

PCA_log_regression

October 1, 2019

1 Creation of synthetic data for Wisconsin Breast Cancer data set using Principal Component Analysis. Tested using a logistic regression model.

1.1 Aim

To test a statistic method (principal component analysis) for synthesising data that can be used to train a logistic regression machine learning model.

1.2 Data

Raw data is available at:

<https://www.kaggle.com/uciml/breast-cancer-wisconsin-data>

1.3 Basic methods description

- Create synthetic data by sampling from distributions based on Principal Component Analysis of original data
- Train logistic regression model on synthetic data and test against held-back raw data

1.4 Code & results

```
[1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA

# Turn warnings off for notebook publication
import warnings
warnings.filterwarnings("ignore")
```

1.4.1 Import Data

```
[2]: def load_data():
      """
      Load Wisconsin Breast Cancer Data Set

      Inputs
      -----
      None

      Returns
      -----
      X: NumPy array of X
      y: Numpy array of y
      col_names: column names for X
      """

      # Load data and drop 'id' column
      data = pd.read_csv('./wisconsin.csv')
      data.drop('id', axis=1, inplace=True)

      # Change 'diagnosis' column to 'malignant', and put in last column place
      malignant = pd.DataFrame()
      data['malignant'] = data['diagnosis'] == 'M'
      data.drop('diagnosis', axis=1, inplace=True)

      # Split data in X and y
      X = data.drop(['malignant'], axis=1)
      y = data['malignant']

      # Get col names and convert to NumPy arrays
      X_col_names = list(X)
      X = X.values
      y = y.values

      return data, X, y, X_col_names
```

1.4.2 Data processing

Split X and y into training and test sets

```
[3]: def split_into_train_test(X, y, test_proportion=0.25):
      """
      Randomly split X and y numpy arrays into training and test data sets

      Inputs
      -----
      X and y NumPy arrays
```

```

Returns
-----
X_test, X_train, y_test, y_train Numpy arrays
"""

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, shuffle=True, test_size=test_proportion)

return X_train, X_test, y_train, y_test

```

Standardise data

```

[4]: def standardise_data(X_train, X_test):
      """
      Standardise training and tets data sets according to mean and standard
      deviation of test set
      Inputs
      -----
      X_train, X_test NumPy arrays
      Returns
      -----
      X_train_std, X_test_std
      """

      mu = X_train.mean(axis=0)
      std = X_train.std(axis=0)

      X_train_std = (X_train - mu) / std
      X_test_std = (X_test - mu) /std

      return X_train_std, X_test_std

```

1.4.3 Calculate accuracy measures

```

[5]: def calculate_diagnostic_performance(actual, predicted):
      """ Calculate sensitivity and specificty.
      Inputs
      -----
      actual, predted numpy arrays (1 = +ve, 0 = -ve)
      Returns
      -----
      A dictionary of results:

```

- 1) accuracy: proportion of test results that are correct
- 2) sensitivity: proportion of true +ve identified
- 3) specificity: proportion of true -ve identified
- 4) positive likelihood: increased probability of true +ve if test +ve
- 5) negative likelihood: reduced probability of true +ve if test -ve
- 6) false positive rate: proportion of false +ves in true -ve patients
- 7) false negative rate: proportion of false -ves in true +ve patients
- 8) positive predictive value: chance of true +ve if test +ve
- 9) negative predictive value: chance of true -ve if test -ve
- 10) actual positive rate: proportion of actual values that are +ve
- 11) predicted positive rate: proportion of predicted vales that are +ve
- 12) recall: same as sensitivity
- 13) precision: the proportion of predicted +ve that are true +ve
- 14) $f1 = 2 * ((precision * recall) / (precision + recall))$

**false positive rate is the percentage of healthy individuals who incorrectly receive a positive test result*
** false neagtive rate is the percentage of diseased individuals who incorrectly receive a negative test result*

"""

Calculate results

```

actual_positives = actual == 1
actual_negatives = actual == 0
test_positives = predicted == 1
test_negatives = predicted == 0
test_correct = actual == predicted
accuracy = test_correct.mean()
true_positives = actual_positives & test_positives
false_positives = actual_negatives & test_positives
true_negatives = actual_negatives & test_negatives
sensitivity = true_positives.sum() / actual_positives.sum()
specificity = np.sum(true_negatives) / np.sum(actual_negatives)
positive_likelihood = sensitivity / (1 - specificity)
negative_likelihood = (1 - sensitivity) / specificity
false_postive_rate = 1 - specificity
false_negative_rate = 1 - sensitivity
positive_predictive_value = true_positives.sum() / test_positives.sum()
negative_predictive_value = true_negatives.sum() / test_negatives.sum()
actual_positive_rate = actual.mean()
predicted_positive_rate = predicted.mean()
recall = sensitivity
precision = \
    true_positives.sum() / (true_positives.sum() + false_positives.sum())
f1 = 2 * ((precision * recall) / (precision + recall))

