UDACITY

PROJECT

Build a Sign Language Recognizer

A part of the Artificial Intelligence Nanodegree Program

PROJECT REVIEW

CODE REVIEW 8

NOTES

my_model_selectors.py 8

```
import math
import statistics
import warnings

import numpy as np
from hmmlearn.hmm import GaussianHMM
from sklearn.model_selection import KFold
from asl_utils import combine_sequences

import logging

class ModelSelector(object):
    '''
    base class for model selection (strategy design pattern)
    '''

    def __init__(self,
                 all_word_sequences: dict,
                 all_word_Xlengths: dict,
                 this_word: str,
                 n_constant=3,
                 min_n_components=2,
                 max_n_components=10,
                 random_state=14, verbose=False):
        self.words = all_word_sequences
        self.hwords = all_word_Xlengths
        self.sequences = all_word_sequences[this_word]
        self.X, self.lengths = all_word_Xlengths[this_word]
        self.this_word = this_word
        self.n_constant = n_constant
        self.min_n_components = min_n_components
        self.max_n_components = max_n_components
        self.random_state = random_state
        self.verbose = verbose

    def select(self):
        raise NotImplementedError

    def base_model(self, num_states):
        # with warnings.catch_warnings():
        warnings.filterwarnings("ignore", category=DeprecationWarning)
        # warnings.filterwarnings("ignore", category=RuntimeWarning)
        try:
            hmm_model = GaussianHMM(n_components=num_states, covariance_type="diag", n_iter=1000,
                                    random_state=self.random_state, verbose=False).fit(self.X, self.lengths)
            if self.verbose:
                print("model created for {} with {} states".format(self.this_word, num_states))
            return hmm_model
        except:
            if self.verbose:
                print("failure on {} with {} states".format(self.this_word, num_states))
            return None


class SelectorConstant(ModelSelector):
```

```
    """ select the model with value self.n_constant

    """

    def select(self):
        """ select based on n_constant value

        :return: GaussianHMM object
        """
        best_num_components = self.n_constant
        return self.base_model(best_num_components)


class SelectorBIC(ModelSelector):
    """
    Abbreviations:
        - BIC - Baysian Information Criterion
        - CV - Cross-Validation

    About BIC:
        - Maximises the likelihood of data whilst penalising large-size models
        - Used to scoring model topologies by balancing fit
          and complexity within the training set for each word
        - Avoids using CV by instead using a penalty term

    BIC Equation:  BIC = -2 * log L + p * log N
        (re-arrangment of Equation (12) in Reference [0])

        - where "L" is likelihood of "fitted" model
          where "p" is the qty of free parameters in model (aka model "complexity"). Reference [2][3]
          where "p * log N" is the "penalty term" (increases with higher "p"
             to penalise "complexity" and avoid "overfitting")
          where "N" is qty of data points (size of data set)

        Notes:
          -2 * log L    -> decreases with higher "p"
          p * log N     -> increases with higher "p"
          N > e^2 = 7.4 -> BIC applies larger "penalty term" in this case

    Selection using BIC Model:
        - Lower the BIC score the "better" the model.
        - SelectorBIC accepts argument of ModelSelector instance of base class
          with attributes such as: this_word, min_n_components, max_n_components,
        - Loop from min_n_components to max_n_components
        - Find the lowest BIC score as the "better" model.

    References:
        [0] - http://www2.imm.dtu.dk/courses/02433/doc/ch6_slides.pdf
        [1] - https://en.wikipedia.org/wiki/Hidden_Markov_model#Architecture
        [2] - https://stats.stackexchange.com/questions/12341/number-of-parameters-in-markov-model
        [3] - https://discussions.udacity.com/t/number-of-parameters-bic-calculation/233235/8
        [4] - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.6208&rep=rep1&type=pdf
    """
    def calc_score_bic(self, log_likelihood, num_states, num_data_points):
        return (-2 * log_likelihood) + (num_states * np.log(num_data_points))

