## Handling data

| Syntax | Summary |
|---|---|
| `pd.read_csv(path, index=False)` | Import a CSV file and reset the indexing |
| `df.rename(columns={old: new})` | Rename columns of a dataframe |
| `pd.merge(df1, df2, on=col)` | Merge two dataframes by using a specific column |
| `pd.concat([df1, df2], axis=1)` | Concatenate two dataframes column-wise |
| `df.drop(col, axis=1)` | Drop a column |
| `df.nlargest(n, col)` | Get the rows with the largest value in a column |
| `df.sort_values(col, ascending=False)` | Sort the values of a column in descending order |
| `df.groupby(group)[col].agg([np.mean, np.std, 'size'])` | Group rows of a dataframe by a certain column and compute different metrics on all groups |
| `df.groupby(...)[...].agg(...).reset_index()` | "Ungroup" the dataframe |
| `round(value, n)` | Round a value to a specific number of decimal digits |
| `df.drop_duplicates([combo], keep='first')` | Drop the duplicates of a certain combination of rows and keep the first one |
| `df[col].isna()` | Returns an array of booleans that match if the values in the column are NaN or not |
| `df.at[index, col]` | Access the dataframe at a certain row (using index) and column |

## Data visualisation

| Syntax | Summary |
|---|---|
| `plt.hist(data, bins=100)` | Create a histogram |
| `plt.boxplot(data)` | Create a boxplot |
| `plt.scatter(x, y, s=10)` | Create a scatter plot with specified dot size |
| `sns.jointplot(data, x, y)` | Create a jointplot (two histograms joining in the middle to form level curves) |
| `sns.barplot(x=input, y=output, data=df)` | Create a bar chart |
| `sns.boxplot(x=input, y=output, data=df)` | Create one boxplot per categorical value |
| `plt.errorbar(x, y, yerr=std)` | Create a lineplot with error bars/CI |

| Syntax | Summary |
|---|---|
| `plt.fill_between(x, y1, y2)` | Create two lineplots in a single graph and fill the area between the curves |
| `fig, ax = plt.subplots(nrows=m, ncols=n, figsize=(a,b), sharey=True, sharex=True)` | Subplot template with specified number of panels |
| `pd.crosstab(x1, x2, values=y, aggfunc='mean')` | Create heatmap data for two categories x1 and x2 with colour coding for y |
| `hist[0]` | Access bin heights of a histogram |
| `hist[1]` | Access bin edges of a histogram |
| `sns.pairplot(df)` | Make a pairplot (many subpanels that compare each feature with all others) |
| `sns.ecdfplot(values, complementary=True)` | Create a (C)CDF plot for an array of values |

## Describing data

| Syntax | Summary |
|---|---|
| `df[column].describe()` | Get different metrics on a dataframe column (mean, std, quartiles, ...) |
| `diagnostic.kstest_normal(df[column].values, dist = 'norm')` | Goodness-of-fit test for the data of a dataframe column |
| `df.sample(n=10, replace=False, weights=df[col])` | Sample rows of a dataframe without replacement with prioritising large values in a column |
| `stats.pearsonr(df[col1], df[col2])` | Get Pearson correlation coefficient between two columns |
| `stats.ttest_ind(df[col1], df[col2])` | Independent t-test to test for similarity of means for two columns |

## Regression analysis

| Syntax | Summary |
|---|---|
| `model = smf.ols(formula='y ~ x1 + x2 + x3:x4', data=df)` | Build a linear regression model |
| `model = smf.logit(formula='y ~ x1 + x2 + x3:x4', data=df)` | Build a logistic regression model |
| `res = model.fit()` | Grab the results of the regression |
| `res.summary()` | Print the estimated coefficients and associated p-values |

| Syntax | Summary |
|---|---|
| `np.log(p / (1 - p))` | Compute the log odds of a certain probability |
| `np.exp(odds) / (1 + np.exp(odds))` | Compute the probability of certain log odds |

## Observational data

| Syntax | Summary |
|---|---|
| `G = nx.Graph()` | Create a NetworkX graph |
| `G.add_weighted_edges_from([(index_t, index_c, similarity)])` | Populate the graph with nodes and edges depending on similarity |
| `matching = nx.max_weight_matching(G)` | Grab the pairs of indices that are the most similar |

