5A. K-Neighbors Regressor

February 18, 2019

1 K-Neighbors Regressor

Your goal is to train a model to predict the bitterness of a beer (in International Bittering Units, or IBU), given features about the beer. You can acquire the data in any one of three places:

- on Kaggle
- on Github
- in the `/data301/data/beer/` directory

A description of the variables is available here.

2 Question 1

Choose at least 5 different *sets* of features that you think might be important. For example, three different sets of features might be:

- `abv`
- `abv`, `available`, `originalGravity`
- `originalGravity`, `srm`

(You do not have to use these sets of features. They are provided just as an example.)

For each set of features, train a 30-nearest neighbor model to predict IBU (`ibu`). Determine which of these models is best at predicting IBU. Is it the model that contained the most features?

```
In [1]: %matplotlib inline
        import numpy as np
        import pandas as pd
        from sklearn.feature_extraction import DictVectorizer
        from sklearn.preprocessing import StandardScaler
        from sklearn.neighbors import KNeighborsRegressor
        from sklearn.pipeline import Pipeline
        from sklearn.model_selection import cross_val_score

        beer = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/master/da
        beer.head()
```

```
Out[1]:    id  abv                                          available  \
        0   0  8.2  Available at the same time of year, every year.
        1   1  5.7  Available at the same time of year, every year.
        2   2  5.8  Available at the same time of year, every year.
        3   3  5.5           Available year round as a staple beer.
        4   4  4.8           Available year round as a staple beer.

                                                 description glass   ibu isOrganic  \
        0  A Belgian-Abbey-Style Tripel that is big in al...   NaN  31.0         N
        1  Covert Hops is a crafty ale. Its stealthy dark...  Pint  45.0         N
        2  This is a traditional German-style Marzen char...   Mug  25.0         N
        3  A West Coast-Style Pale Ale balancing plenty o...  Pint  55.0         N
        4  This Bombshell has a tantalizing crisp and cle...  Pint  11.4         N

                             name  originalGravity srm
        0          LoonyToonTripel            1.070   8
        1              Covert Hops            1.056  35
        2              Oktoberfest            1.048  10
        3                 Pale Ale            1.044   5
        4  Head Turner Blonde Ale             1.045   3
```

```
In [2]: beer.glass = beer.glass.fillna("None")
        beer.head()
```

```
Out[2]:    id  abv                                          available  \
        0   0  8.2  Available at the same time of year, every year.
        1   1  5.7  Available at the same time of year, every year.
        2   2  5.8  Available at the same time of year, every year.
        3   3  5.5           Available year round as a staple beer.
        4   4  4.8           Available year round as a staple beer.

                                                 description glass   ibu isOrganic  \
