In [103]:
```
%matplotlib inline
import matplotlib
import seaborn as sns
sns.set()
matplotlib.rcParams['figure.dpi'] = 144
```

In [104]:
```
from static_grader import grader
```

ML: Predicting Star Ratings

Our objective is to predict a new venue's popularity from information available when the venue opens. We will do this by machine learning from a data set of venue popularities provided by Yelp. The data set contains meta data about the venue (where it is located, the type of food served, etc.). It also contains a star rating. Note that the venues are not limited to restaurants. This tutorial will walk you through one way to build a machine-learning algorithm.

Metric

Your model will be assessed based on the root mean squared error of the number of stars you predict. There is a reference solution (which should not be too hard to beat). The reference solution has a score of 1. Keeping this in mind...

A note on scoring

It **is** possible to score >1 on these questions. This indicates that you've beaten our reference model - we compare our model's score on a test set to your score on a test set. See how high you can go!

Download and parse the incoming data

We start by downloading the data set from Amazon S3:

In [105]:
```
!aws s3 sync s3://dataincubator-course/mldata/ . --exclude '*' --include 'yelp
_train_academic_dataset_business.json.gz'
```

The training data are a series of JSON objects, in a Gzipped file. Python supports Gzipped files natively: `gzip.open` (https://docs.python.org/2/library/gzip.html) has the same interface as `open`, but handles `.gz` files automatically.

The built-in json package has a `loads()` function that converts a JSON string into a Python dictionary. We could call that once for each row of the file. `ujson` (http://docs.micropython.org/en/latest/library/ujson.html) has the same interface as the built-in `json` library, but is *substantially* faster (at the cost of non-robust handling of malformed json). We will use that inside a list comprehension to get a list of dictionaries:

In [106]:
```
import ujson as json
import gzip
import pandas as pd

with gzip.open('yelp_train_academic_dataset_business.json.gz') as f:
    data = [json.loads(line) for line in f]
```

In Scikit Learn, the labels to be predicted, in this case, the stars, are always kept in a separate data structure than the features. Let's get in this habit now, by creating a separate list of the ratings:

In [259]:
```
star_ratings = [entry['stars'] for entry in data]
```

Notes:

1. Pandas (http://pandas.pydata.org/) is able to read JSON text directly. Use the `read_json()` function with the `lines=True` keyword argument. While the rest of this notebook will assume you are using a list of dictionaries, you can complete it with dataframes, if you so desire. Some of the example code will need to be modified in this case.
2. There are obvious mistakes in the data. There is no need to try to correct them.

Building models

For many of the questions below, you will need to build and train an estimator that predicts the star rating given certain features. This could be a custom estimator that you built from scratch, but in most cases will be a pipeline containing custom or pre-built transformers and an existing estimator. We will give you hints of how to proceed, but the only requirement for you is to produce a model that does as well, or better, than the reference models we created. You are welcome to do this however you like. The details are up to you.

The formats of the input and output to the `fit()` and `predict()` methods are ultimately up to you as well, but we recommend that you deal with lists or arrays, for consistency with the rest of Scikit Learn. It is also a good idea to take the same type of data for the feature matrix in both `fit()` and `predict()`. While it is tempting to read the stars from the feature matrix X, you should get in the habit of passing the labels as a separate argument to the `fit()` method.

You may find it useful to serialize the trained models to disk. This will allow to reload it after restarting the Jupyter notebook, without needing to retrain it. We recommend using the `dill` library (https://pypi.python.org/pypi/dill) for this (although the `joblib` library (http://scikit-learn.org/stable/modules/model_persistence.html) also works). Use

```
dill.dump(estimator, open('estimator.dill', 'w'))
```

to serialize the object `estimator` to the file `estimator.dill`. If you have trouble with this, try setting the `recurse=True` keyword arguments in the call of `dill.dump()`. The estimator can be deserialized by calling

```
estimator = dill.load(open('estimator.dill', 'r'))
```

Questions

Each of the "model" questions asks you to create a function that models the number of stars venues will receive. It will be passed a list of dictionaries. Each of these will have the same format as the JSON objects you've just read in. Some of the keys (like the stars!) will have been removed. This function should return a list of numbers of the same length, indicating the predicted star ratings.

This function is passed to the `score()` function, which will receive input from the grader, run your function with that input, report the results back to the grader, and print out the score the grader returned. Depending on how you constructed your estimator, you may be able to pass the predict method directly to the `score()` function. If not, you will need to write a small wrapper function to mediate the data types.

city_avg

The venues belong to different cities. You can imagine that the ratings in some cities are probably higher than others. We wish to build an estimator to make a prediction based on this, but first we need to work out the average rating for each city. For this problem, create a list of tuples (city name, star rating), one for each city in the data set.

