Python and libraries cheat sheet

pythondata.md

Python

Lists

- `[1, 2, 3, 4, 5]`

Power operator (2^5)

- `2 ** 5`

Square root (power operator trick)

- `9 ** 0.5`

Floor division (outputs the int part; always rounds down)

- `15 // 4`

Check method quickdocs with `?`

- `method_name?`

NumPy

- `import numpy as np`

Create arrays

Create an array with 10 zeros.

- `a = np.zeros(10)`

Create an array with 5 ones.

- `a = np.ones(5)`

Create an array with 10 times the number 2.5

- `a = np.full(10, 2.5)`

Create an array from a Python list

- `a = np.array([1, 2, 3, 4])`

Create an array from a number range

- `a = np.arange(10)` -> [0,10), from 0 to 9
- `a = np.arange(3,10)` -> [3, 10)

Create an array with evenly separated numbers (floats)

- `a = np.linspace(0,1,11)` -> 11 numbers in the interval [0,1] (including 1)

Acess the second position in the array

- `a[2]`

Multidimensional arrays

Create a 5x2 (5 rows, 2 columns) matrix of zeros

- `a = np.zeros((5,2))`

Access an element

- `a[0,1]` -> Acess row 0, element 1

Assign numbers to a full row

- `a[1] = [1,2,3]`

Access a column

- `a[:,1]`

Stack arrays together to create multidimensional arrays

- `np.stack(sequence_of_arrays, axis=0)` stacks the sequence of arrays in the specified axis
- `np.column_stack(array_tuple)` takes 1D arrays and stacks them as columns, in the same manner as `axis=1` in `np.stack()`.
- `np.hstack(array_tuple)` concatenates along the second axis except for 1D arrays. the behavior is a little confusing; this is mostly intended for specific scenarios such as image data.
- `np.row_tack()` and `np.vstack()` are also offered and are virtually the same.

Random arrays

Set a seed for reproduceable results (in this example, seed = 10)

- `a = np.random.seed(10)`

Create a random 5x2 array using a uniform distribution (random floats between 0 and 1)

- `a = np.random.rand(5,2)`

Create a random 5x2 array using a uniform distribution, but with numbers between 0 and 100

- `a = 100 * np.random.rand(5,2)`

Create a random 5x2 array of integers in [0, 100) interval

- `a = np.random.randint(low=0, high=100, size=(5,2))`

Element-wise operations

Add a number/scalar to all elements in the array (you can use any operator for multiplications, etc)

- `a + 1`

Element-wise operators (e.g. sum each element with the element of the other array)

- `a + b`

Compare all elements to a scalar -> outputs an array of bools of same size as original array, with each comparison result

- `a >= 2`

Element-wise comparison of 2 arrays

- `a > b`

Create an array that matches the conditions of the comparison (e.g. return an array with all the elements of `a` greater than their `b` counterparts)

- `a[a > b]`

"Summarizing" operations

- `a.min()`
- `a.sum()`
- `a.mean()`

Standar deviation

- `a.std()`

Dot product

Works for vector-vector dot product (multiplies and then adds all elements; returns a single scalar), as well as matrix-vector (returns a vector) and matrix-matrix (returns a matrix)

- `np.dot(a,b)`
- `a @ b`

SPECIAL: log/exp transformation

Sometimes it's convenient to transform the data. Applying a log function to data helps normalize the data (logs of very high values become smaller and closer to logs of regular values) which is very useful for ML with special distributions such as long tail.

- `np.log(list)` applies a log transformation to a list, but `np.log(0)` will return an error.
- `np.log1p(list)` will add 1 to each element of the list before applying log in order to avoid such errors. It's identical to `np.log(list + 1)`.

`np.log1p()` is usually applied to features before using them for training.

After calculating any results using these transformed features, it is necessary to "untransform" the result by applying an exponential function, which is the inverse of log.

- `np.exp(list)` applies an exponential transformation to a list.
- `np.expm1(list)` will substract 1 to each element after applying exp in order to reverse the `np.log1p` transformation. It's identical to `np.exp(list) - 1`.