```

```

# Add results to dictionary
results = dict()
results['accuracy'] = accuracy
results['sensitivity'] = sensitivity
results['specificity'] = specificity
results['positive_likelihood'] = positive_likelihood
results['negative_likelihood'] = negative_likelihood
results['false_postive_rate'] = false_postive_rate
results['false_postive_rate'] = false_postive_rate
results['false_negative_rate'] = false_negative_rate
results['positive_predictive_value'] = positive_predictive_value
results['negative_predicitive_value'] = negative_predicitive_value
results['actual_positive_rate'] = actual_positive_rate
results['predicted_positive_rate'] = predicted_positive_rate
results['recall'] = recall
results['precision'] = precision
results['f1'] = f1

return results

```

1.4.4 Logistic Regression Model

```

[6]: def fit_and_test_logistic_regression_model(X_train, X_test, y_train, y_test):
    """
    Fit and test logistic regression model.
    Return a dictionary of accuracy measures.
    Calls on `calculate_diagnostic_performance` to calculate results

    Inputs
    -----
    X_train, X_test NumPy arrays

    Returns
    -----
    A dictionary of accuracy results.
    """

    # Fit logistic regression model
    lr = LogisticRegression(C=0.1)
    lr.fit(X_train, y_train)

    # Predict tets set labels
    y_pred = lr.predict(X_test_std)

    # Get accuracy results

```

```

accuracy_results = calculate_diagnostic_performance(y_test, y_pred)

return accuracy_results

```

1.4.5 Synthetic Data Method - Principal Component Analysis

- Transform original data by principal components
- Take mean and standard deviation of transformed data
- Create new data by sampling from distributions
- Inverse transform generated data back to original dimension space

```

[7]: def get_principal_component_model(data, n_components=0):
    """
    Principal component analysis

    Inputs
    -----
    data: raw data (DataFrame)

    Returns
    -----
    A dictionary of:
        model: pca model object
        transformed_X: transformed_data
        explained_variance: explained_variance
    """

    # If n_components not passed to function, use number of features in data
    if n_components == 0:
        n_components = data.shape[1]

    pca = PCA(n_components)
    transformed_X = pca.fit_transform(data)

    #fit_transform reduces X to the new datasize if n components is specified
    explained_variance = pca.explained_variance_ratio_

    # Compile a dictionary to return results
    results = {'model': pca,
               'transformed_X': transformed_X,
               'explained_variance': explained_variance}

    return results

[8]: def make_synthetic_data_pc(X_original, y_original, number_of_samples=1000,
                                n_components=0):
    """

```

Synthetic data generation.

Calls on `get_principal_component_model` for PCA model

If number of components not defined then the function sets it to the number of features in X

Inputs

original_data: X, y numpy arrays

number_of_samples: number of synthetic samples to generate

n_components: number of principal components to use for data synthesis

Returns

X_synthetic: NumPy array

y_synthetic: NumPy array

"""

If number of PCA not passed, set to number fo features in X

if n_components == 0:

 n_components = X_original.shape[1]

Split the training data into positive and negative

mask = y_original == 1

X_train_pos = X_original[mask]

mask = y_original == 0

X_train_neg = X_original[mask]

Pass malignant and benign X data sets to Principal Component Analysis

pca_pos = get_principal_component_model(X_train_pos, n_components)

pca_neg = get_principal_component_model(X_train_neg, n_components)

Set up list to hold malignant and benign transformed data

transformed_X = []

Create synthetic data for malignant and benign PCA models

for pca_model **in** [pca_pos, pca_neg]:

Get PCA transformed data

 transformed = pca_model['transformed_X']

Get means and standard deviations, to use for sampling

 means = transformed.mean(axis=0)

 stds = transformed.std(axis=0)

Make synthetic PC data using sampling from normal distributions

 synthetic_pca_data = np.zeros((number_of_samples, n_components))

```

    for pc in range(n_components):
        synthetic_pca_data[:, pc] = \
            np.random.normal(means[pc], stds[pc], size=number_of_samples)
    transformed_X.append(synthetic_pca_data)

    # Reverse transform data to create synthetic data to be used
    X_synthetic_pos = pca_pos['model'].inverse_transform(transformed_X[0])
    X_synthetic_neg = pca_neg['model'].inverse_transform(transformed_X[1])
    y_synthetic_pos = np.ones((X_synthetic_pos.shape[0],1))
    y_synthetic_neg = np.zeros((X_synthetic_neg.shape[0],1))

    # Combine positive and negative and shuffle rows
    X_synthetic = np.concatenate((X_synthetic_pos, X_synthetic_neg), axis=0)
    y_synthetic = np.concatenate((y_synthetic_pos, y_synthetic_neg), axis=0)

    # Randomise order of X, y
    synthetic = np.concatenate((X_synthetic, y_synthetic), axis=1)
    shuffle_index = np.random.permutation(np.arange(X_synthetic.shape[0]))
    synthetic = synthetic[shuffle_index]
    X_synthetic = synthetic[:,0:-1]
    y_synthetic = synthetic[:, -1]

    return X_synthetic, y_synthetic

```

1.4.6 Main code

```

[9]: # Load data
original_data, X, y, X_col_names = load_data()

# Set up results DataFrame
results = pd.DataFrame()

```

Fitting classification model to raw data

```

[10]: # Set number of replicate runs
number_of_runs = 30

# Set up lists for results
accuracy_measure_names = []
accuracy_measure_data = []

for run in range(number_of_runs):

    # Print progress
    print (run + 1, end=' ')

    # Split training and test set
    X_train, X_test, y_train, y_test = split_into_train_test(X, y)

```