VAE_random_forest

October 2, 2019

1 Creation of synthetic data for Wisoncsin Breat Cancer data set using a Variational AutoEncoder. Tested using a random forest model.

1.1 Aim

To test a a Variational AutoEncoder (VAE) for synthesising data that can be used to train a random forest machine learning model.

1.2 Data

Raw data is avilable at:
https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

1.3 Basic methods description

- Create synthetic data by use of a Variational AutoEncoder

Kingma, D.P. and Welling, M. (2013) Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114,2013.

- Train random forest 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.ensemble import RandomForestClassifier
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]:
```python
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]:
```python
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 sensitivty 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
* alse 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_predicitive_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))
```

```python
    # 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 Random Forest Model

[6]:
```python
def fit_and_test_random_forest_model(X_train, X_test, y_train, y_test):
    """
    Fit and test Random Forest 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.
    """

    # Define and fit model
    model = RandomForestClassifier(n_estimators=100,
                                   random_state=0,
                                   n_jobs=-1)
    model.fit(X_train,y_train)

    # Predict tets set labels
    y_pred = model.predict(X_test)
```

```
    # Get accuracy results
    accuracy_results = calculate_diagnostic_performance(y_test, y_pred)

    return accuracy_results
```

1.4.5 Synthetic Data Method - Variational AutoEncoder

[7]:
```
def sampling(args):
    """
    Reparameterization trick by sampling from an isotropic unit Gaussian.
    Instead of sampling from Q(z|X), sample epsilon = N(0,I)
    z = z_mean + sqrt(var) * epsilon

    # Arguments
        args (tensor): mean and log of variance of Q(z|X)

    # Returns
        z (tensor): sampled latent vector
    """

    import tensorflow
    from tensorflow.keras import backend as K

    z_mean, z_log_var = args
    batch = K.shape(z_mean)[0]
    dim = K.int_shape(z_mean)[1]
    # by default, random_normal has mean = 0 and std = 1.0
    epsilon = K.random_normal(shape=(batch, dim))

    sample = z_mean + K.exp(0.5 * z_log_var) * epsilon

    return sample
```

[8]:
```
def make_synthetic_data_vae(X_original, y_original,
                            batch_size=256,
                            latent_dim=8,
                            epochs=10000,
                            learning_rate=2e-5,
                            dropout=0.25,
                            number_of_samples=1000):
    """
    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
```

```python
    ------
    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

    """
    import tensorflow
    from tensorflow.keras import layers
    from tensorflow.keras.models import Model
    from tensorflow.keras.optimizers import Adam
    from tensorflow.keras import backend as K
    from tensorflow.keras.losses import mean_squared_error

    # Standardise X
    mean = X_original.mean(axis=0)
    std = X_original.mean(axis=0)
    X_std = (X_original - mean) / std

    # network parameters
    input_shape = X_original.shape[1]
    intermediate_dim = X_original.shape[1]

    # Split the training data into positive and negative
    mask = y_original == 1
    X_train_pos = X_std[mask]
    mask = y_original == 0
    X_train_neg = X_std[mask]

    # Set up list for positive and negative synthetic data sets
    synthetic_X_sets = []

    # Run fir twice: once for positive label examples, the other for negative
    for training_set in [X_train_pos, X_train_neg]:

        # Clear Tensorflow
        K.clear_session()

        # VAE model = encoder + decoder
        # build encoder model
        inputs = layers.Input(shape=input_shape, name='encoder_input')

        encode_dense_1 = layers.Dense(
```

```
        intermediate_dim, activation='relu')(inputs)

    dropout_encoder_layer_1 = layers.Dropout(dropout)(encode_dense_1)

    encode_dense_2 = layers.Dense(
        intermediate_dim, activation='relu')(dropout_encoder_layer_1)

    z_mean = layers.Dense(latent_dim, name='z_mean')(encode_dense_2)

    z_log_var = layers.Dense(latent_dim, name='z_log_var')(encode_dense_2)

    # use reparameterization trick to push the sampling out as input
    # note that "output_shape" isn't necessary with the TensorFlow backend
    z = layers.Lambda(
        sampling, output_shape=(latent_dim,), name='z')([z_mean, z_log_var])

    # instantiate encoder model
    encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')

    # build decoder model
    latent_inputs = layers.Input(shape=(latent_dim,), name='z_sampling')

    decode_dense_1 = layers.Dense(
        intermediate_dim, activation='relu')(latent_inputs)

    dropout_decoder_layer_1 = layers.Dropout(dropout)(decode_dense_1)

    decode_dense_2 = layers.Dense(
        intermediate_dim, activation='relu')(dropout_decoder_layer_1)

    outputs = layers.Dense(input_shape)(decode_dense_2)

    # instantiate decoder model
    decoder = Model(latent_inputs, outputs, name='decoder')

    # instantiate VAE model
    outputs = decoder(encoder(inputs)[2])
    vae = Model(inputs, outputs, name='vae_mlp')

    # Train the autoencoder

    optimizer = Adam(lr=learning_rate)

    # VAE loss = mse_loss or xent_loss + kl_loss
    vae.compile(optimizer=optimizer, loss = mean_squared_error)

    # Train the autoencoder
```

```
        vae.fit(training_set, training_set,
                batch_size = batch_size,
                shuffle = True,
                epochs = epochs,
                verbose=0)

        # Produce synthetic data
        z_new = np.random.normal(size = (number_of_samples, latent_dim))
        reconst = decoder.predict(np.array(z_new))
        reconst = mean + (reconst * std)
        synthetic_X_sets.append(reconst)

        # Clear models
        K.clear_session()
        del encoder
        del decoder
        del vae

    # Combine data
    # Combine positive and negative and shuffle rows
    X_synthetic = np.concatenate(
            (synthetic_X_sets[0], synthetic_X_sets[1]), axis=0)

    y_synthetic_pos = np.ones((number_of_samples, 1))
    y_synthetic_neg = np.zeros((number_of_samples, 1))

    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