bs$_d s Documentation$

Release 0.11.1

James Irving

Apr 24, 2020

Contents:


{BROAD : STEEL,
DATA : SCIENCE}

CHAPTER 1

Getting Started

To use bs_ds in a project:

```
# For best experience:
from bs_ds import *

# Recommended handle
import bs_ds as bs
```

To use bs_ds in an online learn.co lesson:

```
!pip install bs_ds
from bs_ds import *
```

For brief into to use bs_ds for EDA: https://drive.google.com/open?id=1B54tCSnKzsfIN3fHaJrp3oHIwQKJqmSi

CHAPTER 2

bs_ds package

Top-level package for bs_ds.

2.1 Submodules

2.2 bs_ds.bamboo module

Collection of DataFrame inspection, styling, and EDA plotting.

bs_ds.bamboo.**add_filtered_col_to_df**(*df_source*, *df_to_add_to*, *list_of_exps*, *return_filtered_col_names=False*)
Takes a dataframe source with columns to copy using df.filter(regexp=(list_of_exps)), with list_of_exps being a list of text expressions to find inside column names.

bs_ds.bamboo.**big_pandas**(*user_options=None*, *verbose=0*)
Changes the default pandas display setttings to show all columns and all rows. User may replace settings with a kwd dictionary matching available options.

bs_ds.bamboo.**check_column**(*panda_obj*, *columns=None*, *nlargest='all'*)
Prints column name, datatype, # and % of null values, and unique values for the nlargest # of rows (by **value-count_**. it will only print results for those columns ******** Params: panda_object: pandas DataFrame or Series columns: list containing names of columns (strings)

Returns: None prints values only

bs_ds.bamboo.**check_df_for_columns**(*df*, *columns=None*)
Checks df for presence of columns.

df: pd.DataFrame to find columns in columns: str or list of str. column names

bs_ds.bamboo.**check_null**(*df*, *columns=None*, *show_df=False*)
Iterates through columns and checks for null values and displays # and % of column. Params: ********** df: pandas DataFrame

columns: list of columns to check ******> Returns: displayed dataframe

bs_ds.bamboo.**check_numeric**(*df*, *columns=None*, *unique_check=False*, *return_list=False*, *show_df=False*)

Iterates through columns and checks for possible numeric features labeled as objects. Params: ********** df: pandas DataFrame

unique_check: bool. (default=True) If true, distplays interactive interface for checking unique values in columns.

return_list: bool, (default=False) If True, returns a list of column names with possible numeric types.

******> Returns: dataframe displayed (always), list of column names if return_list=True

bs_ds.bamboo.**check_unique**(*df*, *columns=None*)

Prints unique values for all columns in dataframe. If passed list of columns, it will only print results for those columns 8********** > Params: df: pandas DataFrame, or pd.Series columns: list containing names of columns (strings)

Returns: None prints values only

bs_ds.bamboo.**compare_duplicates**(*df1*, *df2*, *to_drop=True*, *verbose=True*, *return_names_list=False*)

Compare two dfs for duplicate columns, drop if to_drop=True, useful to us before concatenating when dtypes are different between matching column names and df.drop_duplicates is not an option. Params: ———— df1, df2 : pandas dataframe suspected of having matching columns to_drop : bool, (default=True)

If True will give the option of dropping columns one at a time from either column.

verbose: bool (default=True) If True prints column names and types, set to false and return_names list=True if only desire a list of column names and no interactive interface.

return_names_list: bool (default=False), If True, will return a list of all duplicate column names.

Returns: List of column names if return_names_list=True, else nothing.

bs_ds.bamboo.**detect_outliers**(*df*, *n*, *features*)

Uses Tukey's method to return outer of interquartile ranges to return indices if outliers in a dataframe. Parameters: df (DataFrame): DataFrame containing columns of features n: default is 0, multiple outlier cutoff

Returns: Index of outliers for .loc

Examples: Outliers_to_drop = detect_outliers(data,2,["col1","col2"]) Returning value df.loc[Outliers_to_drop] # Show the outliers rows data= data.drop(Outliers_to_drop, axis = 0).reset_index(drop=True)

bs_ds.bamboo.**drop_cols**(*df*, *list_of_strings_or_regexp*, *verbose=0*)

EDA: Take a df, a list of strings or regular expression and recursively removes all matching column names containing those strings or expressions. # Example: if the df_in columns are ['price','sqft','sqft_living','sqft15','sqft_living15','floors','bedrooms'] df_out = drop_cols(df_in, ['sqft','bedroom']) df_out.columns # will output: ['price','floors']

Parameters

- **--** (*regex_list*) – Input dataframe to remove columns from.
- **--** – list of string patterns or regexp to remove.

Returns df_dropped – input df without the dropped columns.

bs_ds.bamboo.**ignore_warnings**()

Ignores all deprecation warnings (future,and pending categories too).

bs_ds.bamboo.**inspect_df**(*df*, *n_rows=3*, *verbose=True*)

EDA: Show all pandas inspection tables. Displays df.head(), df.info(), df.describe(). By default also runs check_null and check_numeric to inspect columns for null values and to check string columns to detect numeric

values. (If verbose==True) :param df: dataframe to inspect :type df: dataframe :param n_rows: number of header rows to show (Default=3). :param verbose: If verbose==True (default), check_null and check_numeric.

Ex: inspect_df(df,n_rows=4)

bs_ds.bamboo.**multiplot**(*df*, *annot=True*, *fig_size=None*)
EDA: Plots results from df.corr() in a correlation heat map for multicollinearity. Returns fig, ax objects

bs_ds.bamboo.**plot_hist_scat**(*df*, *target=None*, *figsize=(12, 9)*, *fig_style='dark_background'*, *font_dict=None*, *plot_kwds=None*)
EDA: Great summary plots of all columns of a df vs target columne. Shows distplots and regplots for columns im datamframe vs target. :param df: DataFrame.describe() columns will be plotted. :type df: DataFrame :param target: Name of column containing target variable.assume first column. :type target: string :param figsize: Tuple for figsize. Default=(12,9). :type figsize: tuple :param fig_style: Figure style to use (in this context, will not change others in notebook).

Default is 'dark_background'.

Parameters **font_dict** – A keywork dictionry containing values for font properties under the following keys: - "fontTitle": font dictioanry for titles , fontAxis, fontTicks

****plot_kwds:** A kew_word dictionary containing any of the following keys for dictionaries containing any valid matplotlib key:value pairs for plotting:

"hist_kws, kde_kws, line_kws,scatter_kws"

Accepts any valid matplotlib key:value pairs passed by searborn to matplotlib. Subplot 1: hist_kws, kde_kws Subplot 2: line_kws,scatter_kws

Returns

Figure object. ax:

Subplot axes with format ax[row,col]. Subplot 1 = ax[0,0]; Subplot 2 = ax[0,1]

Return type fig

bs_ds.bamboo.**reset_pandas**()
Resets all pandas options back to default state.

bs_ds.bamboo.**reset_warnings**()
Restore the default warnings settings

2.3 bs_ds.bs_ds module

Main Moule of Data Pipelines and Data Transformation functions & classes.

class bs_ds.bs_ds.**LabelLibrary**
Bases: object

A Multi-column version of sklearn LabelEncoder, which fits a LabelEncoder to each column of a df and stores it in the index dictionary where .index[keyword=colname] returns the fit encoder object for that column.

Example: lib =LabelLibrary()

Be default, lib will fit all columns. lib.fit(df) # Can also specify columns lib.fit(df,columns=['A','B'])

Can then transform df_coded = lib.transform(df,['A','B']) # Can also use fit_transform df_coded = lib.fit_transform(df,columns=['A','B'])

lib.index contains each col's encoder by col name: col_a_classes = lib.index('A').classes_

__init__()
creates self.index and self.encoder

fit(*df*, *columns=None*)
Creates an encoder object and fits to each columns. Fit encoder is saved in the index dictionary by key=column_name

fit_transform(*df*, *columns=None*)

inverse_transform(*df*, *columns=None*)

transform(*df*, *columns=None*)

bs_ds.bs_ds.**column_report**(*df*, *index_col='iloc'*, *sort_column='iloc'*, *ascending=True*, *name_for_notes_col='Notes'*, *notes_by_dtype=False*, *decision_map=None*, *format_dict=None*, *as_qgrid=True*, *qgrid_options=None*, *qgrid_column_options=None*, *qgrid_col_defs=None*, *qgrid_callback=None*, *as_df=False*, *as_interactive_df=False*, *show_and_return=True*)
Returns a datafarme summary of the columns, their dtype, a summary dataframe with the column name, column dtypes, and a *decision_map* dictionary of datatype. [!] Please note if qgrid does not display properly, enter this into your terminal and restart your temrinal.