## Supervised learning

| Syntax | Summary |
|---|---|
| `lin_reg = LinearRegression()` | Create a linear regression model |
| `lin_reg.fit(X, y)` | Train the model |
| `lin_reg.intercept_` | Get the estimated y-intercept |
| `lin_reg.coef_` | Get the estimated feature coefficients |
| `predicted = cross_val_predict(lin_reg, X, y, cv=n)` | Predict outputs using cross-validation with n folds |
| `Ridge(alpha=a)` | Create a Ridge-regularised model with specific alpha value |
| `pd.get_dummies(df, prefix='onehot-')` | Turn all categorical columns to a one-hot encoded representation |
| `LogisticRegression(solver='lbfgs')` | Create a logistic regression model with specified solver |
| `precision = cross_val_score(logistic, X, y, cv=10, scoring="precision")` | Returns an array of precision scores for the cross-validation |
| `recall = cross_val_score(logistic, X, y, cv=10, scoring="recall")` | Returns an array of recall scores for the cross-validation |
| `fpr, tpr, _ = roc_curve(y, predicted[:, 1])` | Get the false & true positive rates to plot the ROC curve |
| `auc(x, y)` | Get the area under a curve |

| Syntax | Summary |
|---|---|
| `model_name.predict_proba(X_test)` | Get the probability distribution behind a specific prediction |
| `KNeighborsClassifier(k)` | Create a kNN model with specified number of neighbours |
| `RandomForestClassifier(n_estimators=n, max_depth=3, random_state=0)` | Create a random forest model with specific number of trees and depth |
| `cross_validate(model_name, X, y, cv=30, scoring=('accuracy', 'precision', 'recall'))` | Get scores by cross-validation |

### Applied machine learning

| Syntax | Summary |
|---|---|
| `sorted(coeff_dict.items(), key=lambda item: item[1])` | Sort a dictionary by ascending order of values |

### Unsupervised learning

| Syntax | Summary |
|---|---|
| `kmean = KMeans(n_clusters=k, random_state=0).fit(X)` | Get the k-means clustering result from the data X |
| `kmean.labels_` | Get the classification results (0, 1, 2, ...) for each data point |
| `kmean.cluster_centers_` | Get the coordinates of the cluster centers (k centers) |
| `labels = KMeans(n_clusters=k, random_state=0).fit_predict(X)` | Directly get the data labels |
| `silhouette_score(X, labels)` | Get the silhouette score of the clustering for the specific k value |
| `kmean.inertia_` | Get the sum of square errors for the specific k value |
| `X_reduced_tsne = TSNE(n_components=2, init='random', learning_rate='auto', random_state=0).fit_transform(X)` | Reduce dimensionality with t-SNE |
| `X_reduced_pca = PCA(n_components=2).fit(X).transform(X)` | Reduce dimensionality with PCA |

### Handling text

| Syntax | Summary |
|---|---|

| Syntax | Summary |
|---|---|
| `with open(path, encoding="utf-8") as f:` | Open a text document to start parsing it |
| `f.readlines()` | Give a list to iterate through the lines of the document |
| `line.startswith(str)` | Return a boolean that tells if a string starts with a substring |
| `substr1, substr2 = str.split(char, 1)` | Split a string in two substrings with a certain character as the separator |
| `words = line.split()` | Split a line into words |
| `str.replace(char1, char2)` | Replace all characters from a string to another character |

## Exam 2022

| Syntax | Summary |
|---|---|
| `G = nx.from_pandas_edgelist(df, 'SRC', 'TGT', [edge_metrics], create_using=nx.Graph)` | Create a graph from a Pandas dataframe |
| `nx.MultiDiGraph()` | Directed graph that allows multiple edges between two nodes |
| `G.degree()` | Get a dictionary of nodes with their degrees |
| `G.in_degree()` | Get a dictionary of nodes with their in-degrees |
| `G.out_degree()` | Get a dictionary of nodes with their out-degrees |
| `nx.enumerate_all_cliques(G)` | Returns all 'cliques' or groups of inter-connected nodes in a network |
| `nx.get_edge_attributes(G, label)` | Get a specific edge attibute for all edges in the network |
| `nx.get_node_attributes(G, label)` | Get a specific node attibute for all edges in the network |
| `roc_auc_score(y_test, y_pred)` | Get the AUC of the ROC curve in one step |
| `VCT = TfidfVectorizer(max_features=150, stop_words='english')` | Initialise a TF-IDF model |
| `X = VCT.fit_transform(document_list).toarray()` | Create a TF-IDF matrix |