Pandas

Create and access

Pandas is a library for working with tabular data. pandas has its own datatype for tables, called DataFrame

- `import pandas as pd`

Create a DataFrame from 2 Python lists, one for the data and another for the headers ("columns" in pandas)

- `data = [['Nissan', 'Stanza', 1991],['Hyundai', 'Sonata', 2017],['Lotus', 'Elise', 2010]]`
- `headers = ['Make', 'Model', 'Year']`
- `df = pd.DataFrame(data, columns=headers)`
- `colunns` is optional. If missing, the headers will be named numerically in order.

Create a DataFrame from a dictionary list

- `df = pd.DataFrame(dicts)`
- pandas will use the keys as columns headers and the values for the data

Display the first rows of the dataframe

- `df.head()`
- Optionally, `df.head(n=2)` to only show a certain amount of rows (default is 5)

Access a column

- `df.ColumnTitle`
- `df['Make']` -> Useful if the tile contains spaces

Return a dataframe with a column subset

- `df[['Make', 'Model']]`

Add a new column to an existing DataFrame

- `df['id'] = [1,2,3]`

Delete a column

- `del df['id']`

Display row id's ("index" column, created by default)

- `df.index`

Acess a row/table entry by its index

- `df.loc[1]`
- `df.loc[1,2]`

Change indeces to something else

- `df.index = ['a', 'b', 'c']`

Access rows by absolute numerical index even after the indices have been changed to something else

- `df.iloc[1]`

Restore indices to numbers

- `df.reset_index()` -> this actually creates a named `index` column with the user-created indices and it also adds a nameless column with the original numerical indices
- `df.reset_index(drop=True)` -> gets rid of user-created indices and restores numerical indices.

Concatenate 2 DataFrames

- `pd.concat([df1, df2])`

Pivot (reshape) a DataFrame, based on column values.

- `df.pivot()`
- Additional params:
  - `index` (default: `None`): columns to use to to make new DataFrame's index. If `None`, uses existing index.
  - `columns` (default: `None`): columns to use to make new frame's columns.

    - `values` (default: `None`): columns to use for populating new DataFrame's values. If `None`, all remaining columns will be used and the result will have hierarchically indexed columns.

Element-wise operations

Apply operator to every single element of a column

- `df['Year'] - 5`

Compare all values of column with a single value

- `df['Year'] >= 2015 ->` returns a dataframe with 2 columns and the same number of rows as `df`: index column and a column of booleans with the result of the comparison

Filter a DataFrame and output only the rows that match certain condition

- `df[ df['Year'] >= 2015 ]`
- `df[ (df['Year'] >= 2015) & (df['Make'] == 'Nissan') ]`

String operations

Make a string lowercase

- `'FuNkY sTrInG'.lower()`

Make all elements in a DataFrame column lowercase

- `df['Make'].str.lower()`

Replace spaces with underscores

- `A string with spaces'.replace(' ', '_')`

Concatenate operations

- `df['Make'].str.replace(' ', '_').str.lower()`

"Summarizing" operations

Similar operations to NumPy

- `df.Year.min()`

Output multiple summarizing stats

- `df.Year.describe()`
- `df.Year.describe().round(2)` -> Rounds to 2 decimal numbers

Count unique values

- `df.Make.nunique()` -> unique values in a single column
- `df.nunique()` -> unique values of all columns

Display NaN's

- `df.isnull()` -> Displays the DataFrame with False/True for each element (True if NaN)
- `df.isnull().sum()` -> lists amount of NaN's per column

Grouping

For SQL-like operations

```
SELECT
   Make,
   AVG(Year)
FROM
   cars
GROUP BY
   Make
```