'jupyter nbextension enable –py –sys-prefix qgrid'# required for qgrid 'jupyter nbextension enable –py –sys-prefix widgetsnbextension' # only required if you have not enabled the ipywidgets nbextension yet

Default qgrid options:

default_grid_options={ # SlickGrid options 'fullWidthRows': True, 'syncColumnCellResize': True, 'forceFitColumns': True, 'defaultColumnWidth': 50, 'rowHeight': 25, 'enableColumnReorder': True, 'enableTextSelectionOnCells': True, 'editable': True, 'autoEdit': False, 'explicitInitialization': True,

Qgrid options 'maxVisibleRows': 30, 'minVisibleRows': 8, 'sortable': True, 'filterable': True, 'highlightSelectedCell': True, 'highlightSelectedRow': True

}

bs_ds.bs_ds.**convert_gdrive_url**(*share_url_from_gdrive*)
accepts gdrive share url with format 'https://drive.google.com/open?id=' and returns a pandas-usable link with format ''https://drive.google.com/uc?export=download&id='

bs_ds.bs_ds.**is_var**(*name*)

bs_ds.bs_ds.**list2df**(*list*, *index_col=None*, *set_caption=None*, *return_df=True*, *df_kwds={}*)
Quick turn an appened list with a header (row[0]) into a pretty dataframe.

Args list (list of lists): index_col (string): name of column to set as index; None (Default) has integer index. set_caption (string): show_and_return (bool):

EXAMPLE USE: >> list_results = [["Test","N","p-val"]]

... run test and append list of result values ...

>> list_results.append([test_Name,length(data),p])

Displays styled dataframe if caption: >> df = list2df(list_results, index_col="Test",

set_caption="Stat Test for Significance")

bs_ds.bs_ds.**make_gdrive_file_url**(*share_url_from_gdrive*)

bs_ds.bs_ds.**performance_r2_mse**(*y_true*, *y_pred*)
Calculates and returns the performance score between true and predicted values based on the metric chosen.

bs_ds.bs_ds.**performance_roc_auc**(*y_true*, *y_pred*)
Tests the results of an already-fit classifer. Takes y_true (test split), and y_pred (model.predict()), returns the AUC for the roc_curve as a %

bs_ds.bs_ds.**plot_confusion_matrix**(*cm*, *classes*, *normalize=False*, *title='Confusion matrix'*, *cmap=None*, *print_matrix=True*)
Check if Normalization Option is Set to True. If so, normalize the raw confusion matrix before visualizing #Other code should be equivalent to your previous function.

bs_ds.bs_ds.**plot_wide_kde_thin_mean_sem_bars**(*series1*, *sname1*, *series2*, *sname2*)
EDA / Hypothesis Testing: Two subplot EDA figure that plots series1 vs. series 2 against with sns.displot Large wide kde plot with small thing mean +- standard error of the mean (sem) Overlapping sem error bars is an excellent visual indicator of significant difference. .

bs_ds.bs_ds.**print_docstring_template**(*style='google'*, *object_type='function'*, *show_url=False*, *to_clipboard=False*)
Prints out docstring template for that is copy/paste ready. May choose 'google' or 'numpy' style docstrings and templates are available different types ('class','function','module_function').

Parameters

- **style** (*str, optional*) – Which docstring style to return. Options are 'google' and 'numpy'. Defaults to 'google'.
- **object_type** (*str, optional*) – Which type of template to return. Options are 'class','function','module_function'. Defaults to 'function'.
- **show_url** (*bool, optional*) – Whether to display link to reference page for style-type. Defaults to False.

Returns [description]

Return type [type]

bs_ds.bs_ds.**subplot_imshow**(*images*, *num_images*, *num_rows*, *num_cols*, *figsize=(20, 15)*)
Takes image data and plots a figure with subplots for as many images as given.

images: str, data in form data.images ie. olivetti images num_images: int, number of images num_rows: int, number of rows to plot. num_cols: int, number of columns to plot figize: tuple, size of figure default=(20,15)

returns: figure with as many subplots as images given

bs_ds.bs_ds.**tune_params_trees**(*param_name*, *param_values*, *DecisionTreeObject*, *X*, *Y*, *test_size=0.25*, *perform_metric='r2_mse'*)

Test Decision Tree Regressor or Classifier parameter with the values in param_values Displays color-coed dataframe of perfomance results and subplot line graphs.

Parameters

- **parame_name** (*str*) – name of parameter to test with values param_values
- **param_values** (*list/array*) – list of parameter values to try
- **DecisionTreeObject,** – Existing DecisionTreeObject instance.
- **perform_metric** – Can either 'r2_mse' or 'roc_auc'.

Returns: - df of results - Displays styled-df - Displays subplots of performance metrics.

bs_ds.bs_ds.**viz_tree**(*tree_object*)
Takes a Sklearn Decision Tree and returns a png image using graph_viz and pydotplus.

2.4 bs_ds.capstone module

class bs_ds.capstone.**BlockTimeSeriesSplit**(*n_splits=5*, *train_size=None*, *test_size=None*, *step_size=None*, *method='sliding'*)
Bases: sklearn.model_selection._split._BaseKFold

A variant of sklearn.model_selection.TimeSeriesSplit that keeps train_size and test_size constant across folds. Requires n_splits,train_size,test_size. train_size/test_size can be integer indices or float ratios

split(*X*, *y=None*, *groups=None*)
Generate indices to split data into training and test set.

Parameters

- **X** (*array-like, shape (n_samples, n_features)*) – Training data, where n_samples is the number of samples and n_features is the number of features.
- **y** (*array-like, shape (n_samples,)*) – The target variable for supervised learning problems.
- **groups** (*array-like, with shape (n_samples,), optional*) – Group labels for the samples used while splitting the dataset into train/test set.

Yields

- **train** (*ndarray*) – The training set indices for that split.
- **test** (*ndarray*) – The testing set indices for that split.

bs_ds.capstone.**adf_test**(*series*, *title=''*)
Pass in a time series and an optional title, returns an ADF report # UDEMY COURSE ALTERNATIVE TO STATIONARITY CHECK

bs_ds.capstone.**apply_stopwords**(*stopwords_list*, *text*, *tokenize=True*, *return_tokens=False*, *pattern="([a-zA-Z]+(?:'[a-z]+)?)"*)
EX: df['text_stopped'] = df['content'].apply(lambda x: apply_stopwords(stopwords_list,x))

bs_ds.capstone.**arr2series**(*array*, *series_index=[]*, *series_name='predictions'*)
Accepts an array, an index, and a name. If series_index is longer than array: the series_index[-len(array):]

bs_ds.capstone.**auto_filename_time**(*prefix=''*, *sep='_'*, *suffix=''*, *ext=''*, *fname_friendly=True*, *timeformat='%m-%d-%Y %T'*)
Generates a filename with a base string + sep+ the current datetime formatted as timeformat. filename = f"{prefix}{sep}{suffix}{sep}{timesuffix}{ext}

bs_ds.capstone.**bin_df_by_date_intervals**(*test_df*, *time_intervals*, *column='date'*)
Uses pd.cut with half_hour_intervals on specified column. Creates a dictionary/map of integer bin codes. Adds column"int_bins" with int codes. Adds column "left_edge" as datetime object representing the beginning of the time interval. Returns the updated test_df and a list of bin_codes.

bs_ds.capstone.**case_ratio**(*msg*)
Accepts a twitter message (or used with .apply(lambda x:)). Returns the ratio of capitalized characters out of the total number of characters.