## Exam 2021

| Syntax | Summary |
|---|---|
| `G.add_node(value, attribute_1=value, attribute_2=value)` | Add a node to a graph and give it attributes |
| `G.add_edge(value, attribute_1=value, attribute_2=value)` | Add a node to a graph and give it attributes |
| `pd.qcut(values, n_quantiles, labels=[names])` | Cut a list of values in quantiles with specific names |
| `X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.4, random_state=42)` | Cut the regression data into train and test sets |

## Exam 2020

| Syntax | Summary |
|---|---|
| `sns.countplot(values)` | Make a barplot where the bar heights are the counts of the categories in the value list |
| `pd.factorize(values)` | Transform a list of categrories into a list of integer labels (returns a tuple of the factorised and categorical labels) |
| `SGD = SGDClassifier(penalty='l2', loss='log', max_iter=5, tol=None, alpha=1e-4, random_state=42, class_weight='balanced')` | Logit using stochastic gradient descent (possibility of balancing the classes if one is way smaller) |
| `classification_report(y_test, y_pred)` | Get many metrics for the classification result (precision, recall, F1, accuracy, macro avg, ...) |
| `confusion_matrix(y_test, y_pred)` | Get the confusion matrix of the classification |
| `nx.is_weakly_connected(G)` | Boolean that tells if the graph is weakly connected (also exists for strongly connected) |
| `nx.weakly_connected_components(G)` | Return the number of weakly connected components (also exists for strongly connected) |
| `nx.eigenvector_centrality(G)` | Compute eigenvector centrality of graph |
| `nx.maximal_matching(G)` | Get the matched pairs out of the created matching graph |

## Exam 2019

| Syntax | Summary |
|---|---|

| Syntax | Summary |
|---|---|
| `df.dtypes` | Give the type of the elements in each column of the dataframe |
| `pd.to_datetime('2010-01-01')` | Convert a date string to a Pandas datetime object |
| `series.dt.year` | Return the years of a series of datetimes |
| `date.year` | Return the year of a datetime object (also exists for month) |
| `np.arange(m, n)` | Create an array of integer values from m to n-1 included |
| `df[col].apply(lambda x: f(x))` | Apply a function to a whole dataframe column |
| `np.percentile(list, n)` | Get the value in the (not necessarily sorted) list corresponding to the n-th percentile |
| `get_scorer_names()` | Get all the possible values of the 'scoring' parameter of cross_val_score() |
| `cv = GridSearchCV(model_name, {hyperparam:(0.1,0.01,0.001)}, cv=n)` | Create a cross-validation model to tune the hyperparameter |
| `cv.fit(X_train, y_train)` | Train the hyperparameter tuning model |
| `cv.cv_results_['mean_test_score']` | Get the R-squared for all values of the hyperparameter |
| `cv.predict(X_test)` | Perform regression on the test set using the best hyperparameter |
| `lcv = LogisticRegressionCV(Cs=(1,10,100), cv=3, random_state=42, max_iter=200)` | Same thing but specifically for logit |
| `lcv.C_[0]` | Get the optimal hyperparameter C for the logistic model |
| `json.load(open(path, 'r'))` | Load a JSON file to a dictionary |
| `nx.diameter(G)` | Get the diameter (longest shortest path) of the graph (may return an error if graph is not connected) |
| `nx.number_connected_components(G)` | Get the number of connected components in the graph |
| `nx.betweenness_centrality(G)` | Get a dictionary of nodes and their betweenness centralities |
| `kernighan_lin_bisection(G, max_iter=100, seed=42)` | Split the graph into 2 'communities' based on connectedness |