- `df.groupby('Make').Year.mean()`

Get values from the DataFrame

Get NumPy array with column values

- `df.Year.values`

Returns a list of dictionaries with the complete DataFrame

- `df.to_dict(orient = 'records')`

Exploratory Data Analysis

Create a correlation matrix

- `df.corr()`

Calculate the correlation between 2 features

- `df['feature1'].corrwith(df['feature2])`

Scikit-learn

`from sklearn.model_selection import train_test_split`

Split a pandas DataFrame intro train/validation/test splits, 60/20/20

- The function only splits into 2, so if we need 3 splits, we need to use the function twice
- `df_full_train, df_test = train_test_split(df, test_size=0.2, random_state=1)`
- Our original splits are 60/20/20; we already have the last 20% in the test dataset. We now need to split the remaining 80%. We cannot use test_size=0.2 because we would be splitting 20% of that 80%, not of the 100% ! We recalculate: 20 / 80 = 1/4 = 0.25
- `df_train, df_val = train_test_split(df_full_train, test_size=0.25, random_state=1)`

`from sklearn.metrics import mutual_info_score`

Get the mutual information of 2 features (usually a feature and the target)

- `mutual_info_score(df.feature_1, df.feature_2)`
- The closest the value is to `1`, the more dependant they are between each other.
- When comparing a feature with the target, the closest the score is to `1`, the more important is that feature for our model.

`from sklearn.feature_extraction import DictVectorizer`

Encode categorical features as one-hot vectors. We need to create a Vectorizer object first.

- `dv = DictVectorizer(sparse=False)`
- By default, the vectorizer uses Sparse Matrices. If you do not need them, you may set `sparse=False` .
- More info on sparse matrices: https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

Learn a list of feature names -> indices mappings.

- `dv.fit(my_dictionary_list)`
- `my_dictionary_list` is a list of dictionaries; each entry is a dictionary that contains a row of our data. You usually obtain it from a DataFrame with `df.to_dict(orient='records')` ; you may check the different options for `orient` at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_dict.html
- `dv.fit()` returns itself; you do not assign it to a variable.
- For categorical values, each value is converted into its own feature.
- Numerical values are left untouched.

Get the learnt feature names.

- `dv.get_feature_names()`
- Each categorical feature will be listed as many times as values there are in the feature (if the feature has values *yes*, *no* and *maybe*, the feature will appear as *feature=yes*, *feature=no* and *feature=maybe*)
- Numerical values will be displayed as usual.

Transform the feature -> value dicts to an array

- `dv.transform(my_dictionary_list)`
- `my_dictionary_list` would usually be the same dictionary list used in the `fit()` function, or a subset of it.
- Outputs the actual one-hot array we need for our model.
- Each row will contain a vector in which each categorical feature is one-hot encoded and numerical values will be left intact.

Learn a ilst of feature names -> indices mappings and output an array straight ahead (combine both `fit()` and `transform()` in one method)

- `dv.fit_transform(my_dictionary_list)`

`from sklearn.linear_model import LogisticRegression`

Train a Logistic Regression model for Classification (both Binary and Multiclass)

```
model = LogisticRegression()
model.fit(X_train, y_train)
```

Check the bias of the model.

- `model.intercept_`

Check the weights of the model

- `model.coef_`
- Outputs a 2-dimensional array. You may want to obtain the 1D array with all the weights with `model.coef_[0]`

Calculate the ***hard predictions*** (output as either `0` or `1`) of a dataset.

- `model.predict(X)`
- Outputs an array with a prediction for each row of the dataset.
- The threshold for deciding `0` or `1` is `0.5` by default.

Calculate the ***probabilities*** rather than the hard predictions.

- `model.predict_proba(X)`
- Outputs pairs of probabilities for each row: probability of class `0` and of class `1`.
- You may want to obtain the probabilities for just class `1`; you may do so with `model.predict_proba(X)[:,1]`

`from sklearn.metrics import roc_curve, auc, roc_auc_score, mean_square_error`

Calculate the FPR, TPR and threshold values for the ROC curve of a given target and prediction.

- `fpr, tpr, thresholds = roc_curve(y_val, y_pred)`

Calculate the ROC AUC score metric given FPR and TPR arrays of a ROC curve.

- `auc(fpr, tpr)`

Calculare the ROC AUC score metric directly from given target and predictions. Useful evaluation metric for Classification.