```

REQUIRED

I agree with all but your value of `p`, it seems like you just used the number of states.

```
    def calc_best_score_bic(self, score_bics):
        # Min of list of lists comparing each item by value at index 0
        return min(score_bics, key = lambda x: x[0])

    def select(self):
        """ Select best model for self.this_word based on BIC score
        for n between self.min_n_components and self.max_n_components

        :return: GaussianHMM object
        """
        warnings.filterwarnings("ignore", category=DeprecationWarning)

        score_bics = []
        for num_states in range(self.min_n_components, self.max_n_components + 1):
            try:
                hmm_model = self.base_model(num_states)
                log_likelihood = hmm_model.score(self.X, self.lengths)
                num_data_points = sum(self.lengths)
                score_bic = self.calc_score_bic(log_likelihood, num_states, num_data_points)
                score_bics.append(tuple([score_bic, hmm_model]))
```

AWESOME

You calculated almost everything correctly!

```
            except:
                pass
        return self.calc_best_score_bic(score_bics)[1] if score_bics else None

class SelectorDIC(ModelSelector):
    """
    Abbreviations:
        - DIC - Discriminative Information Criterion

    About DIC:
        - In DIC we need to find the number of components where the difference is largest.
        The idea of DIC is that we are trying to find the model that gives a
        high likelihood (small negative number) to the original word and
        low likelihood (very big negative number) to the other words
        - In order to get an optimal model for any word, we need to run the model on all
        other words so that we can calculate the formula
        - DIC is a scoring model topology that scores the ability of a
        training set to discriminate one word against competing words.
        It provides a "penalty" if model likelihoods
        for non-matching words are too similar to model likelihoods for the
        correct word in the word set (rather than using a penalty term for
        complexity like in BIC)
        - Task-oriented model selection criterion adapts well to classification
        problems
        - Classification task accounts for Goal of model  (differs from BIC)

    DIC Equation:

        DIC = log(P(X(i)) - 1/(M - 1) * sum(log(P(X(all but i))

        (Equation (17) in Reference [0]. Assumes all data sets are same size)

            = log likelihood of the data belonging to model
              - avg of anti log likelihood of data X and model M

            = log(P(original word)) - average(log(P(other words)))

        where anti log likelihood means likelihood of data X and model M belonging to competing categories
        where log(P(X(i))) is the log-likelihood of the fitted model for the current word
        (in terms of hmmlearn it is the model's score for the current word)
        where where "L" is likelihood of data fitting the model ("fitted" model)
        where X is input training data given in the form of a word dictionary
        where X(i) is the current word being evaluated
        where M is a specific model

        Note:
            - log likelihood of the data belonging to model
            - anti_log_likelihood of data X vs model M

    Selection using DIC Model:
        - Higher the DIC score the "better" the model.
        - SelectorDIC accepts argument of ModelSelector instance of base class
          with attributes such as: this_word, min_n_components, max_n_components,
        - Loop from min_n_components to max_n_components
        - Find the highest BIC score as the "better" model.

    References:
        [0] - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.6208&rep=rep1&type=pdf
    """

    def calc_sum_anti_log_likelihoods(self, model, other_words):
        anti_log_likelihoods = []
        for word in other_words:
            anti_log_likelihoods.append(model[1].score(word[0], word[1]))
        return sum(anti_log_likelihoods)
```

AWESOME

Very good.

```

    def calc_avg_anti_log_likelihood(self, model, other_words):
        return self.calc_sum_anti_log_likelihoods(model, other_words) / (len(other_words) - 1)
```

SUGGESTION

Since other_words contain `m-1` words, you could just calculated the mean with `np.mean`

```

    def calc_best_score_dic(self, score_dics):
```

```
        # Max of list of lists comparing each item by value at index 0
        return max(score_dics, key = lambda x: x[0])

    def select(self):
        warnings.filterwarnings("ignore", category=DeprecationWarning)

        other_words = []
        models = []
        score_dics = []
        for word in self.words:
            if word != self.this_word:
```