There are many ways to do this; please feel free to experiment on your own. If you get stuck, the steps below attempt to guide you through the process.

A simple approach is to go through all of the dictionaries in our array, calculating the sum of the star ratings and the number of venues for each city. At the end, we can just divide the stars by the count to get the average.

We could create a separate sum and count variable for each city, but that will get tedious quickly. A better approach to to create a dictionary for each. The key will be the city name, and the value the running sum or running count.

One slight annoyance of this approach is that we will have to test whether a key exists in the dictionary before adding to the running tally. The collections module's `defaultdict` class works around this by providing default values for keys that haven't been used. Thus, if we do

In [260]:
```
from collections import defaultdict
star_sum = defaultdict(int)
count = defaultdict(int)
```

we can increment any key of `stars` or `count` without first worrying whether the key exists. We need to go through the `data` and `star_ratings` list together, which we can do with the `zip()` function.

In [261]:
```
for row, stars in zip(data, star_ratings):
    # increment the running sum in star_sum
    star_sum[row['city']] += stars
    # increment the running count in count
    count[row['city']] += 1
```

Now we can calculate the average ratings. Again, a dictionary makes a good container.

In [262]:
```
avg_stars = dict()
for city in star_sum:
    avg_stars.update({city : star_sum[city]/count[city]})
```

There should be 167 different cities:

In [263]:
```
assert len(avg_stars) == 167
```

We can get that list of tuples by converting the returned view object from the `.items()` method into a list.

In [264]:
```
grader.score('ml__city_avg', list(avg_stars.items()))
```

```
==================
Your score:  1.0
==================
```

city_model

Now, let's build a custom estimator that will make a prediction based solely on the city of a venue. It is tempting to hard-code the answers from the previous section into this model, but we're going to resist and do things properly.

This custom estimator will have a `.fit()` method. It will receive `data` as its argument `X` and `star_ratings` as `y`, and should repeat the calculation of the previous problem there. Then the `.predict()` method can look up the average rating for the city of each record it receives.

In [265]:
```python
from sklearn import base

class CityEstimator(base.BaseEstimator, base.RegressorMixin):

    def __init__(self):
        self.avg_stars = dict()
        self.running_average = 0

    def fit(self, X, y):
        star_sum = defaultdict(int)
        count = defaultdict(int)

        for row, stars in zip(X, y):
            # increment the running sum in star_sum
            star_sum[row['city']] += stars
            self.running_average += stars
            # increment the running count in count
            count[row['city']] += 1

        self.running_average /= len(X)

        # Store the average rating per city in self.avg_stars
        for city in star_sum:
            self.avg_stars.update({city : star_sum[city]/count[city]})

    def predict(self, X):
        predictions = []

        for row in X:
            if row['city'] in self.avg_stars:
                predictions.append(self.avg_stars[row['city']])
            else:
                predictions.append(self.running_average)

        return predictions
#[self.avg_stars[row['city']] for row in X]
```

Now we can create an instance of our estimator and train it.

In [266]:
```python
city_est = CityEstimator()
city_est.fit(data, star_ratings)
```

And let's see if it works.

In [267]:
```python
city_est.predict(data[:5])
```

Out[267]:
```
[3.6702903946388683, 3.75, 3.75, 3.75, 3.75]
```

There is a problem, however. What happens if we're asked to estimate the rating of a venue in a city that's not in our training set?

```
In [268]: city_est.predict([{'city': 'Timbuktu'}])
```

```
Out[268]: [3.6729137013021247]
```

Solve this problem before submitting to the grader.

```
In [269]: grader.score('ml__city_model', city_est.predict)
```

```
==================
Your score:  1.0
==================
```

lat_long_model

You can imagine that a city-based model might not be sufficiently fine-grained. For example, we know that some neighborhoods are trendier than others. Use the latitude and longitude of a venue as features that help you understand neighborhood dynamics.

