ISTA 331 HW: K-MEANS CLUSTERING

Due Thursday, 4/2, 2020, 11:59 PM

Introduction. In this homework, we will use `DataFrame`s to represent collections of feature vectors. Each row will be a feature vector. Our row labels will be either `datetime` objects or traditional integer indices, and our column labels will be feature names. We will be examining weather data from Tucson International Airport, TIA, for the thirty-year period 1987-2016.

We will investigate this data using *clustering*, which is a form of *unsupervised learning*. All the models we have looked at before are *supervised*, meaning that we use labeled data (with a known target variable) to train the model. In unsupervised learning, we don't have known values of the target variable (and the target variable may not even be well defined). Unsupervised learning, instead of looking for a "correct" prediction, looks for natural groups or patterns in data.

Clustering is the unsupervised analogue of classification. In clustering, we don't know what the class labels are, and we don't necessarily know in advance what the class variable is supposed to represent.

In this homework, we'll implement a common clustering algorithm called k-means, which sorts points into clusters by their *centroids* – which is just a fancy word for the means of vectors. Each point gets assigned to the centroid that it's closest to. We give the algorithm the data and a value of k – the number of clusters we want to find – and it does its best to find centroids that group the data into reasonable clusters.

k-means is an iterative approximation algorithm, which means it starts with an initial guess and then takes steps to improve this guess bit-by-bit, until the steps no longer lead to an improvement. Once the update steps no longer do anything, we're done and we return the centroids we found. We've seen an example of one of these algorithms before, although we didn't implement it ourselves: the `scipy` function `curve_fit` uses the same idea.

Instructions. Create a module named `hw4.py`. Below is the spec for ten new functions that you must implement, and a few others that you don't have to if you don't want to. Code the new ones up, add the others, and upload your module to the D2L HW4 assignments folder. Unless the spec explicitly instructs you to, don't do any error-checking; assume that valid arguments will be passed.

Testing. Download `hw4_test.py` and auxiliary testing files and put them in the same folder as your `hw4.py` module. Each of the ten functions is worth 10% of your correctness score. You can examine the test module in a text editor to understand better what your code should do if necessary; consider the test module to be part of the spec.

Documentation. Your module must contain a header docstring containing your name, your section leader's name, the date, ISTA 331 HW4, and a brief summary of the module. Each function must contain a docstring. Each function docstring should include a description of the function's purpose, the name, type, and purpose of each parameter, and the type and meaning of the function's return value.

Grading. Your module will be graded on correctness, documentation, and coding style. Code should be clear and concise. You will only lose style points if your code is a real mess. Include inline comments to explain tricky lines and summarize sections of code.

Collaboration. Collaboration is allowed. You are responsible for your learning. Depending too much on others will hurt you on the tests. "Helping" others too much harms them in reality. Cite any sources or collaborators in your header docstring. Leaving this out is dishonest.

Resources. Here are some references that might be helpful:

Pandas and scikit-learn docs:

- `http://pandas.pydata.org/pandas-docs/stable/api.html`
- `http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`
- `http://scikit-learn.org/stable/modules/clustering.html#k-means`
- `http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html#sklearn.preprocessing.MinMaxScaler`
- `http://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_assumptions.html`
- `http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-scaler`
- `http://scikit-learn.org/stable/`

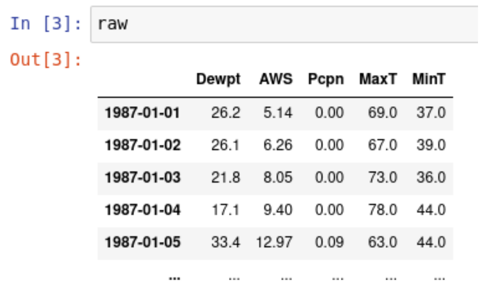
About the data we're working with:

- `https://www.ncdc.noaa.gov/`
- `https://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/global-historical-climatology-network-ghcn`
- `https://www.ncdc.noaa.gov/ghcn-daily-description`
- `https://www.ncdc.noaa.gov/cdo-web/search?datasetid=GHCND`
- `https://catalog.data.gov/dataset/global-surface-summary-of-the-day-gsod`

Function Specifications.

Functions for loading, cleaning, and processing data.