AWESOME

👌 You skipped the word!

```
                other_words.append(self.hwords[word])
        try:
            for num_states in range(self.min_n_components, self.max_n_components + 1):
                hmm_model = self.base_model(num_states)
                log_likelihood = hmm_model.score(self.X, self.lengths)
                models.append((log_likelihood, hmm_model))

        # Note: Situation that may cause exception may be if have more parameters to fit
        # than there are samples, so must catch exception when the model is invalid
        except Exception as e:
            # logging.exception('DIC Exception occurred: ', e)
            pass
        for index, model in enumerate(models):
            log_likelihood, hmm_model = model
            score_dic = log_likelihood - self.calc_avg_anti_log_likelihood(model, other_words)
            score_dics.append(tuple([score_dic, model[1]]))
        return self.calc_best_score_dic(score_dics)[1] if score_dics else None


class SelectorCV(ModelSelector):
    """
    Abbreviations:
        - CV - Cross-Validation

    About CV:
        - Scoring the model simply using Log Likelihood calculated from
        feature sequences it trained on, we expect more complex models
        to have higher likelihoods, but doesn't inform us which would
        have a "better" likelihood score on unseen data. The model will
        likely overfit as complexity is added.
        - Estimate the "better" Topology model using only training data
        by comparing scores using Cross-Validation (CV).
        - CV technique includes breaking-down the training set into "folds",
        rotating which fold is "left out" of the training set.
        The fold that is "left out" is scored for validation.
        Use this as a proxy method of finding the
        "best" model to use on "unseen data".
        e.g. Given a set of word sequences broken-down into three folds
        using scikit-learn Kfold class object.
        - CV useful to limit over-validation

    CV Equation:

    Selection using CV Model:
        - Higher the CV score the "better" the model.
        - Select "best" model based on average log Likelihood
        of cross-validation folds
        - Loop from min_n_components to max_n_components
        - Find the higher score(logL), the higher the better.
        - Score that is "best" for SelectorCV is the
          average Log Likelihood of Cross-Validation (CV) folds.

    References:
        [0] - http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html
        [1] - https://www.r-bloggers.com/aic-bic-vs-crossvalidation/
    """

    def calc_best_score_cv(self, score_cv):
        # Max of list of lists comparing each item by value at index 0
        return max(score_cv, key = lambda x: x[0])

    def select(self):
        warnings.filterwarnings("ignore", category=DeprecationWarning)
        # logging.debug("Sequences: %r" % self.sequences)

        # num_splits = min(3, len(self.sequences))
        kf = KFold(n_splits = 3, shuffle = False, random_state = None)
        log_likelihoods = []
        score_cvs = []
```

```
        for num_states in range(self.min_n_components, self.max_n_components + 1):
            try:
                # Check sufficient data to split using KFold
                if len(self.sequences) > 2:
                    # CV loop of breaking-down the sequence (training set) into "folds" where a fold
                    # rotated out of the training set is tested by scoring for Cross-Validation (CV)
                    for train_index, test_index in kf.split(self.sequences):
                        # print("TRAIN indices:", train_index, "TEST indices:", test_index)

                        # Training sequences split using KFold are recombined
                        self.X, self.lengths = combine_sequences(train_index, self.sequences)

                        # Test sequences split using KFold are recombined
                        X_test, lengths_test = combine_sequences(test_index, self.sequences)
```

AWESOME

Very good making folds of training and testing!

```

                        hmm_model = self.base_model(num_states)
```

AWESOME

Nice, it will be fitted in the parent model!

```
                        log_likelihood = hmm_model.score(X_test, lengths_test)
```

AWESOME

Very good calculating the test score here.

```
                else:
                    hmm_model = self.base_model(num_states)
                    log_likelihood = hmm_model.score(self.X, self.lengths)

                log_likelihoods.append(log_likelihood)

                # Find average Log Likelihood of CV fold
                score_cvs_avg = np.mean(log_likelihoods)
                score_cvs.append(tuple([score_cvs_avg, hmm_model]))

            except Exception as e:
                pass
        return self.calc_best_score_cv(score_cvs)[1] if score_cvs else None

```

▸ my_recognizer.py

▸ asl_recognizer.html

RETURN TO PATH

Student FAQ