        0  A Belgian-Abbey-Style Tripel that is big in al...  None  31.0         N
        1  Covert Hops is a crafty ale. Its stealthy dark...  Pint  45.0         N
        2  This is a traditional German-style Marzen char...   Mug  25.0         N
        3  A West Coast-Style Pale Ale balancing plenty o...  Pint  55.0         N
        4  This Bombshell has a tantalizing crisp and cle...  Pint  11.4         N

                             name  originalGravity srm
        0          LoonyToonTripel            1.070   8
        1              Covert Hops            1.056  35
        2              Oktoberfest            1.048  10
        3                 Pale Ale            1.044   5
        4  Head Turner Blonde Ale             1.045   3
```

```
In [3]: all_features = [["originalGravity"], ["abv", "srm"],
                         ["originalGravity", "srm"],
                         ["abv", "originalGravity"],
                        ["abv", "srm", "originalGravity"],
                        ["abv", "originalGravity", "isOrganic"],
                        ["abv", "available", "glass", "isOrganic", "originalGravity", "srm"]]

        for features in all_features:
            X_dict = beer[features].to_dict(orient="records")
            y = beer["ibu"]

            vec = DictVectorizer(sparse=False)
            scaler = StandardScaler()
            model = KNeighborsRegressor(n_neighbors=30)
            pipeline = Pipeline([("vectorizer", vec), ("scaler", scaler), ("fit", model)])

            total_folds= [2,3,4,5]
            for i in total_folds:
                scores = cross_val_score(pipeline, X_dict, y,
                                 cv=i, scoring="neg_mean_squared_error")

                print(features, i, np.sqrt(np.mean(-scores)))
```

```
['originalGravity'] 2 23.1539406672
['originalGravity'] 3 23.3379228836
['originalGravity'] 4 23.1234731653
['originalGravity'] 5 23.1313079554
['abv', 'srm'] 2 24.3933441211
['abv', 'srm'] 3 24.1708694674
['abv', 'srm'] 4 24.1028954899
['abv', 'srm'] 5 24.0659832385
['originalGravity', 'srm'] 2 24.3688221151
['originalGravity', 'srm'] 3 23.977658822
['originalGravity', 'srm'] 4 23.8003086177
['originalGravity', 'srm'] 5 23.7621349898
['abv', 'originalGravity'] 2 23.702238507
['abv', 'originalGravity'] 3 23.4563295741
['abv', 'originalGravity'] 4 23.4398991725
['abv', 'originalGravity'] 5 23.385530139
['abv', 'srm', 'originalGravity'] 2 23.9553324427
['abv', 'srm', 'originalGravity'] 3 23.7298163673
['abv', 'srm', 'originalGravity'] 4 23.5768929311
['abv', 'srm', 'originalGravity'] 5 23.5703079498
['abv', 'originalGravity', 'isOrganic'] 2 23.7800385539
['abv', 'originalGravity', 'isOrganic'] 3 23.521152455
['abv', 'originalGravity', 'isOrganic'] 4 23.5610546498
['abv', 'originalGravity', 'isOrganic'] 5 23.4628562603
['abv', 'available', 'glass', 'isOrganic', 'originalGravity', 'srm'] 2 25.3612600312
['abv', 'available', 'glass', 'isOrganic', 'originalGravity', 'srm'] 3 25.1724574696
['abv', 'available', 'glass', 'isOrganic', 'originalGravity', 'srm'] 4 25.0307462646
['abv', 'available', 'glass', 'isOrganic', 'originalGravity', 'srm'] 5 24.9822302529
```