EX: df['case_ratio'] = df['text'].apply(lambda x: case_ratio(x))

bs_ds.capstone.**check_class_balance**(*df*, *col='delta_price_class_int'*, *note=''*, *as_percent=True*, *as_raw=True*)

bs_ds.capstone.**check_df_groups_for_exp**(*df_full*, *list_of_exp_to_check*, *check_col='content_min_clean'*, *groupby_col='troll_tweet'*, *group_dict={0: 'Control', 1: 'Troll'}*)
Checks *check_col* column of input dataframe for expressions in list_of_exp_to_check and counts the # present for each group, defined by the groupby_col and groupdict. Returns a dataframe of counts.

bs_ds.capstone.**check_null_small**(*df*, *null_index_column=None*)

bs_ds.capstone.**check_null_times**(*x*)

bs_ds.capstone.**collapse_df_by_group_index_col**(*twitter_df*, *group_index_col='int_bins'*, *new_col_order=None*)
Loops through the group_indices provided to concatenate each group into a single row and combine into one dataframe with the _____ as the index

bs_ds.capstone.**compare_word_cloud**(*text1*, *label1*, *text2*, *label2*)
Compares the wordclouds from 2 sets of texts

bs_ds.capstone.**concatenate_group_data**(*group_df_or_series*)
Accepts a series or dataframe from a groupby.get_group() loop. Adds TweetFreq column for # of rows concatenate. If input is series, TweetFreq=1 and series is returned.

bs_ds.capstone.**create_required_folders**(*full_filenamepath*, *folder_delim='/'*, *verbose=1*)
Accepts a full file name path include folders with '/' as default delimiter. Recursively checks for all sub-folders in filepath and creates those that are missing.

bs_ds.capstone.**custom_BH_freq**()

bs_ds.capstone.**def_cufflinks_solar_theme**(*as_layout=True*, *as_dict=False*)

bs_ds.capstone.**def_plotly_date_range_widgets**(*my_rangeselector=None*, *as_layout=True*, *as_dict=False*)
old name; def_my_plotly_stock_layout, REPLACES DEF_RANGE_SELECTOR

bs_ds.capstone.**def_plotly_solar_theme_with_date_selector_slider**(*as_layout=True*, *as_dict=False*)

bs_ds.capstone.**dict_dropdown**(*dict_to_display*, *title='Dictionary Contents'*)
Display the model_params dictionary as a dropdown menu.

bs_ds.capstone.**disp_df_head_tail**(*df*, *n_head=3*, *n_tail=3*, *head_capt='df.head'*, *tail_capt='df.tail'*)
Displays the df.head(n_head) and df.tail(n_tail) and sets captions using df.style

bs_ds.capstone.**display_df_dict_dropdown**(*dict_to_display*, *selected_key=None*)

bs_ds.capstone.**display_dict_dropdown**(*dict_to_display*)
Display the model_params dictionary as a dropdown menu.

bs_ds.capstone.**display_random_tweets**(*df_tokenize*, *n=5*, *display_cols=['content', 'text_for_vectors', 'tokens']*, *group_labels=[]*, *verbose=True*)
Takes df_tokenize['text_for_vectors']

bs_ds.capstone.**empty_lists_to_strings**(*x*)
Takes a series and replaces any empty lists with an empty string instead.

bs_ds.capstone.**evaluate_classification_model**(*model, X_train, X_test, y_train, y_test, history=None, binary_classes=True, conf_matrix_classes=['Decrease', 'Increase'], normalize_conf_matrix=True, conf_matrix_figsize=(8, 4), save_history=False, history_filename='results/keras_history.png', save_conf_matrix_png=False, conf_mat_filename='results/confusion_matrix.png', save_summary=False, summary_filename='results/model_summary.txt', auto_unique_filenames=True*)

Evaluates kera's model's performance, plots model's history,displays classification report, and plots a confusion matrix. conf_matrix_classes are the labels for the matrix. [negative, positive] Returns df of classification report and fig object for confusion matrix's plot.

bs_ds.capstone.**evaluate_regression**(*y_true*, *y_pred*, *metrics=None*, *show_results=False*, *display_thiels_u_info=False*)

Calculates and displays any of the following evaluation metrics: (passed as strings in metrics param) r2, MAE,MSE,RMSE,U if metrics=None:

metrics=['r2','RMSE','U']

bs_ds.capstone.**evaluate_regression_model**(*model*, *history*, *train_generator*, *test_generator*, *true_train_series*, *true_test_series*, *include_train_data=True*, *return_preds_df=False*, *save_history=False*, *history_filename='results/keras_history.png'*, *save_summary=False*, *summary_filename='results/model_summary.txt'*, *auto_unique_filenames=True*)

Evaluates kera's model's performance, plots model's history,displays classification report, and plots a confusion matrix. conf_matrix_classes are the labels for the matrix. [negative, positive] Returns df of classification report and fig object for confusion matrix's plot.

bs_ds.capstone.**find_null_idx**(*df*, *column=None*)

returns the indices of null values found in the series/column. if df is a dataframe and column is none, it returns a dictionary with the column names as a value and null_idx for each column as the values. Example Usage: 1) >> null_idx = get_null_idx(series) >> series_null_removed = series[null_idx] 2) >> null_dict = get_null_idx()

bs_ds.capstone.**full_sentiment_analysis**(*twitter_df*, *source_column='content_min_clean'*, *separate_cols=True*)

bs_ds.capstone.**full_twitter_df_processing**(*df*, *raw_tweet_col='content'*, *cleaned_tweet_col='content'*, *case_ratio_col='content_min_clean'*, *sentiment_analysis_col='content_min_clean'*, *RT=True*, *urls=True*, *hashtags=True*, *mentions=True*, *str_tags_mentions=True*, *stopwords_list=[]*, *force=False*)

Accepts df_full, which contains the raw tweets to process, the raw_col name, the column to fill. If force=False, returns error if the fill_content_col already exists. Processing Workflow:1) Create has_RT, starts_RT columns. 2) Creates [fill_content_col,'content_min_clean'] cols after removing 'RT @mention:' and urls. 3) Removes hashtags from fill_content_col and saves hashtags in new col. 4) Removes mentions from fill_content_col and saves to new column.

bs_ds.capstone.**get_B_day_time_index_shift**(*test_df*, *verbose=1*)

bs_ds.capstone.**get_day_window_size_from_freq**(*dataset*, *CBH=<CustomBusinessHour: CBH=09:30-16:30>*)

bs_ds.capstone.**get_group_sentiment_scores**(*df*, *score_col='sentiment_scores'*)

bs_ds.capstone.**get_group_texts_tokens**(*df_small*, *groupby_col='troll_tweet'*, *group_dict={0: 'controls', 1: 'trolls'}*, *column='content_stopped'*)

bs_ds.capstone.**get_model_config_df**(*model1*, *multi_index=True*)

bs_ds.capstone.**get_model_preds_from_gen**(*model*, *test_generator*, *true_test_data*, *model_params=None*, *n_input=None*, *n_features=None*, *suffix=None*, *verbose=0*, *return_df=True*)

Gets prediction from model using the generator's timeseries using model.predict_generator() Must provide a model_params dictionary with 'input_params' OR must define ('n_input','n_features').

bs_ds.capstone.**get_source_code_markdown**(*function*)

Retrieves the source code as a string and appends the markdown python syntax notation

bs_ds.capstone.**get_stock_prices_for_twitter_data**(*twitter_df*, *stock_prices*)

bs_ds.capstone.**get_technical_indicators**(*dataset*, *make_price_from='BidClose'*)

bs_ds.capstone.**get_time**(*timeformat='%m-%d-%y_%T%p'*, *raw=False*, *filename_friendly=False*, *replacement_seperator='-'*)

Gets current time in local time zone. if raw: True then raw datetime object returned without formatting. if filename_friendly: replace ':' with replacement_separator

bs_ds.capstone.**get_true_vs_model_pred_df**(*model*, *n_input*, *test_generator*, *test_data_index*, *df_test*, *train_generator*, *train_data_index*, *df_train*, *scaler=None*, *inverse_tf=True*, *plot=True*, *verbose=2*)

Accepts a model, the training and testing data TimeseriesGenerators, the test_index and train_index. Returns a dataframe with True and Predicted Values for Both the Training and Test Datasets.

bs_ds.capstone.**ihelp**(*function_or_mod*, *show_help=True*, *show_code=True*, *return_code=False*, *markdown=True*, *file_location=False*)

Call on any module or functon to display the object's help command printout AND/OR soruce code displayed as Markdown using Python-syntax

bs_ds.capstone.**ihelp_menu**(*function_names*, *show_help=False*, *show_source=True*)