Instead of writing a custom estimator, we'll use one of the built-in estimators in Scikit Learn. Since these estimators won't know what to do with a list of dictionaries, we'll build a `ColumnSelectTransformer` that will return an array containing selected keys of our feature matrix. While it is tempting to hard-code the latitude and longitude in here, this transformer will be more useful in the future if we write it to work on an arbitrary list of columns.

```
In [287]: class ColumnSelectTransformer(base.BaseEstimator, base.TransformerMixin):

              def __init__(self, col_names):
                  self.col_names = col_names  # We will need these in transform()

              def fit(self, X, y=None):
                  # This transformer doesn't need to learn anything about the data,
                  # so it can just return self without any further processing
                  return self

              def transform(self, X):
                  # Return an array with the same number of rows as X and one
                  # column for each in self.col_names
                  new_list = []

                  for row in X:
                      new_list.append([row['latitude'], row['longitude']])

                  return new_list
```

Let's test it on a single row, just as a sanity check:

```
In [288]: cst = ColumnSelectTransformer(['latitude', 'longitude'])
          assert (cst.fit_transform(data[:1])
                  == [[data[0]['latitude'], data[0]['longitude']]])
```

Now, let's feed the output of the transformer in to a `sklearn.neighbors.KNeighborsRegressor`. As a sanity check, we'll test it with the first 5 rows. To truly judge the performance, we'd need to make a test/train split.

```
In [289]: from sklearn.neighbors import KNeighborsRegressor

          data_transform = cst.fit_transform(data)
          knn = KNeighborsRegressor(n_neighbors=5)
          knn.fit(data_transform, star_ratings)
          test_data = data[:5]
          test_data_transform = cst.transform(test_data)
          knn.predict(test_data_transform)

Out[289]: array([4. , 4.2, 4. , 3.8, 4.2])
```

Instead of doing this by hand, let's make a pipeline. Remember that a pipeline is made with a list of (name, transformer-or-estimator) tuples.

```
In [316]: from sklearn.pipeline import Pipeline

          pipe = Pipeline([
              ("ColumnSelectTransformer", cst),
              ("KNeighborsRegressor", KNeighborsRegressor(n_neighbors=2))
               ])

          new_data = train_test_split(data)

          ---------------------------------------------------------------------------
          NameError                                 Traceback (most recent call last)
          <ipython-input-316-2990d8d737ec> in <module>()
                6      ])
                7
          ----> 8 new_data = train_test_split(data)

          NameError: name 'train_test_split' is not defined
```

This should work the same way.

```
In [332]: from sklearn.model_selection import train_test_split

          pipe.fit(data, star_ratings)
          pipe.predict(test_data)


          train, test = train_test_split(data, test_size=0.01)

          expected = []
          for row in data[:5]:
              expected.append(row['stars'])
          print(expected)
```

```
37558
380
[3.5, 4.0, 4.0, 4.5, 4.0]
```

The `KNeighborsRegressor` takes the `n_neighbors` hyperparameter, which tells it how many nearest neighbors to average together when making a prediction. There is no reason to believe that 5 is the optimum value. Determine a better value of this hyperparameter. There are several ways to do this:

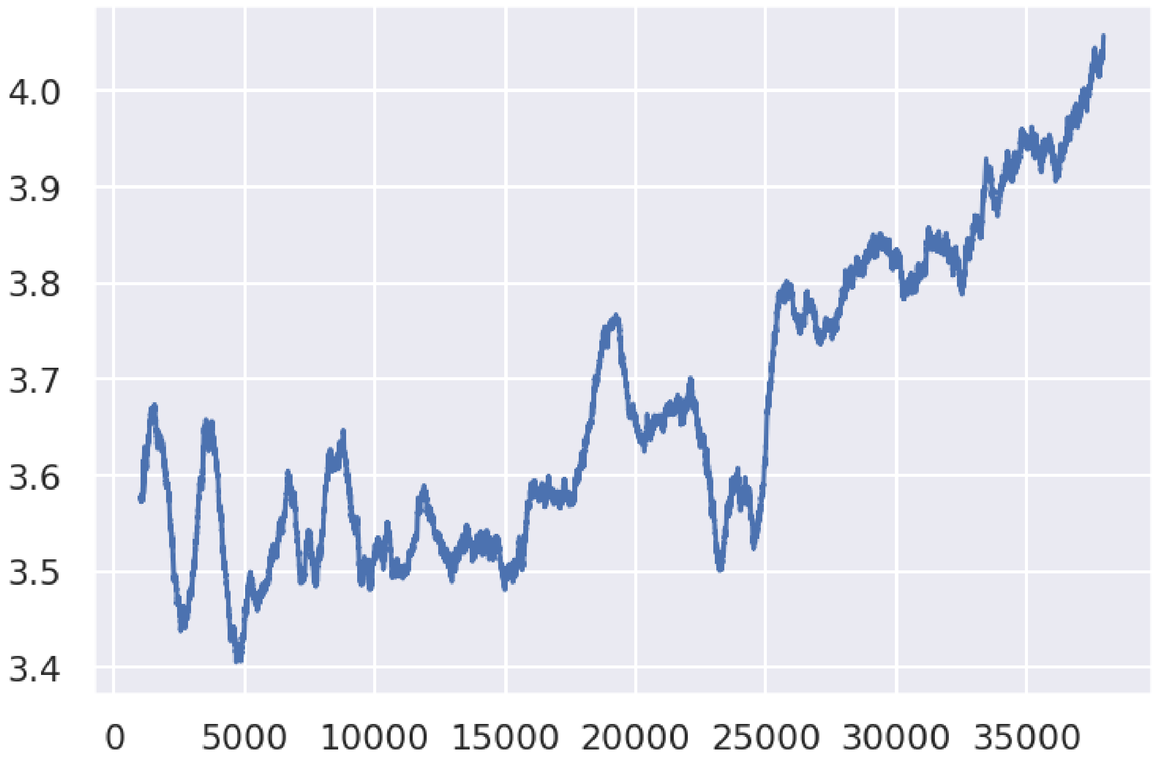
1. Use `train_test_split` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html#sklearn.model_selection.trai to split your data in to a training set and a test set. Score the performance on the test set. After finding the best hyperparameter, retrain the model on the full data at that hyperparameter value.
2. Use `cross_val_score` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html#sklearn.model_selection.c to return cross-validation scores on your data for various values of the hyperparameter. Choose the best one, and retrain the model on the full data.
3. Use `GridSearchCV` (http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.Gri to do the splitting, training, and grading automatically. `GridSearchCV` takes an estimator and acts as an estimator. You can either give it the `KNeighborsRegressor` directly and put it in a pipeline, or you can pass the whole pipeline into the `GridSearchCV`. In the latter case, remember that the hyperparameter `param` of an estimator named `est` in a pipeline becomes a hyperparameter of the pipeline with name `est__param`.

No matter which you choose, you should consider whether the data need to be shuffled. The default k-folds split doesn't shuffle. This is fine, if the data are already random. The code below will plot a rolling mean of the star ratings. Do you need to shuffle the data?

```
In [294]: from pandas import Series
          import matplotlib.pyplot as plt

          plt.plot(Series.rolling(Series(star_ratings), window=1000).mean())

Out[294]: [<matplotlib.lines.Line2D at 0x7fbebde179b0>]
```



Once you've found a good value of `n_neighbors`, submit the model to the grader. (*N.B.* "Good" is a relative measure here. The reference solution has a r-squared value of only 0.02. There is just rather little signal available for modeling.)

```
In [295]: grader.score('ml__lat_long_model', lat_long_est.predict)  # Edit to appropriat
          e name

          ---------------------------------------------------------------------------
          NameError                                 Traceback (most recent call last)
          <ipython-input-295-a25e8458f4e5> in <module>()
          ----> 1 grader.score('ml__lat_long_model', lat_long_est.predict)  # Edit to a
          ppropriate name

          NameError: name 'lat_long_est' is not defined
```

Item for thought: Why do we choose a non-linear model for this estimator?

Extension: Use a `sklearn.ensemble.RandomForestRegressor`, which is a more powerful non-linear model. Can you get better performance with this than with the `KNeighborsRegressor`?

category_model

While location is important, we could also try seeing how predictive the venue's category is. Build an estimator that considers only the categories.

The categories come as a list of strings, but the built-in estimators all need numeric input. The standard way to deal with categorical features is **one-hot encoding**, also known as dummy variables. In this approach, each category gets its own column in the feature matrix. If the row has a given category, that column gets filled with a 1. Otherwise, it is 0.

The `ColumnSelectTransformer` from the previous question can be used to extract the categories column as a list of strings. Scikit Learn provides `DictVectorizer` (http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.DictVectorizer.html#sklearn.feature_extraction.Dict\ which takes in a list of dictionaries. It creates a column in the output matrix for each key in the dictionary and fills it with the value associated with it. Missing keys are filled with zeros. Therefore, we need only build a transformer that takes a list of strings to a dictionary with keys given by those strings and values of one.

In [ ]:
```
class DictEncoder(base.BaseEstimator, base.TransformerMixin):

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        # X will come in as a list of lists of lists.  Return a list of
        # dictionaries corresponding to those inner lists.
```

That should allow this to pass:

In [ ]:
```
assert (DictEncoder().fit_transform([[['a']], [['b', 'c']]])
        == [{'a': 1}, {'b': 1, 'c': 1}])
```

Set up a pipeline with your `ColumnSelectTransformer`, your `DictEncoder`, the `DictVectorizer`, and a regularized linear model, like `Ridge`, as the estimator. This model will have a large number of features, one for each category, so there is a significant danger of overfitting. Use cross validation to choose the best regularization parameter.