- `roc_auc_score(y_val, y_pred)`

Calculate the Root Mean Square Error between a prediction array and the ground truth. Useful error function for Regression.

- `rmse = mean_squared_error(y_val, y_pred, squared=False)`

`from sklearn.model_selection import KFold`

Create a k-fold of size `n_splits` for K-Fold Cross Validation.

- `kfold = KFold(n_splits, shuffle=True, random_state=1)`
- Returns a k-fold object for iterating different training and validation datasets.

Create a list of shuffled indices representing different permutations of training and validation datasets.

- `train_idx, val_idx = kfold.split(df_full_train)`
- `df_full_train` is the dataframe that contains all the data that will be used to create the k-folds (in other words, the complete dataset except the test dataset)
- `kfold` is the object created by the previous `KFold()` method.
- `train_idx` and `val_idx` contain the list of shuffled indices that can be used on `df_full_train` to split it, like so:
  - `df_train = df_full_train.iloc[train_idx]`
  - `df_val = df_full_train.iloc[val_idx]`
- The `split()` method is convenient to use in a `for` loop because it will return multiple iterable sets of arrays for cross-validation.

`from sklearn.import DecisionTreeClassifier, DecisionTreeRegressor`

Train a Decision Tree Classifier

```
dt = DecisionTreeclassifier()
dt.fit(X_train, y_train)
```

Predictions are handled similarly to Logistic Regression.

- `y_pred = dt.predict_proba(X_val)[:, 1]`

Train a Regressor Tree Classifier

```
dt = DecisionTreeRegressor()
dt.fit(X_train, y_train)
```

Important hyperparameters when creating trees:

- `max_depth` (default = `6`)
- `min_samples_leaf` (default = `1`). Minimum amount of samples a leaf must have in order to consider the split as valid/worth considering.

`from sklearn.tree import export_text`

Print a Decision Tree

- `print(export_text(dt, feature_names=dv.get_feature_names()))`

`from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor`

Train a Random Forest Classifier

```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
```

Train a Random Forest Regressor

```
rf = RandomForestRegressor()
rf.fit(X_train, y_train)
```

Additional hyperparams for both Random Forests:

- `max_depth`
- `min_samples_leaf`
- `n_estimators` (default: `100`): number of trees in the forest.

- `max_features` (default: `auto`): Number of features to consider for each split.
- `bootstrap` (default: `True`): if `False`, the whole dataset is used to build each tree; otherwise, each tree is built with random subsamples *with replacement* (datapoints can be repeated), AKA bootstrapping.

`from sklearn.feature_extraction.text import CountVectorizer`

Encode text features as vectors

```
cv = CountVectorizer()
cv.fit(feature)
```

Reduce size of large categorical features as well as vectorize the categories.

1. Preprocess the features so that no whitespaces remain (substitute the whitespaces with underscores, etc).
2. Create a `string` containing the contents of all the features that need processing.
    - `my_features = 'feature_1=' + df.feature_1 + ' ' + 'feature_2=' + df.feature_2`
    - This will output a single string with all of the contents of the chosen features in the `df` dataframe.
3. Train a `CountVectorizer` instance. from sklearn.feature_extraction.text import CountVectorizer

```
    cv_features = CountVectorizer(token_pattern='\S+', min_df=50,
dtype='int32')
    cv_features.fit(my_features)
```

    - `token_pattern` allows you to define a ***regular expression*** to parse the string. Lowercase `\s` means whitespace, uppercase `\S` is everything that is not a whitespace; `+` means that at least 1 instance of the preceding character or token must occur; therefore, this regular expression will parse any collection of consecutive non-whitespace characters as an entity, and whitespaces will be used as the separation between entities.

- `min_df` stands for *minimum document frequency*. `CountVectorizer` will only take into account the categories that appear at least as many times as the amount specified in this parameter (in our example, 50). All other categories with counts smaller than the specified amount will be discarded.
- The default type is `int64`; we switch to `int32` because the generated vectors will be made of zeros and ones, so we can cut the memory usage by half by changing the type without losing any info.
- `cv_features.get_feature_names()` should return a list similar to this:

```
['feature_1=type_1',
'feature_1=type_2',
'feature_2=type_A',
'feature_2=type_B']
```

4. `X = cv_features.transform(my_features)`
   - This will convert your string to vectors.

