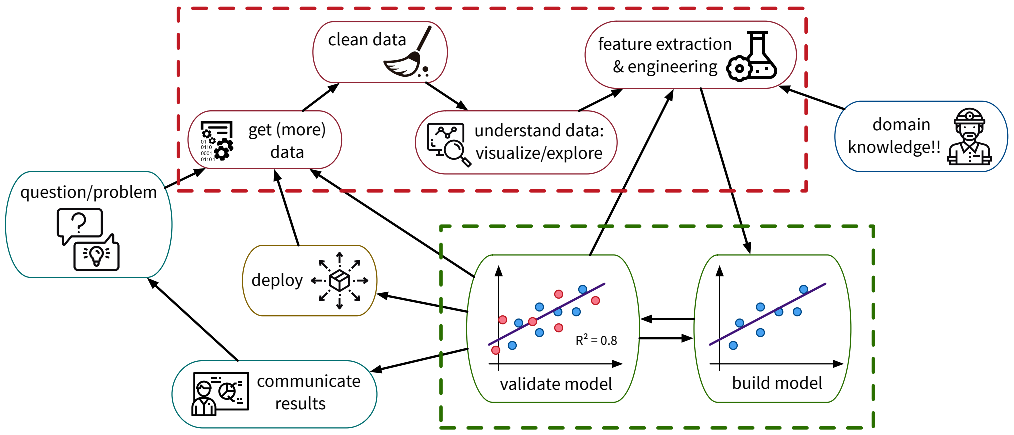
Data Science Cheat Sheet: Devising a working solution

To solve a data science problem, you will typically follow this workflow:



Most steps can be summarized under either "**data preprocessing**" (red) or "**machine learning**" (green).
Preprocessing is at least as important as the "real" machine learning stuff, i.e., you will only arrive at a decent solution if
a) the data points are represented by informative features (to avoid a **"garbage in/garbage out"** situation) and
b) you have chosen a model appropriate for the task (i.e. that is **not over- or underfitting**).

Arriving at a good solution is an iterative process: Typically, you will start with a simple pipeline (loading the data + some basic cleaning; making predictions with a simple model to get a baseline evaluation score) and then improve the results with feature engineering and a careful model selection. This means in the steps listed below you will often find yourself jumping from the preprocessing part to the machine learning steps and back again.

The code examples given in the following rely on these Python libraries:

```
import numpy as np                   # math
import pandas as pd                  # data manipulation
import matplotlib.pyplot as plt      # plotting
from sklearn.xxx import Model        # machine learning algorithms
```

1. Understand the problem/goal

- What constitutes *one data point*? What will be the raw inputs for your ML solution?
- What kind of output should be produced for each data point?
  → What category of ML algorithms can solve this task (e.g. regression, classification, clustering, ...)?

2. Preprocessing: Make the most of your data

Data Loading & Basic Cleaning

1. Get some raw data (e.g. as a .csv file) & load it:
`df = pd.read_csv("path/to/data.csv")`

2. Inspect the data, e.g., using `df.head()`, `df.describe()`, or `df.info()`

3. Feature extraction: Possibly transform raw data into meaningful numerical features (e.g. categorical variables with `sklearn.preprocessing.OneHotEncoder` or text with `sklearn.feature_extraction.text.TfidfVectorizer`)

4. Basic data cleaning:

- handle missing values (e.g. remove with `df.dropna()` or impute)
- exclude zero variance features and nonsensical variables (e.g. IDs)
- identify wrongly entered data, e.g., with an outlier detection algorithm like the γ-index
- make sure any timestamp/date columns are represented accordingly (important for plotting) → `pd.to_datetime()`

Exploratory Analysis

5. Visualize individual variables (e.g. as histograms or by plotting their time course)

```
df.hist()  # distribution of values for each variable
df.plot()  # plot values over index (possibly set a timestamp column as df.index)
```

→ Are the variables in a reasonable range and normally/uniformly distributed?

6. Check if pairs of variables are correlated and remove completely redundant features (with a correlation of 1)

```
df.corr()                              # correlation matrix
plt.scatter(df["var1"], df["var2"])    # plot one variable vs. another
```

7. Examine multiple variables at once with an interactive parallel coordinates plot (using `plotly`)

```
px.parallel_coordinates(df)    # px: plotly.express
```

Unsupervised Learning

8. If the variables are on very different scales, consider normalizing them (`StandardScaler` or `MinMaxScaler`)

```
# transform pd.DataFrame into np.array where columns have mean: 0, std: 1
scaler = sklearn.preprocessing.StandardScaler().fit(df)
X = scaler.transform(df)
```

9. Examine the intrinsic dimensionality of the data & remove noise with PCA

```
pca = sklearn.decomposition.PCA(n_components)
# transformed data has n_components columns
X_reduced = pca.fit_transform(X)
# information content of each component
print(pca.explained_variance_ratio_)
```

10. Create a 2D visualization using a dimensionality reduction algorithm like t-SNE

```
X_2d = sklearn.manifold.TSNE().fit_transform(X)
plt.scatter(X_2d[:, 0], X_2d[:, 1])
```

→ Can you already see clear patterns? If so, you know your prediction model should do okay as well.

