Cooking with Machine Learning: Dimension Reduction

Recently I came across this cooking recipes data set in Kaggle, and it inspired me to combine 2 of my main interests in life. Food and machine learning.

Diego Toledo Aug 1, 2018 · 6 min read ★

What makes this data set special is that it contains recipes from 20 different cuisines, 6714 different ingredients, but only 26648 samples. Some cuisines have way fewer recipes than others.

This is way too many features for this amount of data. The first step before working with this data set, should be reducing its dimensions. And in this post I will show how to use PCA to reduce those 6714 ingredients into a latent space with only 700 dimensions. And as a bonus we will use this model as an anomaly detector.

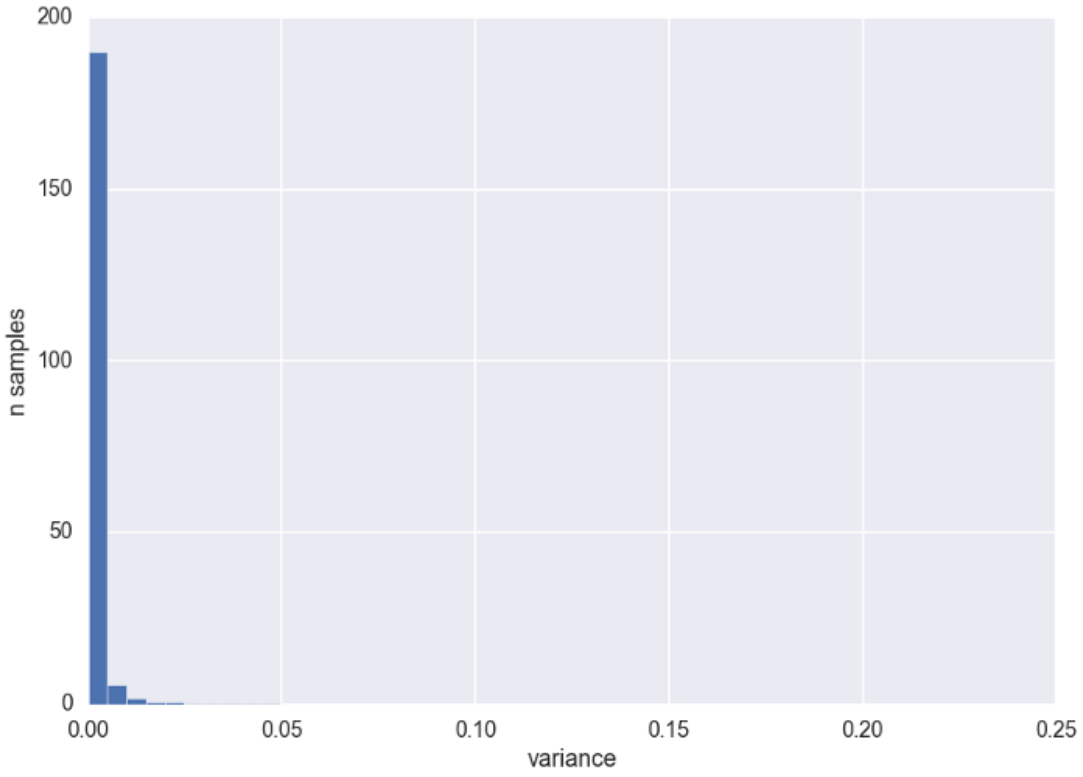
Here is how the data looks like:

```
{
 "id": 24717,
 "cuisine": "indian",
 "ingredients": [
     "tumeric",
     "vegetable stock",
     "tomatoes",
     "garam masala",
     "naan",
     "red lentils",
     "red chili peppers",
     "onions",
     "spinach",
     "sweet potatoes"
 ]
 },
```