Join processed text and categorical features

- `scipy.sparse.hstack([X_categories, X_texts])`

```
  # Categorical features
  cv_categories= CountVectorizer(token_pattern='\S+', min_df=50,
dtype='int32')
  cv_categories.fit(my_features)

  # Text features
  cv_texts = CountVectorizer()
  cv_texts.fit(my_text_features)

  # Creating the feature matrices
  X_categories = cv_categories.transform(my_features)
  X_texts = cv_texts.transform(my_text_features)

  # Stacking the 2 feature matrices together into one
  import scipy
  X = scipy.sparse.hstack([X_categories, X_texts])

  # Optional matrix reformatting
  X = X.tocsr()
```

from sklearn.compose import ColumnTransformer

Apply transformations to a Pandas dataframe

1. Define transformations in an array.
2. Create a ColumnTransformer
3. Fit the ColumnTransformer on the dataframe
4. Transform the dataframe

```
transformations = [
    ('numerical', 'passthrough', ['num_feat1', 'num_feat2', 'num_feat3']),
    ('categories', OneHotEncoder(dtype='int32'), ['cat_feat1', 'cat_feat2',
'cat_feat_3']),
    ('name', CountVectorizer(min_df=100, dtype='int32'), 'name')
]

transformer = ColumnTransformer(transformations, reminder='drop')

transformer.fit(df_train)

X = transformer.transform(df_train)
y = df_train.target_feature.values
```

from sklearn.pipeline import Pipeline

Create a pipeline which automatically handles the dataframe transformations with a ColumnTransformer

```
transformations = [
    ('numerical', 'passthrough', ['num_feat1', 'num_feat2', 'num_feat3']),
    ('categories', OneHotEncoder(dtype='int32'), ['cat_feat1', 'cat_feat2',
'cat_feat_3']),
    ('name', CountVectorizer(min_df=100, dtype='int32'), 'name')
]

tranformer = ColumnTransformer(transformations, remainder='drop')

pipeline = Pipeline([
    ('transformer', transformer),
    ('lr', LinearRegression())
])
```

```
pipeline.fit(df_train, df_train.target_feature.values)
pipeline.predict(df_val)
```

from sklearn.base import TransformerMixin

Create a custom transformer

1. Create a new class that extends `TransformerMixin`
    - Implement the `fit` and `transform` methods.

```
class ConcatenatingTranformer(TransformerMixin):

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        columns = list(X.columns)
        res = ''
        for c in columns:
            res = res + ' ' + c + '=' + X[c]
        return res.str.strip()
```

2. Instantiate the custom class
3. Add it to a pipeline if desired

```
ct = ConcatenatingTranformer()

p2 = Pipeline([
    ('concatenate', ConcatenatingTranformer()),
    ('vectorize', CountVectorizer(token_pattern='\S+', min_df=100))
])

# Optional
X = p2.fit_transform(df[['cat_feat1', 'cat_feat2', 'cat_feat_3']])
```

XGBoost

```
import xgboost as xgb
```

Create a DMatrix (optimized structure for XGBoost)

- `dtrain = xgb.DMatrix(X_train, label=y_train, feature_names=features)`
  - `X_train` and `y_train` are obtained as usual with scikit-learn.
  - `features` can be obtained for example from `DictVectorizer` with `dv.get_feature_names()` (soon to be deprecated!)

Train a model

- `model = xgb.train(xgb_params, dtrain, num_boost_round=10)`
  - `num_boost_round` (default: `10`) is the amount of iterations. Equivalent to `n_estimators` in Scikit-Learn.
  - Additional hyperparameters:
    - `verbose_eval` (default: `True`): prints the evaluation metrics on `stdout`. If set to an integer, it will print the evaluation metrics at every given `verbose_eval` stage (if set to `5`, it will print every 5 steps).
    - `evals` can take a list of pairs `(Dmatrix, string)` known as *watchlist*; if set, the model will evaluate with the given datasets (ususally defined as `watchlist = [(dtrain, 'train'), (dval, 'val')]` and then `evals = watchlist`)
  - `xgb_params` is a dictionary with various params for XGBoost. Example below.