Accepts a list of functions or function_names as strings. if show_help: display *help(function.* if show_source: retreive source code and display as proper markdown syntax

bs_ds.capstone.**index_report**(*df*, *label=''*, *time_fmt='%Y-%m-%d %T'*, *return_index_dict=False*)

Sorts dataframe index, prints index's start and end points and its datetime frequency. if return_index_dict=True then it returns these values in a dictionary as well as printing them.

bs_ds.capstone.**inspect_variables**(*local_vars=None*, *sort_col='size'*, *exclude_funcs_mods=True*, *top_n=10*, *return_df=False*, *always_display=True*, *show_how_to_delete=True*, *print_names=False*)

Displays a dataframe of all variables and their size in memory, with the largest variables at the top.

bs_ds.capstone.**int_to_ts**(*int_list*, *as_datetime=False*, *as_str=True*)

Accepts one Panda's interval and returns the left and right ends as either strings or Timestamps.

bs_ds.capstone.**inverse_transform_series**(*series*, *scaler*)

Takes a series of df column and a fit scaler. Intended for use with make_scaler_library's dictionary Example Usage: scaler_lib, df_scaled = make_scaler_library(df, transform = True) series_inverse_transformed = inverse_transform_series(df['price_data'],scaler_lib['price'])

bs_ds.capstone.**is_var**(*name*)

bs_ds.capstone.**load_model_weights_params**(*base_filename='models/model_'*, *load_model_params=True*, *load_model_layers_excel=True*, *trainable=False*, *model_filename=None*, *weight_filename=None*, *model_params_filename=None*, *excel_filename=None*, *verbose=1*)

Loads in Keras model from json file and loads weights from .h5 file. optional set model layer trainability to False

bs_ds.capstone.**load_raw_stock_data_from_txt**(*filename='IVE_bidask1min.txt'*, *folderpath='data/'*, *start_index='2016-12-31'*, *clean=True*, *fill_or_drop_null='drop'*, *fill_method='ffill'*, *freq='CBH'*, *verbose=2*)

bs_ds.capstone.**load_raw_twitter_file**(*filename='data/trump_tweets_01202017_06202019.csv'*, *date_as_index=True*, *rename_map={'created_at': 'date', 'text': 'content'}*)

bs_ds.capstone.**load_stock_df_from_csv**(*filename='ive_sp500_min_data_match_twitter_ts.csv'*, *folderpath='/content/drive/My Drive/Colab Notebooks/Mod 5 Project/data/'*, *clean=True*, *freq='T'*, *fill_method='ffill'*, *verbose=2*)

bs_ds.capstone.**load_stock_price_series**(*filename='IVE_bidask1min.txt'*, *folderpath='data/'*, *start_index='2017-01-23'*, *freq='T'*)

bs_ds.capstone.**load_twitter_df**(*overwrite=True*, *set_index='time_index'*, *verbose=2*, *replace_na=''*)

bs_ds.capstone.**load_twitter_df_stock_price**(*twitter_df*, *stock_price=None*)

bs_ds.capstone.**make_X_y_timeseries_data**(*data*, *x_window=35*, *verbose=2*, *as_array=True*)

Creates an X and Y time sequence trianing set from a pandas Series. - X_train is a an array with x_window # of samples for each row in X_train - y_train is one value per X_train window: the next time point after the X_window. Verbose determines details printed about the contents and shapes of the data.

Example Usage: X_train, y_train = make_X_y_timeseries(df[‘price’], x_window= 35) print(X_train[0]]): # returns: arr[X1,X2…X35] print(y_train[0]) # returns X36

bs_ds.capstone.**make_date_range_slider**(*start_date*, *end_date*, *freq='D'*)

bs_ds.capstone.**make_df_timeseries_bins_by_column**(*df*, *x_window=35*, *verbose=2*, *one_or_two_dfs=1*)

Function will take each column from the dataframe and create a train_data dataset (with X and Y data), with each row in X containing x_window number of observations and y containing the next following observation

bs_ds.capstone.**make_model_menu**(*model1*, *multi_index=True*)

bs_ds.capstone.**make_qgrid_model_menu**(*model*, *return_df=False*)

bs_ds.capstone.**make_scaler_library**(*df*, *transform=False*, *columns=[]*)

Takes a df and fits a MinMax scaler to the columns specified (default is to use all columns). Returns a dictionary (scaler_library) with keys = columns, and values = its corresponding fit’s MinMax Scaler

Example Usage: scale_lib, df_scaled = make_scaler_library(df, transform=True)

to get the inverse_transform of a column with a different name: # use *inverse_transform_series* scaler = scale_lib[‘price’] # get scaler fit to original column of interest price_column = inverse_transform_series(df[‘price_labels’], scaler) #get the inverse_transformed series back

bs_ds.capstone.**make_stopwords_list**(*incl_punc=True, incl_nums=True, add_custom=['http', 'https', '...', '…', '‘‘', 'co', '“', "’", '‘', '”', "n't", "’’", 'u', 's', "'s", '|', '\\|', 'amp', "i'm"]*)

bs_ds.capstone.**make_time_index_intervals**(*twitter_df*, *col='date'*, *start=None*, *end=None*, *freq='CBH'*, *num_offset=1*)

Takes a df, rounds first timestamp down to nearest hour, last timestamp rounded up to hour. Creates 30 minute intervals based that encompass all data.

bs_ds.capstone.**match_data_colors**(*fig1*, *fig2*)

bs_ds.capstone.**match_stock_price_to_tweets**(*tweet_timestamp*, *time_after_tweet=30*, *time_freq='T'*, *stock_price=[]*)

bs_ds.capstone.**my_rmse**(*y_true*, *y_pred*)

RMSE calculation using keras.backend

bs_ds.capstone.**open_image_mask**(*filename*)

bs_ds.capstone.**plot_auc_roc_curve**(*y_test*, *y_test_pred*)

Takes y_test and y_test_pred from a ML model and plots the AUC-ROC curve.

bs_ds.capstone.**plot_confusion_matrix**(*conf_matrix*, *classes=None*, *normalize=False*, *title='Confusion Matrix'*, *cmap=None*, *print_raw_matrix=False*, *fig_size=(5, 5)*, *show_help=False*)

Check if Normalization Option is Set to True. If so, normalize the raw confusion matrix before visualizing #Other code should be equivalent to your previous function. Note: Taken from bs_ds and modified

bs_ds.capstone.**plot_decomposition**(*TS*, *decomposition*, *figsize=(12, 8)*, *window_used=None*)

Plot the original data and output decomposed components

bs_ds.capstone.**plot_fit_cloud**(*troll_cloud*, *contr_cloud*, *label1='Troll'*, *label2='Control'*)

bs_ds.capstone.**plot_keras_history**(*history*, *title_text=''*, *fig_size=(6, 6)*, *save_fig=False*, *no_val_data=False*, *filename_base='results/keras_history'*)

Plots the history['acc','val','val_acc','val_loss']

bs_ds.capstone.**plot_technical_indicators**(*dataset*, *last_days=90*)

bs_ds.capstone.**plot_time_series**(*stocks_df*, *freq=None*, *fill_method='ffill'*, *figsize=(12, 4)*)

bs_ds.capstone.**plot_true_vs_preds_subplots**(*train_price*, *test_price*, *pred_price*, *subplots=False*, *verbose=0*, *figsize=(12, 5)*)

bs_ds.capstone.**plotly_time_series**(*stock_df*, *x_col=None*, *y_col=None*, *layout_dict=None*, *title='S&P500 Hourly Price'*, *theme='solar'*, *as_figure=True*, *show_fig=True*, *fig_dim=(900, 400)*, *iplot_kwargs=None*)

bs_ds.capstone.**plotly_true_vs_preds_subplots**(*df_model_preds*, *true_train_col='true_train_price'*, *true_test_col='true_test_price'*, *pred_test_columns='pred_from_gen'*, *subplot_mode='lines+markers'*, *marker_size=5*, *title='S&P 500 True Price Vs Predictions ($)'*, *theme='solar'*, *verbose=0*, *figsize=(1000, 500)*, *debug=False*, *show_fig=True*)

y_col_kws={'col_name':line_color}

bs_ds.capstone.**predict_model_make_results_dict**(*model*, *scaler*, *X_test_in*, *y_test*, *test_index*, *X_train_in*, *y_train*, *train_index*, *return_as_dfs=False*)
Accepts a fit keras model, X_test, y_test, and y_train data. Uses provided fit-scaler that transformed original data. By default (return_as_dfs=False): returns the results as a panel (dictioanry of dataframes), with panel['train'],panl['test'] Setting return_as_dfs=True will return df_train, df_test

bs_ds.capstone.**preview_dict**(*d*, *n=5*, *print_or_menu='print'*, *return_list=False*)
Previews the first n keys and values from the dict

bs_ds.capstone.**print_array_info**(*X*, *name='Array'*)
Test function for verifying shapes and data ranges of input arrays

bs_ds.capstone.**quick_ref_pandas_freqs**()

bs_ds.capstone.**quick_table**(*tuples*, *col_names=None*, *caption=None*, *display_df=True*)
Accepts a bigram output tuple of tuples and makes captioned table.