11. If applicable: Cluster the data to detect natural groupings, e.g., using *k*-means (→ compare different `n_clusters`!)

```
clusteridx = sklearn.cluster.KMeans(n_clusters).fit_predict(X)
# scatter plot with colors based on cluster index
plt.scatter(X_2d[:, 0], X_2d[:, 1], c=clusteridx)
```

Feature Engineering

12. Can you think of additional, more complex (non-linear) features that might be helpful, e.g.,

- transformations like $\log(x+1)$ of non-normally distributed variables or
- combinations of features, e.g., computing the product/ratio of two variables?

→ Compute new features from the original values (i.e. without scaling) and then normalize all variables afterwards!

13. For regression problems, you can also transform & normalize the target variable – just make sure to undo these transformations again when generating the final predictions/output.

3. Machine Learning: Predict with and evaluate (supervised learning) models

14. Split the data into training and test (+ validation) sets (& don't look at the test set until you're done!)

```
# X: input features, y: targets
X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(X, y)
```

15. What do you want to predict?

- to which category a data point belongs (`sklearn: Classifier`)
- a continuous target variable (`sklearn: Regressor`)

→ Choose an adequate evaluation metric based on which you want to select your model (see `sklearn.metrics`), e.g., for classification problems with unequal class distributions use `balanced_accuracy_score` instead!
→ Check the baseline performance on the task (regression: predict mean; classification: most frequent class)

16. Try a simple model, e.g., linear regression or a decision tree

```
# fit a linear ridge regression model on the training set
reg = sklearn.linear_model.Ridge().fit(X_train, y_train)
# compute the R^2 score on the test set
print(reg.score(X_test, y_test))

# fit a decision tree classifier on the training set
clf = sklearn.tree.DecisionTreeClassifier().fit(X_train, y_train)
# compute the accuracy (!) on the test set
print(clf.score(X_test, y_test))
# make predictions for new test data points
y_pred = clf.predict(X_test)
# compute the balanced accuracy on the test set
print(sklearn.metrics.balanced_accuracy_score(y_test, y_pred))
```

17. Compare errors on the training & test sets to determine if the model is over- or underfitting

18. Select better hyperparameters for the model

```
# sklearn model with some hyperparameters already set
clf = sklearn.linear_model.LogisticRegression(class_weight="balanced")
# parameter values to try out (in a dict with model parameter names as keys)
params = {"C": [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]}
# perform a grid search over the parameter space (with custom scoring function)
gs = sklearn.model_selection.GridSearchCV(clf, params, scoring="balanced_accuracy")
gs.fit(X_train, y_train)
# best parameter values
print(gs.best_params_)
# predictions with the best model
y_pred = gs.predict(X_test)
```

→ Don't just rely on the best parameter values that were found, but plot results to check boundaries.
→ Choose most restrictive parameter values (for model regularization) that still give good results.

19. In case of overfitting, it might be helpful to (very carefully, manually) remove individual uninformative features, as they might facilitate the model's memorization of the training dataset. But make sure the performance on the validation set does not decrease as you remove any features.

20. In case of underfitting: consider using a more complex, non-linear model:
- dataset with few samples and many features: similarity-based model (e.g. `sklearn.neighbors.KNeighborsRegressor`, `sklearn.svm.SVC`)
- many samples / data where feature engineering is difficult (e.g. images, speech recordings): neural networks (check out the `torch` or `keras` libraries)

21. Examine individual prediction errors: Would a human make these mistakes as well & can you fix ambiguous labels? Do you notice other patterns that, e.g., call for further preprocessing steps to improve the performance?

22. If possible: interpret your model/explain the predictions: Do the results make sense? Is the model cheating by relying on features that shouldn't matter for the prediction? Can you derive any insights? (Check e.g. `model.coef_` or `model.feature_importances_` or perform a sensitivity analysis by observing how the predictions change as you vary one input variable while keeping all others at their baseline value. See also the `sklearn.inspection` submodule.)

23. Evaluate your final model on untouched test data

4. Deliver the solution

What's the story behind the results you want to communicate?
How can your model (incl. preprocessing pipeline) be deployed and used in production?