The model that is best at predicting the IBU is the model using only 'originalGravity'. As we can see, this is not the model with the most features. The model with the most features was had the highest root mean square error in predicting the IBU's. Additionally, I tested whether using different values of k-folds makes an impact on the root mean square error. In almost every features group, having 5 folds yielded the lowest error, except in solely 'orginalGravity' and not by much. Despite the minimal advantage, I will continue onward with 4 folds.

I obtained this conclusion by first creating a list of lists of features that might be worth comparing to predict IBU. Then using the Pipeline methods to calculate the 30 nearest neighbors, I iterated through each list of features. Nested within this loop, I decided it would be best to compare different number of folds before calculating the cross-validation error. From the final results, I concluded it would be best to continue onward with 'abv' and 'originalGravity' at 5 folds.

3 Question 2

Let's see how the distance metric and the scaling method influence prediction accuracy. Use the set of features from Question 1 that you determined to be the best. Continue to use $k = 30$ nearest neighbors, but try fitting models with different distance metrics and scaling methods. Which distance metric and/or scaling method gives the best prediction accuracy?

```
In [4]: from sklearn.preprocessing import MinMaxScaler
        from sklearn.preprocessing import MaxAbsScaler
        from sklearn.preprocessing import StandardScaler
        from sklearn.preprocessing import RobustScaler
        from sklearn.preprocessing import Normalizer
        from sklearn.preprocessing import QuantileTransformer

        features = ["originalGravity"]

        X_dict = beer[features].to_dict(orient="records")
        y = beer["ibu"]

        vec = DictVectorizer(sparse=False)
        scalers = [MinMaxScaler(), MaxAbsScaler(),
                   StandardScaler(), RobustScaler(), Normalizer(),
                   QuantileTransformer()]

        for scaler in scalers:
            models = [KNeighborsRegressor(n_neighbors=30),
                      KNeighborsRegressor(metric="manhattan"),
                      KNeighborsRegressor(metric="euclidean")]

            for model in models:
                pipeline = Pipeline([("vectorizer", vec), ("scaler", scaler), ("fit", model)])
                scores = cross_val_score(pipeline, X_dict, y, cv=4,
                                         scoring="neg_mean_squared_error")
                print(scaler, model, np.sqrt(np.mean(-scores)))

MinMaxScaler(copy=True, feature_range=(0, 1)) KNeighborsRegressor(algorithm='auto', leaf_size=3
          metric_params=None, n_jobs=1, n_neighbors=30, p=2,
          weights='uniform') 23.1190462558
MinMaxScaler(copy=True, feature_range=(0, 1)) KNeighborsRegressor(algorithm='auto', leaf_size=3
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.6869077271
MinMaxScaler(copy=True, feature_range=(0, 1)) KNeighborsRegressor(algorithm='auto', leaf_size=3
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.6869077271
MaxAbsScaler(copy=True) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski'
          metric_params=None, n_jobs=1, n_neighbors=30, p=2,
          weights='uniform') 23.120870934
MaxAbsScaler(copy=True) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='manhattan'
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.6770895392
MaxAbsScaler(copy=True) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='euclidean'
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.6770895392
StandardScaler(copy=True, with_mean=True, with_std=True) KNeighborsRegressor(algorithm='auto',
          metric_params=None, n_jobs=1, n_neighbors=30, p=2,
          weights='uniform') 23.1234731653
StandardScaler(copy=True, with_mean=True, with_std=True) KNeighborsRegressor(algorithm='auto',
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.6828140237
StandardScaler(copy=True, with_mean=True, with_std=True) KNeighborsRegressor(algorithm='auto',
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.6828140237
RobustScaler(copy=True, quantile_range=(25.0, 75.0), with_centering=True,
       with_scaling=True) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski
          metric_params=None, n_jobs=1, n_neighbors=30, p=2,
          weights='uniform') 23.119650674
RobustScaler(copy=True, quantile_range=(25.0, 75.0), with_centering=True,
       with_scaling=True) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='manhatta
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.6818187092
RobustScaler(copy=True, quantile_range=(25.0, 75.0), with_centering=True,
       with_scaling=True) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='euclidea
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.6818187092
Normalizer(copy=True, norm='l2') KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='mi
          metric_params=None, n_jobs=1, n_neighbors=30, p=2,
          weights='uniform') 28.9843522335
Normalizer(copy=True, norm='l2') KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='ma
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 29.1423893582
```

```
Normalizer(copy=True, norm='l2') KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='eu
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 29.1423893582
QuantileTransformer(copy=True, ignore_implicit_zeros=False, n_quantiles=1000,
          output_distribution='uniform', random_state=None,
          subsample=100000) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkows
          metric_params=None, n_jobs=1, n_neighbors=30, p=2,
          weights='uniform') 23.1188674699
QuantileTransformer(copy=True, ignore_implicit_zeros=False, n_quantiles=1000,
          output_distribution='uniform', random_state=None,
          subsample=100000) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='manhat
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.7053249239
QuantileTransformer(copy=True, ignore_implicit_zeros=False, n_quantiles=1000,
          output_distribution='uniform', random_state=None,
          subsample=100000) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='euclid
          metric_params=None, n_jobs=1, n_neighbors=5, p=2,
          weights='uniform') 24.7053249239
```

I found that the Quantile Transformer scaling method with 30 nearest neighbors to be the best distance metric/scaling method because this combination yielded the lowest error. From the output, the Quantile Transformer always yielded the lowest errors, and then matched with 30 Nearest Neighbors, we obtain the minimal error. Therefore, I will continue onwards with the Quantile Transformer and k-nearest neighbors to minimize the test error.

I obtained this answer by first fixing the features to 'originalGravity' as dicussed from Question 1. Similarly to how I iterated through different lists of features, I created a list of various scalers and distance metrics. I first iterated through the scalers and then for each scaler, I calculated a different distance metric. Therefore, I was able to reach my conclusion that the Quantile Transformer and k-nearest neighbors is the best distance metric/scaling method.