bs_ds.capstone.**reload**(*mod*)
Reloads the module from file. Example: import my_functions_from_file as mf # after editing the source file: # mf.reload(mf)

bs_ds.capstone.**reorder_twitter_df_columns**(*twitter_df*, *order=[]*)

bs_ds.capstone.**replace_bad_filename_chars**(*filename*, *replace_spaces=False*, *replace_with='_'*)
removes any characters not allowed in Windows filenames

bs_ds.capstone.**save_ihelp_menu_to_file**(*function_list*, *filename*, *save_help=False*, *save_code=True*, *folder='readme_resources/ihelp_outputs/'*, *as_md=True*, *as_txt=False*, *verbose=1*)
Accepts a list of functions and uses save_ihelp_to_file with mode='a' to combine all outputs. Note: this function REQUIRES a filename

bs_ds.capstone.**save_ihelp_to_file**(*function*, *save_help=False*, *save_code=True*, *as_md=False*, *as_txt=True*, *folder='readme_resources/ihelp_outputs/'*, *filename=None*, *file_mode='w'*)
Saves the string representation of the ihelp source code as markdown. Filename should NOT have an extension. .txt or .md will be added based on as_md/as_txt. If filename is None, function name is used.

bs_ds.capstone.**save_model_weights_params**(*model*, *model_params=None*, *filename_prefix='models/model'*, *filename_suffix=''*, *check_if_exists=True*, *auto_increment_name=True*, *auto_filename_suffix=True*, *save_model_layer_config_xlsx=True*, *sep='_'*, *suffix_time_format='%m-%d-%Y_%I%M%p'*)
Saves a fit Keras model and its weights as a .json file and a .h5 file, respectively. auto_filename_suffix will use the date and time to give the model a unique name (avoiding overwrites). Returns the model_filename and weight_filename

bs_ds.capstone.**seasonal_decompose_and_plot**(*ive_df*, *col='BidClose'*, *freq='H'*, *fill_method='ffill'*, *window=144*, *model='multiplicative'*, *two_sided=False*, *plot_components=True*)
Perform seasonal_decompose from statsmodels.tsa.seasonal. Plot Output Decomposed Components

bs_ds.capstone.**set_timeindex_freq**(*ive_df*, *col_to_fill=None*, *freq='CBH'*, *fill_method='ffill'*, *verbose=3*)

bs_ds.capstone.**show_del_me_code**(*called_by_inspect_vars=False*)
Prints code to copy and paste into a cell to delete vars using a list of their names. Companion function inspect_variables(locals(),print_names=True) will provide var names tocopy/paste

bs_ds.capstone.**show_random_img**(*image_array*, *n=1*)
Display n rendomly-selected images from image_array

bs_ds.capstone.**stationarity_check**(*df*, *col='BidClose'*, *window=80*, *freq='BH'*)
From learn.co lesson: use ADFuller Test for Stationary and Plot

bs_ds.capstone.**thiels_U**(*ys_true=None*, *ys_pred=None*, *display_equation=True*, *display_table=True*)
Calculate's Thiel's U metric for forecasting accuracy. Accepts true values and predicted values. Returns Thiel's U

bs_ds.capstone.**train_test_split_by_last_days**(*stock_df*, *periods_per_day=7*, *num_test_days=90*, *num_train_days=180*, *verbose=1*, *plot=True*)
Takes the last num_test_days of the time index to use as testing data, and take shte num_Trian_days prior to that date as the training data.

bs_ds.capstone.**train_test_val_split**(*X*, *y*, *test_size=0.2*, *val_size=0.1*)
Performs 2 successive train_test_splits to produce a training, testing, and validation dataset

bs_ds.capstone.**transform_cols_from_library**(*df*, *scaler_library*, *inverse=False*, *columns=[]*)
Accepts a df and a scaler_library that was transformed using make_scaler_library. Inverse tansforms listed columns (if columns =[] then all columns) Returns a dataframe with all columns of original df.

bs_ds.capstone.**transform_image_mask_white**(*val*)
Will convert any pixel value of 0 (white) to 255 for wordcloud mask.

bs_ds.capstone.**twitter_column_report**(*twitter_df*, *decision_map=None*, *sort_column=None*, *ascending=True*, *interactive=True*)

bs_ds.capstone.**undersample_df_to_match_classes**(*df*, *class_column='delta_price_class'*, *class_values_to_keep=None*, *verbose=1*)
Resamples (undersamples) input df so that the classes in class_column have equal number of occruances. If class_values_to_keep is None: uses all classes.

bs_ds.capstone.**unpack_match_stocks**(*stock_dict*)

2.5 bs_ds.glassboxes module

A collection of modified tools to visualize the inner-workings of model objects, especially Catboot Models.

class bs_ds.glassboxes.**Clock**(*display_final_time_as_minutes=True*, *verbose=2*)
Bases: object

A clock meant to be used as a timer for functions using local time. Clock.tic() starts the timer, .lap() adds the current laps time to clock._list_lap_times, .toc() stops the timer. If user initiializes with verbose =0, only start and final end times are displays.

If verbose=1, print each lap's info at the end of each lap. If verbose=2 (default, display instruction line, return datafarme of results.)

class datetime(*year*, *month*, *day*[, *hour*[, *minute*[, *second*[, *microsecond*[, *tzinfo*]]]]])
Bases: datetime.date

The year, month and day arguments are required. tzinfo may be None, or an instance of a tzinfo subclass. The remaining arguments may be ints.

astimezone()
tz -> convert to local time in new timezone tz

combine()
date, time -> datetime with same date and time fields

ctime()
Return ctime() style string.

date()
Return date object with same year, month and day.

dst()
Return self.tzinfo.dst(self).

fold

fromisoformat()
string -> datetime from datetime.isoformat() output

fromtimestamp()
timestamp[, tz] -> tz's local time from POSIX timestamp.

hour

isoformat()
[sep] -> string in ISO 8601 format, YYYY-MM-DDT[HH[:MM[:SS[.mmm[uuu]]]]][+HH:MM]. sep is used to separate the year from the time, and defaults to 'T'. timespec specifies what components of the time to include (allowed values are 'auto', 'hours', 'minutes', 'seconds', 'milliseconds', and 'microseconds').

max = datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)

microsecond

min = datetime.datetime(1, 1, 1, 0, 0)

minute

now()
Returns new datetime object representing current time local to tz.
tz Timezone object.
If no tz is specified, uses local timezone.

replace()
Return datetime with new specified fields.

resolution = datetime.timedelta(microseconds=1)

second

strptime()
string, format -> new datetime parsed from a string (like time.strptime()).

time()
Return time object with same time but with tzinfo=None.

timestamp()
Return POSIX timestamp as float.

timetuple()
Return time tuple, compatible with time.localtime().

timetz()
Return time object with same time and tzinfo.

tzinfo

tzname()
Return self.tzinfo.tzname(self).

utcfromtimestamp()
Construct a naive UTC datetime from a POSIX timestamp.

utcnow()
Return a new datetime representing UTC day and time.

utcoffset()
Return self.tzinfo.utcoffset(self).

utctimetuple()
Return UTC time tuple, compatible with time.localtime().

get_localzone()
Get the computers configured local timezone, if any.

get_time(*local=True*)
Returns current time, in local time zone by default (local=True).

lap(*label=None*)
Records time, duration, and label for current lap. Output display varies with clock verbose level. Calls .mark_lap_list() to document results in clock._list_lap_ times.

list2df(*index_col=None*, *set_caption=None*, *return_df=True*, *df_kwds={}*)
Quick turn an appened list with a header (row[0]) into a pretty dataframe.