- `is_leap_year`: takes an integer year and returns `True` if that year is a leap year, `False` otherwise. (Look up when leap years occur if you don't know.)
- `euclidean_distance`: takes two feature vectors and returns the Euclidean distance between them.
- `make_frame`: This function doesn't need to take any arguments. Make a `DataFrame` by calling `read_csv` and passing it the `TIA_1987_2016.csv` filename. Replace its index with `datetime` objects by assigning a pandas `date_range` object to the index, and then return the frame. Your returned frame should look something like the following:

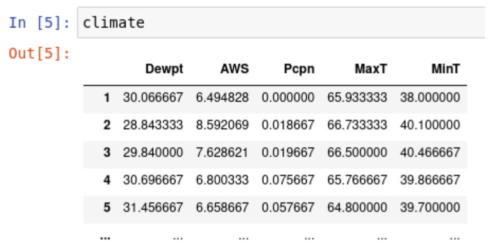


```
In [3]: raw
Out[3]:
```

	Dewpt	AWS	Pcpn	MaxT	MinT
1987-01-01	26.2	5.14	0.00	69.0	37.0
1987-01-02	26.1	6.26	0.00	67.0	39.0
1987-01-03	21.8	8.05	0.00	73.0	36.0
1987-01-04	17.1	9.40	0.00	78.0	44.0
1987-01-05	33.4	12.97	0.09	63.0	44.0
...	...	...	...	...	...

- `clean_dewpoint`: This function takes the frame created by `make_frame`. Looking at our datafile, we see that the dewpoints for March 10th and 11th, 2010, are `-999`. Those dates are missing in the GSOD data, so we have added them manually. Replace those values in the `DataFrame` with the average of the dewpoints on those days for the other 29 years. You can ignore any other missing data (`pandas` has replaced them with `NaN`s, and that's ok here).
- `day_of_year`: This function takes a `datetime` object and returns the day of the year it represents as an `int` between 1 and 365. The nontrivial part: if the year is a leap year, return the day of the year as though it were not, unless the date is February 29; in that case, return `366`. (Hint: lookup the `timetuple` method for `datetime` objects. It returns an object with a useful instance variable, `tm_yday`. Of course, you need to recall how to access an instance variable.)
- `climatology`: This function takes the data frame we have created and uses it to calculate 30-year averages of our feature variables for each day of the year. Instead of calculating these averages manually, we will use a piece of `pandas` functionality: the `groupby` method. This is a `DataFrame` method that takes a function as an argument.[1]

  Call the `groupby` method, passing it your `day_of_year` function. This will return a `GroupBy` object. All you need to know about this `GroupBy` object is that you can call its `mean` method and it will return a `DataFrame` indexed by the days of the year (1-365) whose values are the averages of the values in the original frame; then you can return this new frame. The frame should look something like this:



```
In [5]: climate
Out[5]:
```

	Dewpt	AWS	Pcpn	MaxT	MinT
1	30.066667	6.494828	0.000000	65.933333	38.000000
2	28.843333	8.592069	0.018667	66.733333	40.100000
3	29.840000	7.628621	0.019667	66.500000	40.466667
4	30.696667	6.800333	0.075667	65.766667	39.866667
5	31.456667	6.658667	0.057667	64.800000	39.700000
...	...	...	...	...	...

- `scale`: This function takes a `DataFrame` and scales each of the features so that they run from 0 to 1. This is similar in spirit to calculating z-scores for each value, and often improves the performance of k-means; without this step, variables with higher spread would dominate the distance calculation.

  To save a bit of effort we will use some canned code from `scikit-learn`, a machine learning module. Make sure to include this import:

  ```
  from sklearn.preprocessing import MinMaxScaler
  ```

  Then, in the `scale` function, instantiate a `MinMaxScaler` object with a line like

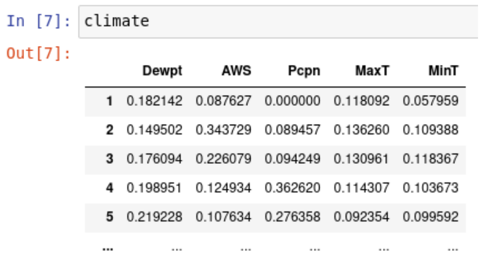
  ```
  scaler = MinMaxScaler(copy = False)
  ```

[1]We first met the concept of passing a function as an argument in HW 3, in the `r_squared` function. The property of Python that allows us to do this is called "first-class functions", and is shared by many, but not all, modern programming languages.

and call its `fit_transform` method, passing it the frame you want to scale:

`scaler.fit_transform(df)`

(substituting the name of your argument for `df` if necessary). Your frame should now look like this:



```
In [7]: climate
Out[7]:
```

	Dewpt	AWS	Pcpn	MaxT	MinT
1	0.182142	0.087627	0.000000	0.118092	0.057959
2	0.149502	0.343729	0.089457	0.136260	0.109388
3	0.176094	0.226079	0.094249	0.130961	0.118367
4	0.198951	0.124934	0.362620	0.114307	0.103673
5	0.219228	0.107634	0.276358	0.092354	0.099592
...	...	...	...	...	...