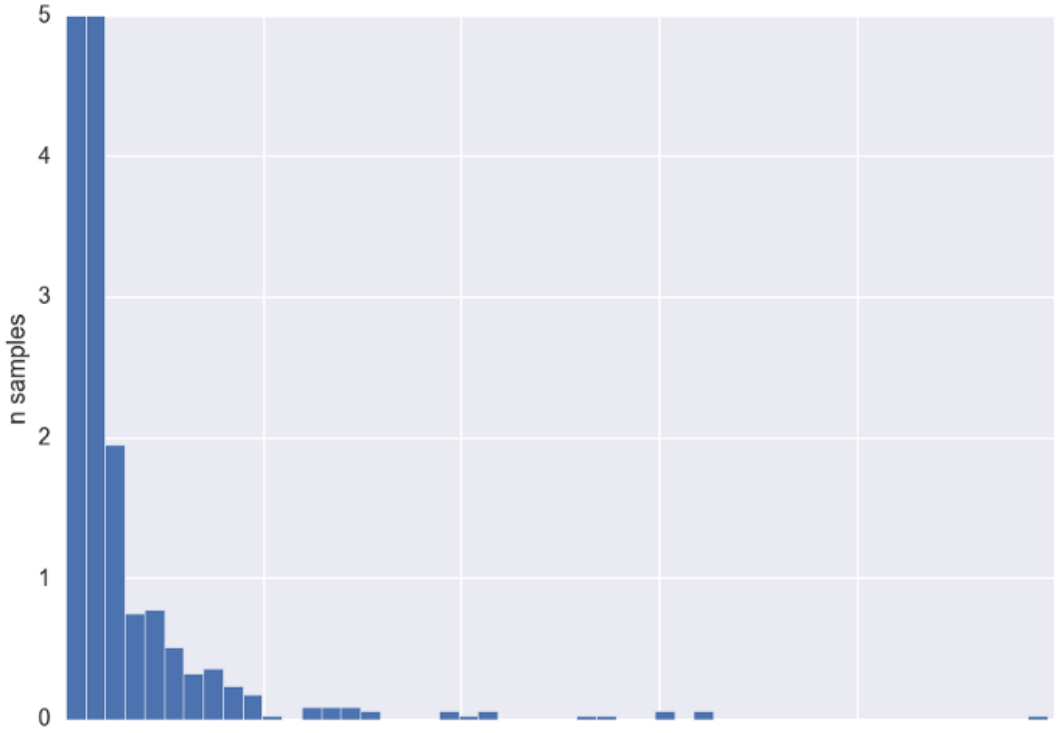
The most challenging aspect is that it is very sparse, here is the breakdown of ingredients per recipe:

```
mean 10.76771257605471
min 1   #rice!
max 65 #complicated fettuccine recipe
std 4.428921893064523
```

Another way to see it, is to check this variance histogram:



And if we zoom a bit:



Which means that in average, each row of 6714 features has only 10 features active. Way less than 1%. This should make things hard to split the data into a training and test set. Because of that, you are very likely to end up with recipes with completely different patterns in both sets. A good way to address this issue would be to k-fold the data, but not in this case. The data is too sparse, it would not improve much.

For a sparse data set with so many features, a first step is usually to reduce the number of dimensions. Haven't you heard of the dimensionality curse?

For this post I will be using a very popular method to reduce dimensions: PCA

Transforming the data

Time to get busy! Let's do some basic transformation on the data. The main idea here is that because we have qualitative data, we need to do something called one-hot-encoding. Long story short: 6714 ingredients -> 6714 columns. When one ingredient is present in a recipe, its column goes to 1. All the rest stays as a 0. In average only 10 of those columns will be 'active' in each row. This code will create the "transformer", that will get an ingredient and output its vector representation

```
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from numpy import array
import json

f = open('recipes_train.json', 'r')
recipes_train_txt = f.read()
recipes_train_json = json.loads(recipes_train_txt)

#get list of ingredients
ingredients = set()
ingredients_matrix = []

for recipe in recipes_train_json:

    ingredients_matrix.append(recipe["ingredients"])
    for ingred in recipe["ingredients"]:

        ingredients.add(ingred)

ingredients = list(ingredients)

ingredients.sort() #it made my life easier to have it sorted when i
needed to check what is what in the encoded vector

values = array(ingredients)

label_encoder = LabelEncoder()

#gives a unique int value for each string ingredient, and saves the
#mapping. you need that for the encoder. something like:
#['banana'] -> [1]
```

```
integer_encoded = label_encoder.fit_transform(values)

onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)

#here you encode something like : [2] -> [0,1,0,0,...]
onehot_encoded = onehot_encoder.fit_transform(integer_encoded)

def transform_value(s):

    l = array([s])
    integer_encoded = label_encoder.transform(l)
    integer_encoded = integer_encoded.reshape(len(integer_encoded),
1)
    onehot_encoded = onehot_encoder.transform(integer_encoded)

    return onehot_encoded[0]
```

This code gives us an encoder that will get a ingredient (string) as input and output its vector representation. The final vector containing all the recipe's ingredients will be the result of a 'logical or' on every one of those ingredient vectors