Args list (list of lists): index_col (string): name of column to set as index; None (Default) has integer index. set_caption (string): show_and_return (bool):

EXAMPLE USE: >> list_results = [[“Test”,”N”,”p-val”]]

. . . run test and append list of result values . . .

>> list_results.append([test_Name,length(data),p])

Displays styled dataframe if caption: >> df = list2df(list_results, index_col=”Test”,

set_caption=”Stat Test for Significance”)

mark_lap_list(*label=None*)
Used internally, appends the current laps’ information when called by .lap() self._lap_times_list_ = [[‘Lap #’ , ‘Start Time’,’Stop Time’, ‘Stop Label’, ‘Duration’]]

summary()
Display dataframe summary table of Clock laps

tic(*label=None*)
Start the timer and display current time, appends label to the _list_lap_times.

timezone()
Return a datetime.tzinfo implementation for the given timezone

```
>>> from datetime import datetime, timedelta
>>> utc = timezone('UTC')
>>> eastern = timezone('US/Eastern')
>>> eastern.zone
'US/Eastern'
```

(continues on next page)

(continued from previous page)

```
>>> timezone(unicode('US/Eastern')) is eastern
True
>>> utc_dt = datetime(2002, 10, 27, 6, 0, 0, tzinfo=utc)
>>> loc_dt = utc_dt.astimezone(eastern)
>>> fmt = '%Y-%m-%d %H:%M:%S %Z (%z)'
>>> loc_dt.strftime(fmt)
'2002-10-27 01:00:00 EST (-0500)'
>>> (loc_dt - timedelta(minutes=10)).strftime(fmt)
'2002-10-27 00:50:00 EST (-0500)'
>>> eastern.normalize(loc_dt - timedelta(minutes=10)).strftime(fmt)
'2002-10-27 01:50:00 EDT (-0400)'
>>> (loc_dt + timedelta(minutes=10)).strftime(fmt)
'2002-10-27 01:10:00 EST (-0500)'
```

Raises UnknownTimeZoneError if passed an unknown zone.

```
>>> try:
...     timezone('Asia/Shangri-La')
... except UnknownTimeZoneError:
...     print('Unknown')
Unknown
```

```
>>> try:
...     timezone(unicode('\N{TRADE MARK SIGN}'))
... except UnknownTimeZoneError:
...     print('Unknown')
Unknown
```

toc(*label=None*, *summary=True*)
Stop the timer and displays results, appends label to final _list_lap_times entry

bs_ds.glassboxes.**make_activations_model**(*model*, *idx_layers_to_show=None*, *verbose=True*)
Accepts a Keras image convolution model and exports a new model, with just the intermediate activations to plot with plot_activations().

bs_ds.glassboxes.**plot_activations**(*activations_model*, *img_tensor*, *n_cols=16*, *process=True*, *colormap='viridis'*)
Accepts an activations_model from make_activations_model. Plots all channels' outputs for every image layer in the model.

bs_ds.glassboxes.**plot_auc_roc_curve**(*y_test*, *y_test_pred*)
Takes y_test and y_test_pred from a ML model and uses sklearn roc_curve to plot the AUC-ROC curve.

bs_ds.glassboxes.**plot_cat_feature_importances**(*cb_clf*)
Accepts a fitted CatBoost classifier model and plots the feature importances as a bar chart. Returns the results as a Series.

bs_ds.glassboxes.**plot_confusion_matrix**(*cm*, *classes*, *normalize=False*, *title='Confusion matrix'*, *cmap=None*, *print_matrix=True*)
Check if Normalization Option is Set to True. If so, normalize the raw confusion matrix before visualizing #Other code should be equivalent to your previous function.

bs_ds.glassboxes.**viz_tree**(*tree_object*)
Takes a Sklearn Decision Tree and returns a png image using graph_viz and pydotplus.

2.6 bs_ds.importSklearn module

2.7 bs_ds.imports module

Convience module. 'from bs_ds.imports import *' will pre-load pd,np,plt,mpl,sns

bs_ds.imports.**import_packages**(*import_list_of_tuples=None*, *display_table=True*)

Uses the exec function to load in a list of tuples with: [('module','md','example generic tuple item')] formatting. >> Default imports_list: [('pandas', 'pd', 'High performance data structures and tools'), ('numpy', 'np', 'scientific computing with Python'), ('matplotlib', 'mpl', "Matplotlib's base OOP module with formatting artists"), ('matplotlib.pyplot', 'plt', "Matplotlib's matlab-like plotting module"), ('seaborn', 'sns', "High-level data visualization library based on matplotlib"), ('bs_ds','bs','Custom data science bootcamp student package')]

2.8 bs_ds.prettypandas module

A collection of function to change the aesthetics of Pandas DataFrames using CSS, html, and pandas styling.

bs_ds.prettypandas.**color_scale_columns**(*df*, *matplotlib_cmap='Greens'*, *subset=None*)

DataFrame Styler: Takes a df, any valid matplotlib colormap column names (matplotlib.org/tutorials/colors/colormaps.html) and returns a dataframe with a gradient colormap applied to column values.

Example: df_styled = color_scale_columns(df,cmap = "YlGn",subset=['Columns','to','color'])

df: DataFrame containing columns to style.

subset: Names of columns to color-code.

cmap: Any matplotlib colormap. https://matplotlib.org/tutorials/colors/colormaps.html

df_style: styled dataframe.

bs_ds.prettypandas.**color_true_green**(*val*)

DataFrame Styler: Changes text color to green if value is True Ex: style_df = df.style.applymap(color_true_green)

style_df #to display

bs_ds.prettypandas.**display_side_by_side**(**args*)

Display all input dataframes side by side. Also accept captioned styler df object (df_in = df.style.set_caption('caption') Modified from Source: https://stackoverflow.com/questions/38783027/jupyter-notebook-display-two-pandas-tables-side-by-side

bs_ds.prettypandas.**highlight**(*df*, *hover_color='gold'*)

DataFrame Styler: Highlight row when hovering. Accept and valid CSS colorname as hover_color.

bs_ds.prettypandas.**hover**(*hover_color='gold'*)

DataFrame Styler: Called by highlight to highlight row below cursor. Changes html background color.

Parameters:

hover_Color

bs_ds.prettypandas.**html_off**()

bs_ds.prettypandas.**html_on**(*CSS=None*, *verbose=False*)
Applies HTML/CSS styling to all dataframes. 'CSS' variable is created by make_CSS() if not supplied. Verbose =True will display the default CSS code used. Any valid CSS key: value pair can be passed.

bs_ds.prettypandas.**make_CSS**(*show=False*)

2.9 bs_ds.saycheese module

A collection of image processing tools

bs_ds.saycheese.**show_random_img**(*image_array*, *n=1*)
Display n rendomly-selected images from image_array

2.10 bs_ds.saywhat module

A collection of language processing tools.

class bs_ds.saywhat.**W2vVectorizer**(*w2v*, *glove*)
Bases: object

From Learn.co Text Classification with Word Embeddings Lab. An sklearn-comaptible class containing the vectors for the fit Word2Vec.

fit(*X*, *y*)

transform(*X*)

bs_ds.saywhat.**connect_twitter_api**(*api_key*, *api_secret_key*)
Use tweepy to connect to the twitter-API and return tweepy api object.

bs_ds.saywhat.**make_stopwords**(*punctuation=True*)
Makes and returns a stopwords_list for enlgish combined with punctuation(default).

bs_ds.saywhat.**process_article**(*article, stopwords_list=['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't", '!', '"', '#', '$', '%', '&', "'", '(', ')', '*', '+', ',', '-', '.', '/', ':', ';', '<', '=', '>', '?', '@', '[', '\\', ']', '^', '_', '`', '{', '|', '}', '~', "''", "'''", '...', '‘‘’’']*)

Source: Learn.Co Text Classification Lab

bs_ds.saywhat.**search_twitter_api**(*api_object*, *searchQuery*, *maxTweets*, *fName*, *tweetsPerQry=100*, *max_id=0*, *sinceId=None*)

Take an authenticated tweepy api_object, a search queary, max# of tweets to retreive, a desintation filename. Uses tweept.api.search for the searchQuery until maxTweets is reached, saved harvest tweets to fName.