Functions implementing the k-means algorithm. As we saw in class, k-means uses a predict-update loop (a common technique in learning algorithms). Starting from initial guesses, the algorithm makes predictions, then uses those predictions to update its guesses, then uses the new guesses to make predictions, etc., etc. We alternate between predict and update steps until the update no longer changes the predictions, at which point we're done.

- `get_initial_centroids`: The k-means algorithm needs an initial guess for the centroids to start running. In common practice these guesses are chosen randomly, and the algorithm is run a few times with different starting points to try to find the best result. But we'll just do a *non-random* process and pick k evenly spaced days.

  This function should take a climatology frame and a value of k and return a `DataFrame` indexed with standard integer indexing (`0, 1, ..., k - 1`). Row `i` of this frame should contain the values from row `i * (len(df) // k) + 1` of the climatology frame. (the +1 is important because our climate frame is indexed from 1, not 0).
- `classify`: this function takes a centroids frame and a feature vector (i.e. a row from the climatology frame) and returns the label of the cluster whose centroid is closest.
- `get_labels`: This function takes a `DataFrame` (in our case, intended to be a scaled climatology frame) and a labels series and returns a `Series` that maps the indices (days of year) from the first argument to the labels of the cluster those days belong to. In other words, for each day in the frame, get the closest centroid to that day (use the `classify` function you just defined) and map that day to that centroid's label. When this is done, return the `Series`. (This is the 'predict' step.)
- `update_centroids`: Here is the 'update' step, probably the trickiest part. This function takes the `DataFrame`, a centroids frame, and a labels series. It replaces the existing values in the centroids frame with the averages of the clusters according to the labels. You'll have to think carefully about the relationships between the three inputs to do this calculation.
- `k_means`: Finally we have all the components. Most of the hard work is done, so we just have to build the predict-update loop. `k_means` should take a `DataFrame` and a value of k. Get k initial centroids, get the initial labels, and then run a loop until the algorithm stabilizes (i.e. the update step doesn't change anything).

  The following pseudocode can be a guide:

```
centroids = get the initial centroids
labels = get labels from initial centroids
loop until done:
    update centroids using labels
    get new labels from the updated centroids
    check if the old labels == new labels # if they're the same, we're done!
```

Once you have completed this, you've completed the graded portion of the assignment.

The next few functions let us evaluate the creature we've created. I've included implementations of these below but it would be a good exercise to write your own if you have some extra time (ha).

- `distortion`: This function measures how well the clustering detected by $k$-means fits the data. It takes a (scaled) `DataFrame`, a labels `Series`, and a centroids frame. It returns the sum, over all clusters, of the sum of distances from points in the cluster to its centroid. This is a double sum – a sum of a sum. Mathematically, it would be written like this:

$$\sum_{i=0}^{k-1} \sum x \in C_i d(x, \bar{x}_i)$$

where $\bar{x}_i$ is the centroid of the ith cluster.
- `list_of_kmeans`: This function takes a frame and a maximum $k$ and returns a list of $k$-means dictionaries for $k$ running from 1 to the max $k$. Each $k$-means dictionary maps `'centroids'` to the final centroids frame, `'labels'` to the final labels `Series`, `'k'` to the value of $k$, and `'distortion'` to the distortion for that $k$.
- `extract_distortion_dict`: This function takes a $k$-means list and returns a dictionary mapping values of $k$ to their distortion values.
- `plot_clusters`: This function takes a frame and a label `Series` and creates a grid of scatterplots, plotting each combination of two variables against one another, and plotting points from different clusters in different colors (and/or with different shaped points). This allows us to visualize the clusters that our model has found.

To evaluate the $k$-means clustering, open an `iPython` shell with

```
ipython --matplotlib
```

Import your code using

```
from hw4 import *
```

and use the following sequence of commands to create a plot of the distortion:

```
raw = make_frame()
clean_dewpoint(make_frame)
climo = climatology(raw)
scale(climatology)
list_of_kmeans = kmeans(climo, 10) # This will probably take a while
distortion_dict = extract_distortion_dict(list_of_kmeans)
distortion_series = pd.Series(distortion_dict)
ax = distortion_series.plot()
ax.set_ylabel('Distortion', size = 24)
ax.set_xlabel('$k$', size = 24)
```

Distortion isn't always a great measure of clustering quality, because it doesn't penalize higher values of $k$ in any way, so it always decreases when $k$ goes up.

Sometimes it is useful to apply the knowledge we have about the quantities we're working with. What values of $k$ does your intuition tell you might be making particularly informative clusterings?

Why? (Remember the meaning of the data we are working with – how many natural “groups” do you think there should be?)

As another exercise, compute the $k = 4$ clustering and plot the cluster labels against the day of the year. What does this tell you?

I have uploaded a file called `cluster_eval.py` that contains my implementations of these four functions, so you can use those if you don’t have time to write your own.