimated rating of movie by user
                - `W (m x n)`: movies x detected **features**, feature weights of each movie
                - `H (n x u)`: detected **features** x users, weight given by each user to each feature
            - Document Topic Identification:
                - `V (m x u)`: documents x terms, frequency of term in document as TF-IDF
                - `W (m x n)`: documents x detected **topics**, topic weights of each document
                - `H (n x u)`: detected **topics** x terms, importance of each term in each topic
        - Differences wrt. PCA/SVD
            - PCA/SVD works very well for dimensionality reduction, i.e., for compression.
            - PCA/SVD is suited for datasets that have negative values.
            - NNMF works only with positive values, therefore the discovered latent features or basis components cannot be cancelled, i.e., they must be really important. In consequence, the obtained latent components represent positive and often more human interpretable elements in the dataset; e.g., if we apply NNMF to face images, each component represents the shades of eyes, nose, ears, etc.
            - If we truncate NNMF components, we lose more information, because the components are more informative.
            - NNMF doesn't provide with orthogonal latent vectors, as PCA/SVD does.

In my notes on Unsupervised Learning there are many examples and further notes.

Tips for Production

Software engineering for production deployments is out of the scope of this guide, because the goal is to focus on the data processing and analysis; however, some minor guidelines are provided. Going from a research/development environment to a production environment implies we need to:

- assure reproducibility,
- apply all checks, encodings and transformations to new data points on-the-fly,
- score the model as new data arrives seamlessly and track the results.

Thus, among others, we should do the following:

- Try to use MLOps tools like mlflow and wandb to track experiments and artifacts.
- Persist any transformation objects / encodings / parameters generated.
- Track and persist any configuration we created.
- Use seeds whenever any random variables is created.
- We should have full control over the model parameters; define dictionaries with all params in a YAML and use them at instantiation. We can apply hyperparameter tuning on top, but nothing should be left

to use default values, because **defaults can change from version to version!**
- Create python environments and use same software versions in research/production.
  - Docker containers are a nice option, or at least conda environments.
- I we don't have access to MLOps tools like mlflow and wandb, use at least `Pipelines` and pack any transformations to them; then, these can be easily packaged and deployed. That implies:
  - Using `sklearn` transformers/encoders instead of manually encoding anything,
  - or `feature_engine` classes: feature_engine,
  - or creating our own functions embedded in derived classes of these, so that they can be added to a `Pipeline` (see above: Creation of Transformer Classes),
  - or with `FunctionTransformer`, which converts any custom function into a transformer.
- Complex/hierarchical pipelines: Use `ColumnTransformer` and `make_pipeline`; look for example in `data_processing.py`.
- The inference artifact should be a *hierarchical* `Pipeline` composed by two items at the highest level:
  - `processing`: all the processing should be packed into a `Pipeline` using `ColumnTransformer`, as noted above.
  - `classifier` or `regressor`: the model.
- Avoid leaving default parameters to models, because defaults can change. Instead, save parameters in YAML files and load them as dicts; we can pass them to models at instantiation! Of course, dict key-values must be as defined in the model class.
- Modularize the code into functions to transfer it to python scripts.
- Catch errors and provide a robust execution.
- Log events.
- Go for Test Driven Development (TDD).
- Use code and dataset and version control.
- Monitor the results of the model.

Pipelines

In the following, a vanilla example with `Pipeline`; see also the section `Hierarchical Pipelines` in `data_processing.py`.

```python
# Import necessary Transformers & Co.
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline

# Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    random_state=0)

# Add sequential steps to the pipeline: (step_name, class)
pipe = Pipeline([('scaler', StandardScaler()), ('svc', SVC())])

# The pipeline can be used as any other estimator
pipe.fit(X_train, y_train)
```

```
# Inference with the Pipeline
pred = pipe.predict(X_test)
pipe.score(X_test, y_test)
```

Related Links

- My notes of the IBM Machine Learning Professional Certificate from Coursera: machine_learning_ibm.
- My notes of the Statistics with Python Specialization from Coursera (University of Michigan): statistics_with_python_coursera.
- My notes of the Udacity Data Science Nanodegree: data_science_udacity.
- My notes of the Udacity Machine Learning DevOps Nanodegree: mlops_udacity.
- My forked repository of the Udemy course Deployment of Machine Learning Models by Soledad Galli and Christopher Samiullah: deploying-machine-learning-models.
- An example where I apply some of the techniques explained here: airbnb_data_analysis.
- A Template Package to Transform Machine Learning Research Notebooks into *Production-Level* Code and Its Application to Predicting Customer Churn: customer_churn_production.

Authorship

Mikel Sagardia, 2022.
No guarantees.

You can freely use and forward this repository if you find it useful. In that case, I'd be happy if you link it to me 😊.