2.11 bs_ds.waldos_work module

class bs_ds.waldos_work.**MetaClassifier**(*classifiers=None*, *meta_classifier=None*, *use_probability=False*, *double_down=False*, *average_probs=False*, *clones=True*, *verbose=2*)

Bases: sklearn.base.BaseEstimator, sklearn.base.ClassifierMixin, sklearn.base.TransformerMixin

A model stacking classifier for sklearn classifiers. Uses Sklearn API to fit and predict, can be used with PipeLine and other sklearn estimators. Must be passed primary list of estimator(s) and secondary(meta) classifier. Secondary model trains a predicts on primary level estimators.

Parameters: ————————

classifiers [{array-like} shape = [n_estimators]] list of instantiated sklearn estimators.

meta_classifier [instatiated sklearn estimator.] This is the secondary estimator that makes the final prediction based on predicted values of classifiers.

use_probability [bool, (default=False) If True calling fit will train meta_classifier on the predicted probabilities] instead of predicted class labels.

double_down [bool, (default=False) If True, calling fit will train meta_classifier on both the primary] classifiers predicted lables and the original dataset. Otherwise meta_classifier will only

be trained on primary classifier's predicted labels.

average_probability [bool, (default = False) If True, calling fit will fit the meta_classifier with averaged] the probabalities from primiary predictions.

clones [bool, (default = True), If True, calling fit will fit deep copies of classifiers and meta classifier] leaving the original estimators unmodified. False will fit the passed in classifiers directly. This param is for use with non-sklearn estimators who cannot are not compatible with being cloned. This may be unecesary but I read enough things about it not working to set it as an option for safe measure. It is best to clone.

verbose : int, (0-2) Sets verbosity level for output while fitting.

clfs_ : list, fitted classifers (primary classifiers) **meta_clf_** : estimator, (secondary classifier) **meta_features_** : predictions from primary classifiers

Methods:

fit(X, y, sample_weight=None): fit entire ensemble with training data, including fitting meta_classifier with meta_dat

params: (See sklearns fit model for any estimator) X : {array-like}, shape = [n_samples, n_features] y : {array-like}, shape =[n_samples] sample_weight : array-like, shape = [n_samples], optional

fit_transform(X, y=None, fit_params) : Refer to Sklearn docs predict(X) : Predict labels get_params(params) : get classifier parameters, refer to sklearn class docs set_params(params) : set classifier parameters, mostly used internally, can be used to set parameters, refer to sklearn docs. score(X, y, sample_weight=None): Get accuracy score predict_meta(X): predict meta_features, primarily used to train meta_classifier, but can be used for base ensemeble performance predict_probs(X) : Predict label probabilities for X.

********** **EXAMPLE**********

EXAMPLE: # Instantiate classifier objects for base ensemble

>>>> xgb = XGBClassifier() >>>> svc = svm.SVC() >>>> gbc = GradientBoostingClassifier()

Store estimators in list

>>>> classifiers = [xgb, svc, gbc]

Instantiate meta_classifier for making final predictions

>>>> meta_classifier = LogisticRegression()

instantiate MetaClassifer object and pass classifiers and meta_classifier # Fit model with training data

>>>> clf = Metaclassifier(classifiers=classifiers, meta_classifier=meta_classifier) >>>> clf.fit(X_train, y_train)

Check accuracy scores, predict away...

>>>> print(f"MetaClassifier Accuracy Score: {clf.score(X_test, y_test)}") >>>> clf.predict(X)

---

fitting 3 classifiers... fitting 1/3 classifers... ... fitting meta_classifier...

time elapsed: 6.66 minutes MetaClassifier Accuracy Score: 99.9 Get it!

**********>

class BaseEstimator
Bases: object

Base class for all estimators in scikit-learn

Notes

All estimators should specify all the parameters that can be set at the class level in their __init__ as explicit keyword arguments (no *args or **kwargs).

get_params(*deep=True*)
Get parameters for this estimator.
Parameters **deep** (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.
Returns **params** – Parameter names mapped to their values.
Return type mapping of string to any

set_params(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.
Parameters ****params** (*dict*) – Estimator parameters.
Returns **self** – Estimator instance.
Return type object

class ClassifierMixin
Bases: object

Mixin class for all classifiers in scikit-learn.

score(*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.
Parameters
- **X** (*array-like of shape (n_samples, n_features)*) – Test samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs)*) – True labels for X.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns **score** – Mean accuracy of self.predict(X) wrt. y.
Return type float

class TransformerMixin
Bases: object

Mixin class for all transformers in scikit-learn.

fit_transform(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.
Parameters
- **X** (*numpy array of shape [n_samples, n_features]*) – Training set.
- **y** (*numpy array of shape [n_samples]*) – Target values.

- ****fit_params** (*dict*) – Additional fit parameters.

Returns **X_new** – Transformed array.

Return type numpy array of shape [n_samples, n_features_new]

fit (*X*, *y*, *sample_weight=None*)

Fit base classifiers with data and meta-classifier with predicted data from base classifiers.

Parameters: .————————.-. X : {array-like}, shape =[n_samples, n_features]

Training data m number of samples and number of features

y [{array-like}, shape = [n_samples] or [n_samples, n_outputs]] Target feature values.

Returns: .————————.-.

self [object,] Fitted MetaClassifier

predict (*X*)

Predicts target values.

X : np.array, shape=[n_samples, n_features]

predicted labels : array-like, shape = [n_samples] or [n_samples, n_outputs]

predict_meta (*X*)

Predicts on base estimators to get meta_features for MetaClassifier.

X : np.array, shape=[n_samples, n_features]

meta_features [np.array, shape=[n_samples, n_classifiers]] the 'new X' for the MetaClassifier to predict with.

predict_probs (*X*)

Predict probabilities for X

X : np.array, shape=[n_samples, n_features]

probabilities : array-like, shape = [n_samples, n_classes]

bs_ds.waldos_work.**compare_pipes** (*X_train*, *y_train*, *X_test*, *y_test*, *config_dict=None*, *n_components='mle'*, *search='random'*, *scaler=None*, *n_iter=5*, *random_state=42*, *cv=3*, *verbose=2*, *n_jobs=-1*, *save_pickle=False*)

Runs any number of estimators through pipeline and gridsearch(exhaustive or radomized) with cross validations, can print dataframe with scores, returns dictionary of all results.

estimator: estimator object, This is assumed to implement the scikit-learn estimator interface. Ex. sklearn.svm.SVC

params: dict, or list of dictionaries if using GridSearchcv, cannot pass lists if search='random

Dictionary with parameters names (string) as keys and distributions or lists of parameters to try. Distributions must provide a rvs method for sampling (such as those from scipy.stats.distributions). If a list is given, it is sampled uniformly.

MUST BE IN FORM: 'clf__param_'. ex. 'clf__C':[1, 10, 100]

X_train, y_train, X_test, y_test: training and testing data to fit, test to model

n_components: int, float, None or str. default='mle' Number of components to keep. if n_components is not set all components are kept. If n_components == 'mle' Minka's MLE is used to guess the dimension. For PCA.

search: str, 'random' or 'grid', Type of gridsearch to execute, 'random' = RandomizedSearchCV, 'grid' = GridSearchCV.

scaler: sklearn.preprocessing class instance, MUST BE IN FORM: StandardScaler(), (default=StandardScaler())

n_iter: int, Number of parameter settings that are sampled. n_iter trades off runtime vs quality of the solution.

random_state: int, RandomState instance or None, optional, default=42 Pseudo random number generator state used for random uniform sampling from lists of possible values instead of scipy.stats distributions. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

cv: int, cross-validation generator or an iterable, optional

Determines the cross-validation splitting strategy. Possible inputs for cv are: None, to use the default 3-fold cross validation, integer, to specify the number of folds in a (Stratified)KFold, CV splitter, An iterable yielding (train, test) splits as arrays of indices.

verbose [int,] Controls the verbosity: the higher, the more messages.

n_jobs [int or None, optional (default = -1)] Number of jobs to run in parallel. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. See Glossary for more details.

bs_ds.waldos_work.**describe_outliers**(*df*)
Returns a new_df of outliers, and % outliers each col using detect_outliers.

bs_ds.waldos_work.**draw_violinplot**(*x*, *y*, *hue=None*, *data=None*, *title=None*, *ticklabels=None*, *leg_label=None*)
Plots a violin plot with horizontal mean line, inner stick lines y must be arraylike in order to plot mean line. x can be label in data

bs_ds.waldos_work.**find_outliers**(*column*)

bs_ds.waldos_work.**fit_pipes**(*pipes_dict*, *train_test*, *predict=True*, *verbose=True*, *score='accuracy'*)
Fits piplines to training data, if predict=True, it displays a dataframe of scores. score can be either 'accuracy' or 'roc_auc'. rco_auc_score should be used with binary classification.

bs_ds.waldos_work.**make_config_dict**(*verbose=True*)
Generates a default dictioanry of models to test and hyperparameters Returns dictionary of configuration to use in compare_pipes. :param verbose: Defult=True, Displays contents of generated configs.