Example of `xgb_params` dictionary for XGBoost

```
xgb_params = {
  'eta': 0.3,
  'max_depth': 6,
  'min_child_weight': 1,

  'objective': 'binary:logistic',
  'nthread': 8,

  'seed': 1,
  'verbosity': 1,
}
```

- `eta` (default: `0.3`) is the learning rate.
- `max_depth` (default: `0.6`) behaves identically as in scikit-learn.
- `min_child_weight` (default: `1`) behaves identically to `min_samples_leaf` in scikit-learn.
- `colsample_bytree` (default: `1`) is the subsample ratio of features/columns when constructing each tree.

- `subsample` (default: `1`) is the subsample ratio of the training instances/rows.
- `lambda` (default: `1`) AKA L2 regularization.
- `alpha` (default: `0`) AKA L1 regularization.

Predict with XGBoost

- `y_pred = model.predict(dval)`

XGBoost outputs to `stdout` during training. Auxiliary functions may be defined to capture the output and store it as DataFrames if necessary.

Tensorflow & Keras

```
import tensorflow as tf

from tensorflow import keras
```

Load an image

```
from tensorflow.keras.preprocessing.image import load_img

# filepath is the path to the file containing an image
img = load_img(filepath, target_size=(299, 299))
```

Create image datasets from directories

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_gen = ImageDataGenerator(preprocessing_function=preprocess_input)
train_ds = train_gen.flow_from_directory('./clothing-dataset-small/train',
target_size=(150, 150), batch_size=32, class_mode='categorical')

val_gen = ImageDataGenerator(preprocessing_function=preprocess_input)
val_ds = train_gen.flow_from_directory('./clothing-dataset-small/train',
target_size=(150, 150), batch_size=32, class_mode='categorical', shuffle=False)
```

- There is no need to shuffle the validation dataset.
- `class_mode` specifies the kind of label that the dataset should take into account. `'categorical'` for multiclass classification, `'binary'` for binay classification.

Data augmentation

```
train_gen = ImageDataGenerator(
    preprocessing_function=preprocess_input,
    rotation_range=30,
    width_shift_range=10.0,
    height_shift_range=10.0,
    shear_range=10.0,
    zoom_range=0.1,
    vertical_flip=True,
    horizontal_flip=False,
)
```

Wrap an image inside an array ("batchify" a single image)

```
X = np.array([x])
```

Load a pretrained network (Xception), preprocess and predict

```
from tensorflow.keras.applications.xception import Xception
from tensorflow.keras.applications.xception import preprocess_input
from tensorflow.keras.applications.xception import decode_predictions

model = Xception(weights='imagenet', input_shape(299, 299, 3))

# X is an image batch
X = preprocess_input(X)

pred = model.predict(X)
decore_predictions(pred)
```

Freeze a model or layer

```
model.trainable = False
```

Define a network, functional style

```
inputs = keras.Input(shape=(150, 150, 3))
base = base_model(inputs, training=False)
vectors = keras.layers.GlobalAveragePooling2D()(base)
outputs = keras.layers.Dense(10)(vectors)
model = keras.Model(inputs, outputs)
```

Define a network, sequential style

```
model = keras.models.Sequential()

model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Define optimizer, loss function and compile a model for classification

```
learning_rate = 0.01
optimizer = keras.optimizers.Adam(learning_rate=learning_rate)

loss = keras.losses.CategoricalCrossentropy(from_logits=True)

model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])
```

Define optimizer, loss function and compile a model for regression

```
loss = keras.losses.MeanSquaredError()
rmse = keras.metrics.RootMeanSquaredError()

model.compile(optimizer=optimizer, loss=loss, metrics=[rmse])
```