4 Question 3

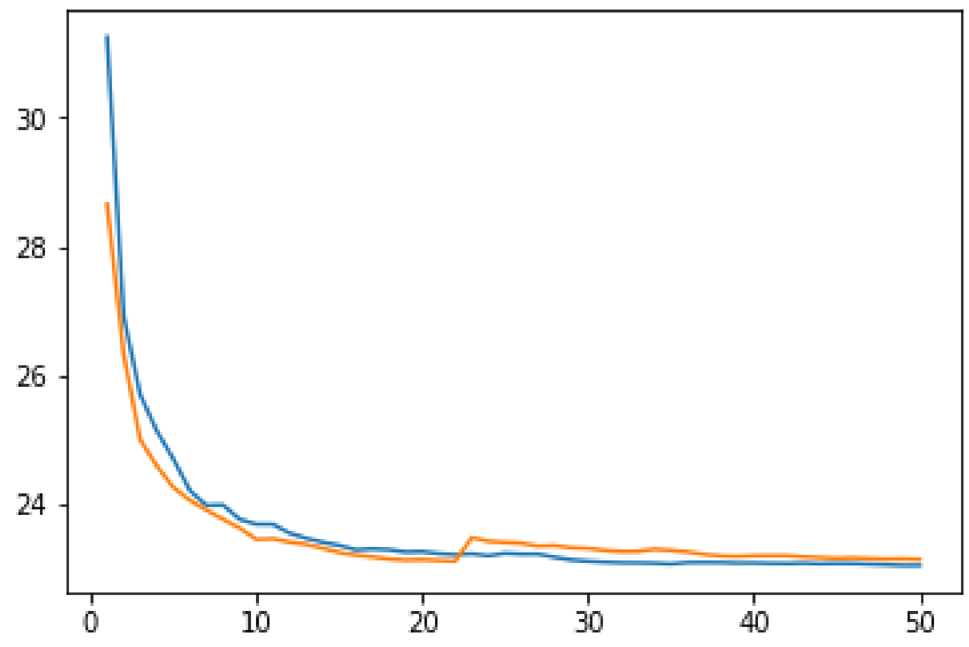
Now, we will determine the right value of k. Use the set of features, the distance metric, and the scaling method that you determined to be best in Questions 1 and 2. Fit k-nearest neighbor models for different values of k. Plot the training error and the test error as a function of k, and determine the optimal value of k.

```
In [5]: features = ["originalGravity"]

        X_dict = beer[features].to_dict(orient="records")
        y = beer["ibu"]

        def get_test_error(k):
            vec = DictVectorizer(sparse=False)
            scaler = QuantileTransformer()
            model = KNeighborsRegressor(n_neighbors=k)
            pipeline = Pipeline([("vectorizer", vec), ("scaler", scaler), ("fit", model)])
```

```
    scores = cross_val_score(pipeline, X_dict, y,
                 cv=4, scoring="neg_mean_squared_error")

    return np.sqrt(np.mean(-scores))

def get_train_error(k):
    vec = DictVectorizer(sparse=False)
    scaler = QuantileTransformer()
    model = KNeighborsRegressor(n_neighbors=k)
    pipeline = Pipeline([("vectorizer", vec), ("scaler", scaler), ("fit", model)])

    scores = cross_val_score(pipeline, X_dict, y,
                 scoring="neg_mean_squared_error")

    return np.sqrt(np.mean(-scores))

ks = pd.Series(range(1,51,1))
ks.index = range(1,51,1)
test_error = ks.apply(get_test_error)
train_error = ks.apply(get_train_error)
```

```
In [6]: test_error.plot.line()
        train_error.plot.line()
        test_error.sort_values().head()
```

```
Out[6]: 49    23.054290
        50    23.054899
        48    23.064056
        47    23.072211
        35    23.075306
        dtype: float64
```



The optimal value for k in k nearest-neighbors is 49. Using only 49 neighbors, yields the lowest test error of 23.054290. However, values for k such as 50, 48, 47, and 35 are all within .02 points.

I obtained this answer by fixing the features, scaling method, and distance metric to what we discussed before. I then created test error and training error functions that take "k" as input and return the rmse of each k. From there, it was simply sorting the values and calling .plot.line().

5 Submission Instructions

Once you are finished, follow these steps:

1. Restart the kernel and re-run this notebook from beginning to end by going to `Kernel > Restart Kernel and Run All Cells`.
2. If this process stops halfway through, that means there was an error. Correct the error and repeat Step 1 until the notebook runs from beginning to end.
3. Double check that there is a number next to each code cell and that these numbers are in order.

Then, submit your lab as follows:

1. Go to `File > Export Notebook As > PDF`.
2. Double check that the entire notebook, from beginning to end, is in this PDF file. (If the notebook is cut off, try first exporting the notebook to HTML and printing to PDF.)
3. Upload the PDF to PolyLearn.