Ex: config_dict = make_config_dict()

bs_ds.waldos_work.**make_estimators_dict**()
Instantiates models as first step for creating pipelines.

bs_ds.waldos_work.**make_pipes**(*estimators_dict*, *scaler=None*, *n_components='mle'*, *random_state=42*)
Makes pipelines for given models, outputs dictionaries with keys as names and pipeline objects as values.

estimators: dict, dictionary with name (str) as key and estimator objects as values.

scaler: sklearn.preprocessing instance. Defaults to StandardScaler

bs_ds.waldos_work.**make_random_config_dict**(*verbose=True*)
Generates a default dictioanry of models to test and hyperparameters for a random grid search. Returns dictionary of configuration to use in random_pipes or compare_pipes. :param verbose: Defult=True, Displays contents of generated configs.

Ex: random_config_dict = make_random_config_dict()

bs_ds.waldos_work.**pipe_search**(*estimator*, *params*, *X_train*, *y_train*, *X_test*, *y_test*, *n_components='mle'*, *scaler=None*, *random_state=42*, *cv=3*, *verbose=2*, *n_jobs=-1*)

Fits pipeline and performs a grid search with cross validation using with given estimator and parameters.

estimator: estimator object, This is assumed to implement the scikit-learn estimator interface. Ex. sklearn.svm.SVC

params: dict, list of dicts,

Dictionary with parameters names (string) as keys and lists of parameter

settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored.This enables searching over any sequence of parameter settings. MUST BE IN FORM: 'clf__param_'. ex. 'clf__C':[1, 10, 100]

X_train, y_train, X_test, y_test: training and testing data to fit, test to model

n_components: int, float, None or str. default='mle' Number of components to keep. if n_components is not set all components are kept. If n_components == 'mle' Minka's MLE is used to guess the dimension. For PCA.

random_state: int, RandomState instance or None, optional, default=42 Pseudo random number generator state used for random uniform sampling from lists of possible values instead of scipy.stats distributions. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

cv: int, cross-validation generator or an iterable, optional

Determines the cross-validation splitting strategy. Possible inputs for cv are: None, to use the default 3-fold cross validation, integer, to specify the number of folds in a (Stratified)KFold, CV splitter, An iterable yielding (train, test) splits as arrays of indices.

verbose [int,] Controls the verbosity: the higher, the more messages.

n_jobs [int or None, optional (default = -1)] Number of jobs to run in parallel. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. See Glossary for more details.

dictionary: keys are: 'test_score' , 'best_accuracy' (training validation score), 'best_params', 'best_estimator', 'results'

bs_ds.waldos_work.**random_pipe**(*estimator*, *params*, *X_train*, *y_train*, *X_test*, *y_test*, *n_components='mle'*, *scaler=None*, *n_iter=10*, *random_state=42*, *cv=3*, *verbose=2*, *n_jobs=-1*)

Fits pipeline and performs a randomized grid search with cross validation.

estimator: estimator object, This is assumed to implement the scikit-learn estimator interface. Ex. sklearn.svm.SVC

params: dict,

Dictionary with parameters names (string) as keys and distributions or lists of parameters to try. Distributions must provide a rvs method for sampling (such as those from scipy.stats.distributions). If a list is given, it is sampled uniformly.

MUST BE IN FORM: 'clf__param_'. ex. 'clf__C':[1, 10, 100]

n_components: int, float, None or str. default='mle' Number of components to keep. if n_components is not set all components are kept. If n_components == 'mle' Minka's MLE is used to guess the dimension. For PCA.

X_train, y_train, X_test, y_test: training and testing data to fit, test to model

scaler: sklearn.preprocessing class instance, MUST BE IN FORM: StandardScaler(), (default=StandardScaler())

n_iter: int, Number of parameter settings that are sampled. n_iter trades off runtime vs quality of the solution.

random_state: int, RandomState instance or None, optional, default=42 Pseudo random number generator state used for random uniform sampling from lists of possible values instead of scipy.stats distributions. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by np.random.

cv: int, cross-validation generator or an iterable, optional

Determines the cross-validation splitting strategy. Possible inputs for cv are: None, to use the default 3-fold cross validation, integer, to specify the number of folds in a (Stratified)KFold, CV splitter, An iterable yielding (train, test) splits as arrays of indices.

verbose [int,] Controls the verbosity: the higher, the more messages.

n_jobs [int or None, optional (default = -1)]

Number of jobs to run in parallel. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. See Glossary for more details.

Returns:

dictionary: keys are: 'test_score' , 'best_accuracy' (training validation score), 'best_params', 'best_estimator', 'results'

bs_ds.waldos_work.**scale_data**(*data*, *scaler='standard'*, *log=False*)

Takes df or Series, scales it using desired method and returns scaled df.

Parameters

- **data** (*pd.Series or pd.DataFrame*) – entire dataframe of series to be scaled
- **method** (*str*) – The method for scaling to be implemented(default is 'minmax'). Other options are 'standard' or 'robust'.
- **log** (*bool, optional*) – Takes log of data if set to True(deafault is False).

Returns

Return type pd.DataFrame of scaled data.

bs_ds.waldos_work.**select_pca**(*features*, *n_components_list=None*)

Takes features and list of n_components to run PCA on. Default value of n_components_lists= None tests 2 to n_features-1.

features: pd.Dataframe n_components_list: List of n_components (ints) to test in PCA. Default = 2:n_features-1;

pd.DataFrame, displays number of components and their respective explained variance ratio

bs_ds.waldos_work.**thick_pipe**(*features, target, n_components='mle', classifiers=[None], test_size=0.25, random_state=42, verbose=False*)

Takes features and target, train/test splits and runs each through pipeline, outputs accuracy results models and train/test set in dictionary.

features: pd.Dataframe, variable features target: pd.Series, classes/labels n_components: int, number of prinicible components, use select_pca() to determine this number classifiers: list, classification models put in pipeline test_size: float, size of test set for test_train_split (default=.25) split_rand: int, random_state parameter for test_train_split (default=None) class_rand: int, random_state parameter for classifiers (default=None) verbose: bool, will print pipline instances as they are created (default=False)

dictionary: keys are abbreviated name of model ('LogReg', 'DecTree', 'RandFor', 'SVC'), 'X_train', 'X_test', 'y_train', 'y_test'. Values are dictionaries with keys for models: 'accuracy', 'model'. values are: accuracy score,and the classification model.

values for train/test splits.

bs_ds.waldos_work.**train_test_dict**(*X*, *y*, *test_size=0.25*, *random_state=42*)

Splits data into train/test sets and returns diction with each variable its own key and value.

CHAPTER 3

Installation

3.1 Stable release

To install bs_ds, run this command in your terminal:

```
$ pip install bs_ds
```

This is the preferred method to install bs_ds, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

3.2 From sources

The sources for bs_ds can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/jirvingphd/bs_ds
```

Or download the tarball:

```
$ curl  -OL https://github.com/jirvingphd/bs_ds/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/jirvingphd/bs_ds/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

bs_ds could always use more documentation, whether as part of the official bs_ds docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/jirvingphd/bs_ds/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *bs_ds* for local development.

1. Fork the *bs_ds* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/bs_ds.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv bs_ds
$ cd bs_ds/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 bs_ds tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/jirvingphd/bs_ds/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_bs_ds
```

4.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 5

Credits

5.1 Development Lead

- James M. Irving <james.irving.phd@outlook.com>
- Michael V. Moravetz <moravetzmichael@gmail.com>

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 0.2.0 (2019-05-03)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W