Train a network and obtain accuracy scores

```
history = model.fit(train_ds, epochs=10, validation_data=val_ds)
scores[lr] = history.history
```

Checkpointing

```
model.save_weights('model_v1.h5', save_format='h5')

checkpoint = keras.callbacks.ModelCheckpoint(
    'xception_v1_{epoch:02d}_{val_accuracy:.3f}.h5',
    save_best_only=True,
    monitor='val_accuracy',
    mode='max'
)

learning_rate = 0.001

model = make_model(learning_rate=learning_rate)

history = model.fit(
    train_ds,
```

```
    epochs=10,
    validation_data=val_ds,
    callbacks=[checkpoint]
)
```

Evaluate a model

```
model.evaluate(test_ds)
```

Matplotlib and Searborn

```
import matplotlib.pyplot as plt

import searborn as sns
```

Plot a histogram of a DataFrame column

- `sns.histplot(df.column_name, bins=50)`
- `sns.histplot(df.column_name[df.column_name < 100], bins=50)`
- `bins` is the amount of columns or groupings that the histogram will use to group values ("bins")

Pickle

```
import pickle
```

Save a variable (such as a model) to a file.

- `f_out = open('filename', 'wb')`
  - This will create a `file` object to which we can write whatever we want.
  - `filename` is just a string with the filename we want.
  - `'wb'` states that we want to *write* to the file (`w`) and that the file contents will be in binary format (`b`).
- `pickle.dump(varname, f_out)`
  - This will dump the contents of `varname` into the `f_out` file.
  - You can pack multiple vars in a struct such as a tuple. Convenient for saving a model and a vectorizer for one-hot encoding in a single file.
- `f_out.close()`
  - This will close the file to store the contents safely.

Save a variable (such as a model) to a file, but in a safer way.

```
with open('varname', 'wb') as f_out:
  pickle.dump(varname, f_out)
```

- Using the `with` keyword there is no need to add the `close()` method. Once the code inside the `with` scope runs, the file will close. This makes the code both shorter and safer.

Load a file and assign its contents to a variable

```
with open('filename', 'rb') as f_in:
  var = pickle.load(f_in)
```

- Opening the file in `wb` mode as before would overwrite the previous file. For opening the file, we need to do so in `rb` mode (read binary).
- If the file contained a tuple, you may unpack it directly by assigning the load output to multiple variables: `dv, model = pickle.load(f_in)`

Flask

```
from flask import Flask, request, jsonify
```

Create a basic Flask service

```
app = Flask(name)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

- `app = Flask(name)` instances a Flask object. You may use `__name__` as a param most of the time, but you may define any string as the Flask app name.
- `@app.route("/")` is a ***decorator*** (a wrapper around a function that extends the function's behavior) that lets Flask know which URL to use for the app. In this example, the root URL `/` is used, so the `hello_world()` function will be called when we access `http://localhost/`. If it were `/app` instead, then the function would be called when accessing `http://localhost/app`.

Create a Flask service with an HTTP POST method

```
@app.route('/predict', methods=['POST'])
def my_function():
  # do stuff
```

- The `methods` param expects a list with the HTTP methods used by the function.

Capture the params that the client is sending you in the HTTP request.

- `my_var = request.get_json()`
  - `request.get_json()` will return a dictionary with all the params.

Create a JSON object to send to the client

- `jsonify(my_dictionary)`
  - `my_dictionary` is a dictionary struct, which is very similar to the JSON format.
  - `jsonify()` cannot convert numpy types automatically, so they need to be converted to native types:
    - `jsonifyable_float = float(numpy_float)`
    - `jsonifyable_bool = bool(numpy_bool)`

(Client side) Send a POST request to a server.

```
import requests

requests.post(url, json = my_dictionary).json()
```

- `url` is a string that contains the server's URL, like `0.0.0.0:9696/app_entry_point` or whatever.
- `my_dictionary` is a simple Python dictionary with the params you want to pass to the server.
- `requests.post().json()` will automatically convert the dictionary to JSON and send the POST request to the server.

dr563105 commented on May 23, 2022

Thanks!