15-488 Spring '20 - Homework 2

Data cleaning and Feature selection: Practicing with the Python ecosystem for ML and Data Science

OUT: Feb 21, 2020 - DUE: Feb 27, 2020, 11:55pm

Instructions

The total number of points available from the questions is 125, where 25 points are *bonus* points (i.e., you only need 100 points to get the maximum grade).

By the due date, you have to submit on Autolab:

- The four functions as described in the writeup, putting them in a `hw2.zip` file

You can make a maximum of 10 submission attempts. At each submission you'll get a feedback about the performance of your function on our selected datasets.

1 Data cleaning: finding and imputing missing values

Problem 1.1: (38 points)

Write the function `handle_missing_values(df, drop_threshold)` that takes as input a pandas `DataFrame`, `df`, and a float in $[0, 1]$, `drop_threshold`.

`df` contains feature data (multiple features, multiple data entries). Data can be of numeric or categorical type (the data values in each column/feature have the same type).

Values can be missing and/or encoded as NaN values.

The numeric parameter `drop_threshold` represents the fraction of missing values that triggers data dropping. The use of `drop_threshold` is described below. In general, it controls when to drop a row, a column, or maybe the entire data frame, based on the ratio between the number of missing values and the total number of values.

The goal of the function is to check for the presence of missing values / not valid values and take action accordingly. At this aim, the methods from Pandas should be used.

The function must perform the operations described below, in the same precise same order.

1. **Check if there are too many missing entries.** Compute m, the total number of missing entries. If $m > n \cdot drop\_threshold$ the function returns an empty data frame, where n is the total number of values in the data frame, and $drop\_threshold$ is the input argument representing the fraction of missing values that triggers data dropping. In other words, if $m > n \cdot drop\_threshold$ it means that more than $drop\_threshold$ percent of the data entries are missing, such that it is better to drop the entire data frame.

   E.g., if m is 10, $n = 100$, and $drop\_threshold = 0.15$, the number of missing values, m, is less than 15% of the data, such that we decide to keep using the data frame. Instead, if $m = 20$, then we decide to drop the entire data frame (i.e., on returning an empty data frame, the function tells that the provided data are not clean enough for being used).

2. **Drop rows with too many missing entries.** While the previous check was global, this second step focuses on the presence of missing entries row by row. Similarly to the previous case, a row is removed from the data frame if the number of missing values in the row, mr, is greater than $r \cdot drop\_threshold$, where r is the number of data values in the row (which is equal to the number of columns in the data frame).

   E.g., if mr is 8, $r = 30$, and $drop\_threshold = 0.15$, the number of missing values in the row, mr, is greater than the 15% of the row data, 4.5 in this case, such that we decide to drop the entire row.

   Note that the `df.dropna()` method with the `threshold` parameter, allows to keep the row / columns with *at least* `threshold` entries. Acting on the rows or on the columns is regulated by the value of the `axis` parameter (0 rows, 1 columns).

   At the end of this step, the data frame is updated by removing the selected rows.

3. **Drop columns with too many missing entries.** This step is analogous to the previous one, except for the fact that the check/drop actions are executed on the columns.

   A column is removed from the data frame if the number of missing values in the column, mc, is greater than $c \cdot drop\_threshold$, where c is the number of data values in the column (which is equal to the number of rows in the data frame).

   E.g., if mc is 14, $c = 250$, and $drop\_threshold = 0.15$, the number of missing values in the column, mc, is less than the 15% of the column data, 37.5 in this case, such that we decide to keep the entire column.

At the end of this step, the data frame is updated by removing the selected columns.

4. **Impute numeric columns by statistical substitutions.** Now it's time to impute the missing values remained in the data frame after the above cleaning phase. First, imputation is performed on the columns with numeric values according to the following strategy.

   - If the average z-score over the column is less than or equal to 2, all the missing values in the column are substituted with the column mean. In other words, if the data in the column is more or less distributed as Normal data, it is reasonable to use the mean as surrogate data.
   - If the average z-score over the column is greater than 2, all the missing values in the column are substituted with the column median. In other words, if the data in the column is far from being distributed as Normal data, it is safe to use the median as surrogate data.

   Note that the average z-score is computed as the sum of the z-scores of all the c column values divided by the number of values (i.e., by c).

   At the end of this step, the data frame is updated by imputing all missing values in the numeric columns.

5. **Impute categorical columns by the most frequent category.** The next step consists in imputing the values of the columns with categorical data. Here a straightforward strategy consists in substituting all the missing values in a column with the most frequent categorical value of the column.

   At the end of this step, the data frame is updated by imputing all missing values in the categorical columns.

   Note that the `df.mode()` method from pandas works also with categorical data.

6. **Binarize categorical columns.** As a next step, all the categorical columns are made numeric by create new binary features for each category in each column.

   At the end of this step, the data frame is updated by transforming all categorical columns in multiple binary columns.

At the end of all the above steps, the new updated frame is returned by the function.

Since the binarization of the categorical columns creates new feature columns, the returned data frame must consist of first the numeric columns, followed by the categorical columns.

E.g., if the following data frame is given as input (note the convenient way to label columns as `xi`):

```
df = pd.DataFrame( [[1, 2, np.nan, 'A', 10],
                    [0.5, 3, False, 'B', 11],
                    [8, 1.5, True, 'B', 12],
                    [5, np.nan, np.nan, 'C', 20],
                    [np.nan, 2.5, False, np.nan, np.nan],
                    [np.nan, np.nan, True, np.nan, np.nan]],
                    columns = ['x{}'.format(i) for i in range(1, 6)])
```

The execution of the step-by-step operations implemented by the function result in the following data frame (using $drop\_threshold = 0.35$):

```
    x1   x2    x5     x3  x4_A  x4_B
0  1.0  2.0  10.0  False     1     0
1  0.5  3.0  11.0  False     0     1
2  8.0  1.5  12.0   True     0     1
```

2 Detect outliers

Problem 2.1: (30 points)

After handling the missing data, the next step of the data cleaning process consists in detecting and dealing with outliers. Here we consider a univariate data frame (technically a `Series`) of only numeric data (i.e., we only have one numeric feature to consider).