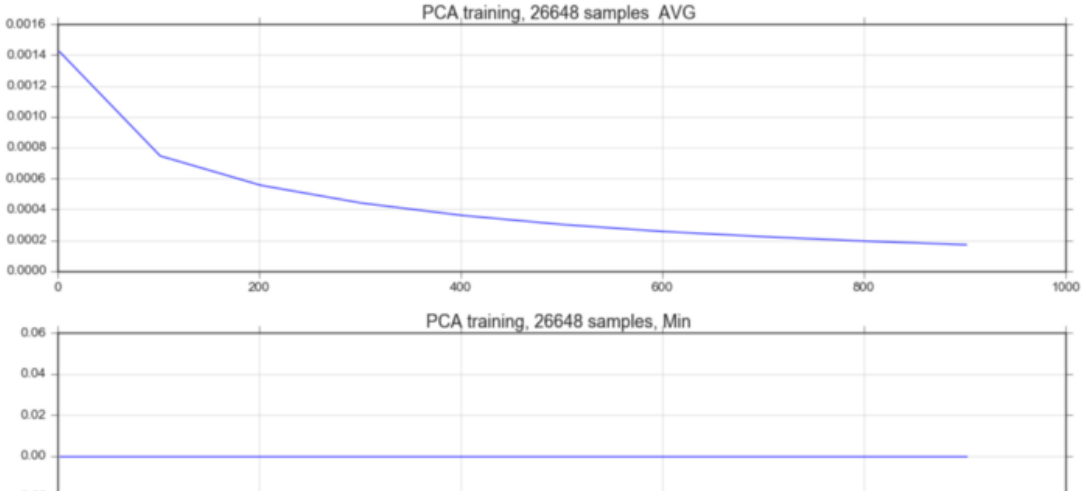
PCA

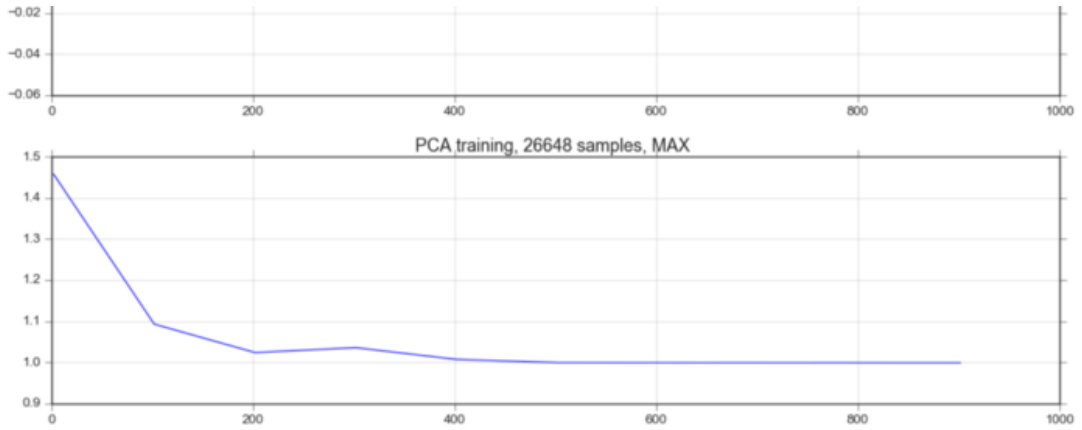
PCA is a very popular choice. It is used as a preprocessing tool before feeding the new reduced data set to be visualized with t-sne but also is the tool you may want to use to reduce your features before feeding into a machine learning algorithm.

This is desirable because the more features you have, the more data you will need, and slower is the learning process. So, keeping things small will boost your performance.

But before minimizing the data, you need to make one call: how small you want that? There is a tradeoff here, the smaller you go, more info you lose. You can measure that by using the same trained model you use to minimize the data, to later maximize back to the original size. After that you can compare the 2 samples and measure how different they are (remember, you lose info when you go down).

So, let's just train a bunch of different models, and pick one with very few features but with a low reconstruction error.





In the X axis we have the number of component vectors, while in the Y axis is the reconstruction error for the whole sample (using L2). Those results look good right? But let's dig deeper here. 700 seems to be a safe number to pick, there isn't much improvement around that area. It is already a huge improvement from 6714 features. Let's compare with some unseen data, the test set.

Here we can see that PCA did a decent job in generalizing the structure of the data. The train mean square error ~= 0.000171% and the **test mean square error ~= 0.0002431%**. Not bad.

We know from before that the data has in average 10 ingredients, standard deviation of 4.42. What if we create some 'random recipes' using that distribution (picking ingredients at random)? This can be achieved using a gaussian generator. If PCA learned anything, we should be seeing some major reconstruction errors.

```
for n_candidates in range(N_CANDIDATES):
    dna = [0 for i in range(N_INGREDIENTS)]

    n_flips = int(round(random.gauss(mu=MEAN, sigma=STD)))
    indexes_to_flip = [ random.randint(0, N_INGREDIENTS-1 ) for i in
range(n_flips) ]

    for i in range(n_flips):

        dna[ indexes_to_flip[i]  ] = 1

    BAD_DATA.append(dna)
```

Let's see how the model does with this fake data.

That was not the initial goal here, but looks like we got a nice model to detect anomaly recipes. If we set a threshold to 0.0004, and consider anything with a reconstruction error bigger than that an anomaly, we get the following matrix:

Conclusion

We reduced this data set from 6714 features to only 700.

We can conclude now that the model did learn something from the training set. We tried to trick the PCA model and we learned that some ingredients usually come together and some don't mix. You can also use this model as an anomaly detection, where the bad recipes are anomalies (you should not eat those!). Maybe as a follow up project I can try to take advantage of this 'learning'.

As you can see, there is a pattern among the different cuisines. We already have a model to detect anomaly recipes that do not fall in any of those patterns, how hard would it be to generate new recipes? Would it be possible to create new french food?

Machine Learning Pca Anomaly Detection Dimensionality Reduction Scikit