In [ ]:
```
grader.score('ml__category_model', category_est.predict)  # Edit to appropriat
e name
```

Extension: Some categories (e.g. Restaurants) are not very specific. Others (Japanese sushi) are much more so. One way to deal with this is with an measure call term-frequency-inverse-document-frequency (TF-IDF). Add in a `sklearn.feature_extraction.text.TfidfTransformer` between the `DictVectorizer` and the linear model, and see if that improves performance.

Extension: Can you beat the performance of the linear estimator with a non-linear model?

attribute_model

There is even more information in the attributes for each venue. Let's build an estimator based on these.

Venues attributes may be nested:

```
{
  'Attire': 'casual',
  'Accepts Credit Cards': True,
  'Ambiance': {'casual': False, 'classy': False}
}
```

We wish to encode them with one-hot encoding. The `DictVectorizer` can do this, but only once we've flattened the dictionary to a single level, like so:

```
{
  'Attire_casual' : 1,
  'Accepts Credit Cards': 1,
  'Ambiance_casual': 0,
  'Ambiance_classy': 0
}
```

Build a custom transformer that flattens the attributes dictionary. Place this in a pipeline with a `DictVectorizer` and a regressor.

You may find it difficult to find a single regressor that does well enough. A common solution is to use a linear model to fit the linear part of some data, and use a non-linear model to fit the residual that the linear model can't fit. Build a residual estimator that takes as an argument two other estimators. It should use the first to fit the raw data and the second to fit the residuals of the first.

In [ ]:
```
grader.score('ml__attribute_model', attribute_est.predict)  # Edit to appropri
ate name
```

full_model

So far we have only built models based on individual features. Now we will build an ensemble regressor that averages together the estimates of the four previous regressors.

In order to use the existing models as input to an estimator, we will have to turn them into transformers. (A pipeline can contain at most a single estimator.) Build a custom `ModelTransformer` class that takes an estimator as an argument. When `fit()` is called, the estimator should be fit. When `transform()` is called, the estimator's `predict()` method should be called, and its results returned.

Note that the output of the `transform()` method should be a 2-D array with a single column, in order for it to work well with the Scikit Learn pipeline. If you're using NumPy arrays, you can use `.reshape(-1, 1)` to create a column vector. If you are just using Python lists, you will want a list of lists of single elements.

```
In [ ]: class EstimatorTransformer(base.BaseEstimator, base.TransformerMixin):

            def __init__(self, estimator):
                # What needs to be done here?

            def fit(self, X, y):
                # Fit the stored estimator.
                # Question: what should be returned?

            def transform(self, X):
                # Use predict on the stored estimator as a "transformation".
                # Be sure to return a 2-D array.
```

This should work as follows:

```
In [ ]: city_trans = EstimatorTransformer(city_est)
        city_trans.fit(data, star_ratings)
        assert ([r[0] for r in city_trans.transform(data[:5])]
                == city_est.predict(data[:5]))
```

Create an instance of `ModelTransformer` for each of the previous four models. Combine these together in a single feature matrix with a `FeatureUnion` (http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.FeatureUnion.html#sklearn.pipeline.FeatureUnion).

```
In [ ]: from sklearn.pipeline import FeatureUnion

        union = FeatureUnion([
                # FeatureUnions use the same syntax as Pipelines
            ])
```

This should return a feature matrix with four columns.

```
In [ ]: union.fit(data, star_ratings)
        trans_data = union.transform(data[:10])
        assert trans_data.shape == (10, 4)
```

Finally, use a pipeline to combine the feature union with a linear regression (or another model) to weight the predictions.

```
In [ ]: grader.score('ml__full_model', full_est.predict)  # Edit to appropriate name
```

Extension: By combining our models with a linear model, we will be unable to notice any correlation between features. We don't expect all attributes to have the same effect on all venues. For example, "Ambiance: divey" might be a bad indicator for a restaurant but a good one for a bar. Nonlinear models can pick up on this interaction. Replace the linear model combining the predictions with a nonlinear one like `RandomForestRegressor` (http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html#sklearn.ensemble.RandomF
Better yet, use the nonlinear model to fit the residuals of the linear model.

The score for this question is just a ratio of the score of your model to the score of a reference solution. Can you beat the reference solution and get a score greater than 1.0?