Your task is to implement the function `detect_outliers(df)` that takes as input a univariate data frame, `df`, and returns the list of outliers (and their indexes) present in the data, if any.

In principle, once we have the outliers, we can remove them, and then re-run the previous function `handle_missing_values()` that handles the missing data.

The function `detect_outliers(df)` performs the following actions.

1. Outlier detection can be done using either parametric or non-parametric methods. We use parametric methods when the data can be described by a parametric probability distribution, such as a Normal distribution. Therefore, as a first step, let's check whether this is the case for the data we are dealing with, limiting the check to the normality or not of the data. In other words, let's first answer to the question: are the data distributed as a Normal or not?

   In the literature, there are many different Normality tests, each with different pros and cons. Here we use the *Shapiro-Wilk* test https://en.wikipedia.org/wiki/Shapiro%E2%80%93Wilk_test.

   The `scipy.stats` module provides the `shapiro()` method that implements the test. The method returns two floats, the test statistic, and the p-value for the null hypothesis test (the *probability*-value). The test assumes the data sample was drawn from a Normal distribution. Technically, this is called the null hypothesis, or H_0. The p-value is the probability of obtaining the observed test results assuming H_0 is correct. You can (and should) read more about p-values here https://en.wikipedia.org/wiki/P-value

   A threshold level is chosen, usually called α, and is used to interpret the observed p-value:

   - $p \leq \alpha$: reject H_0, data is not normally distributed;
   - $p > \alpha$: fail to reject H_0, data can be assumed normally distributed.

   This means that, in general, we are seeking results with a larger p-value to confirm that the data sample was likely drawn from a Normal distribution.

   Note that a result above the given *alpha* does not mean that the null hypothesis is true. It means that it is very likely true given available evidence. The p-value is not the probability of the data fitting a Normal distribution. Instead, it should be thought of as a value that helps us interpret the statistical test.

   In practice the above test is all you have to execute, using a value of $\alpha = 0.02$

   Once executed the normality test, the result tells us whether we should use a parametric approach or not for outlier detection.

2. If the parametric test was passed positively, then the detection of the outliers makes use of the z-score. The function identifies all the entries that have a value $|z| > 3$, as well as their positions in the data series. Both the list of outlier values, `outliers`, and the list of their positions `where`, are returned by the function in a tuple `(outliers, where)`.

3. If the parametric test wasn't passed positively, then the detection of the outliers makes use of non-parametric methods, in particular of the IQR indicator. The function identifies as outliers all the entries with a value outside of the $1.5IQR$ range. As before, both the list of outlier

values, `outliers`, and the list of their positions `where`, are returned by the function in a tuple `(outliers, where)`.

3 Multiple regression and ordinary least squares

Problem 3.1: (28 points)

Implement the function `multiple_linear_regression(df)`, that takes as input a numeric data frame, `df`, that represents a dataset for multiple regression with n features. The n feature data are in the first n columns of the data frame, while the last column contains target values.

We are looking for a linear model of the form:

$$\hat{y}(\boldsymbol{x}) = c_1 x_1 + c_2 x_2 + \ldots c_n x_n + c_0,$$

where we aim to use the data in `df` to estimate the coefficients $\boldsymbol{c}$.

The function must make use of:

- `LinearRegression()` from `sklearn.linear_model` for building the linear model;
- `cross_validate()` from `sklearn.model_selection`, for fitting and scoring the linear model, using `cv=3` and `return_train_score=True` in the passed parameters.

The function returns the following tuple of floats:

`(mean_test_score, std_test_score, mean_train_score, std_train_score)`,

where `mean_test_score` is the mean of the test scores from CV and `std_test_score` is its associated standard deviation. `mean_train_score` is the mean of the training scores from CV and `std_train_score` is its associated standard deviation. All these data are obtainable from the return values of `cross_validate()`.

4 Multiple regression and polynomial feature transformation

Problem 4.1: (29 points)

Implement the function `multiple_linear_regression_poly_features(df, order)`, that takes as input a numeric data frame, `df`, that represents a dataset for multiple regression with n features. The n feature data are in the first n columns of the data frame, while the last column contains target values. The numeric parameter `order` represents the order of the polynomial feature transformation that is required.

We are looking for a linear model where the original features have been transformed according to a polynomial transformation of order `order` (bias should be included).

The function must make use of:

- `PolynomialFeatures()` from `sklearn.preprocessing` for transforming the features;

- `LinearRegression()` from `sklearn.linear_model` for building the linear model;
- `cross_validate()` from `sklearn.model_selection`, for fitting and scoring the linear model, using `cv=3` and `return_train_score=True` in the passed parameters.

You may want to use also the `Pipeline()` method.

As before, the function returns the following tuple of floats:

`(mean_test_score, std_test_score, mean_train_score, std_train_score)`,

where `mean_test_score` is the mean of the test scores from CV and `std_test_score` is its associated standard deviation. `mean_train_score` is the mean of the training scores from CV and `std_train_score` is its associated standard deviation. All these data are obtainable from the return values of `cross_validate()`.

You might want to check the performance of the regressor with polynomial features vs. the case of OLS with different datasets.