Dimensionality Reduction with PCA

Introduction

We will be using customer data from a Portuguese wholesale distributor for clustering. This data file is called `Wholesale_Customers_Data`.

It contains the following features:

- Fresh: annual spending (m.u.) on fresh products
- Milk: annual spending (m.u.) on milk products
- Grocery: annual spending (m.u.) on grocery products
- Frozen: annual spending (m.u.) on frozen products
- Detergents_Paper: annual spending (m.u.) on detergents and paper products
- Delicatessen: annual spending (m.u.) on delicatessen products
- Channel: customer channel (1: hotel/restaurant/cafe or 2: retail)
- Region: customer region (1: Lisbon, 2: Porto, 3: Other)

In this data, the values for all spending are given in an arbitrary unit (m.u. = monetary unit).

In [1]:
```
import os, pandas as pd, numpy as np, seaborn as sns, matplotlib.pyplot as plt
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
```

EDA

- Importing the data and checking the data types.
- Droping the channel and region columns as they won't be used since we focus on numeric columns for this example.
- Converting the remaining columns to floats if necessary.
- Copying this version of the data (using the `copy` method) to a variable to preserve it. We will be using it later.

In [3]:
```
data = pd.read_csv("C:\\Users\\rsnen\\Desktop\\IBM\\Wholesale_Customers_Data.csv", sep=',')
```

In [4]:
```
data.shape
```

Out[4]:
```
(440, 8)
```

In [5]:
```
data.head()
```

Out[5]:

	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	2	3	12669	9656	7561	214	2674	1338
1	2	3	7057	9810	9568	1762	3293	1776
2	2	3	6353	8808	7684	2405	3516	7844
3	1	3	13265	1196	4221	6404	507	1788
4	2	3	22615	5410	7198	3915	1777	5185

In [6]:
```
data = data.drop(['Channel', 'Region'], axis=1)
```

In [7]:
```
data.dtypes
```

Out[7]:
```
Fresh               int64
Milk                int64
Grocery             int64
Frozen              int64
Detergents_Paper    int64
Delicassen          int64
dtype: object
```

In [8]:
```
# Convert to floats
for col in data.columns:
    data[col] = data[col].astype(np.float)
```

Preserve the original data.

In [9]:
```
data_orig = data.copy()
```

Scaling data

ensuring the data is scaled and (relatively) normally distributed.

- Examining the correlation and skew.
- Performing any transformations and scale with a scaling method.
- Viewing the pairwise correlation plots of the new data.

Examining the correlation

with data.core we can see the correlation between each one of the different features. So, this will give us for each feature the correlation with all the other features in a square matrix in a square data frame. And just to ensure that we can get the highest correlation which feature is the highest correlation and because of one feature with itself will always have a correlation of one. We're going to replace that diagonal value which are going to start off as all 1's with all 0's. So we're saying for x in the range of format.shape 0, it's a square matrix we could have called shape zero or shape one. So that's going to be for every single value in our matrix, for every single numeric value for the range of our matrix, we're going to take the diagonal value. So 0 0,1 1,2 2 and replace that 1 with a 0. And we can see now our correlation matrix has the correlation between fresh and milk and grocery. And then for fresh and fresh, it's just a 0 across each one of the different diagonals.

In [12]:
```
corr_mat = data.corr()

# Strip the diagonal for future examination
for x in range(corr_mat.shape[0]):
    corr_mat.iloc[x,x] = 0.0

corr_mat
```

Out[12]:

	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
Fresh	0.000000	0.100510	-0.011854	0.345881	-0.101953	0.244690
Milk	0.100510	0.000000	0.728335	0.123994	0.661816	0.406368
Grocery	-0.011854	0.728335	0.000000	-0.040193	0.924641	0.205497
Frozen	0.345881	0.123994	-0.040193	0.000000	-0.131525	0.390947
Detergents_Paper	-0.101953	0.661816	0.924641	-0.131525	0.000000	0.069291
Delicassen	0.244690	0.406368	0.205497	0.390947	0.069291	0.000000

In [25]:
```
#  the two categories with their respective most strongly correlated variable.
corr_mat.abs().idxmax()
```

Out[25]:
```
Fresh                         Frozen
Milk                         Grocery
Grocery             Detergents_Paper
Frozen                    Delicassen
Detergents_Paper             Grocery
Delicassen                      Milk
dtype: object
```

Examing the skew values

Examing skew for each one of our ten values. And then take the long transformation, if necessary, for those that have higher skew. skew's going to be a value with 0 being no skew,positive value being a right skew,and a negative value being a left skew. The higher that value is, the stronger the skew.

Those that have a higher skew and we see here we have these values that have tend to have a higher skew. And for those, we're going to to take the log transformation with each, hopefully creating more normally distributed data. So for calling each one of these log columns.index, so these are, this is our log columns that we just defined is that pandas series for called the index we get each one of these Delicatessen frozen milk and so on, which is going to also match up with each one of our different data columns. So we're going to place those columns in place with the log transformation of those columns.

In [28]:
```
log_columns = data.skew().sort_values(ascending=False) # We sort them from largest to smallest.
log_columns = log_columns.loc[log_columns > 0.75] # log columns that are greater than 0.75.

log_columns
```

Out[28]:
```
Series([], dtype: float64)
```

In [30]:
```
# The log transformations
for col in log_columns.index:
    data[col] = np.log1p(data[col])
```

Scale the data again. Let's use `MinMaxScaler` this time just to mix things up.

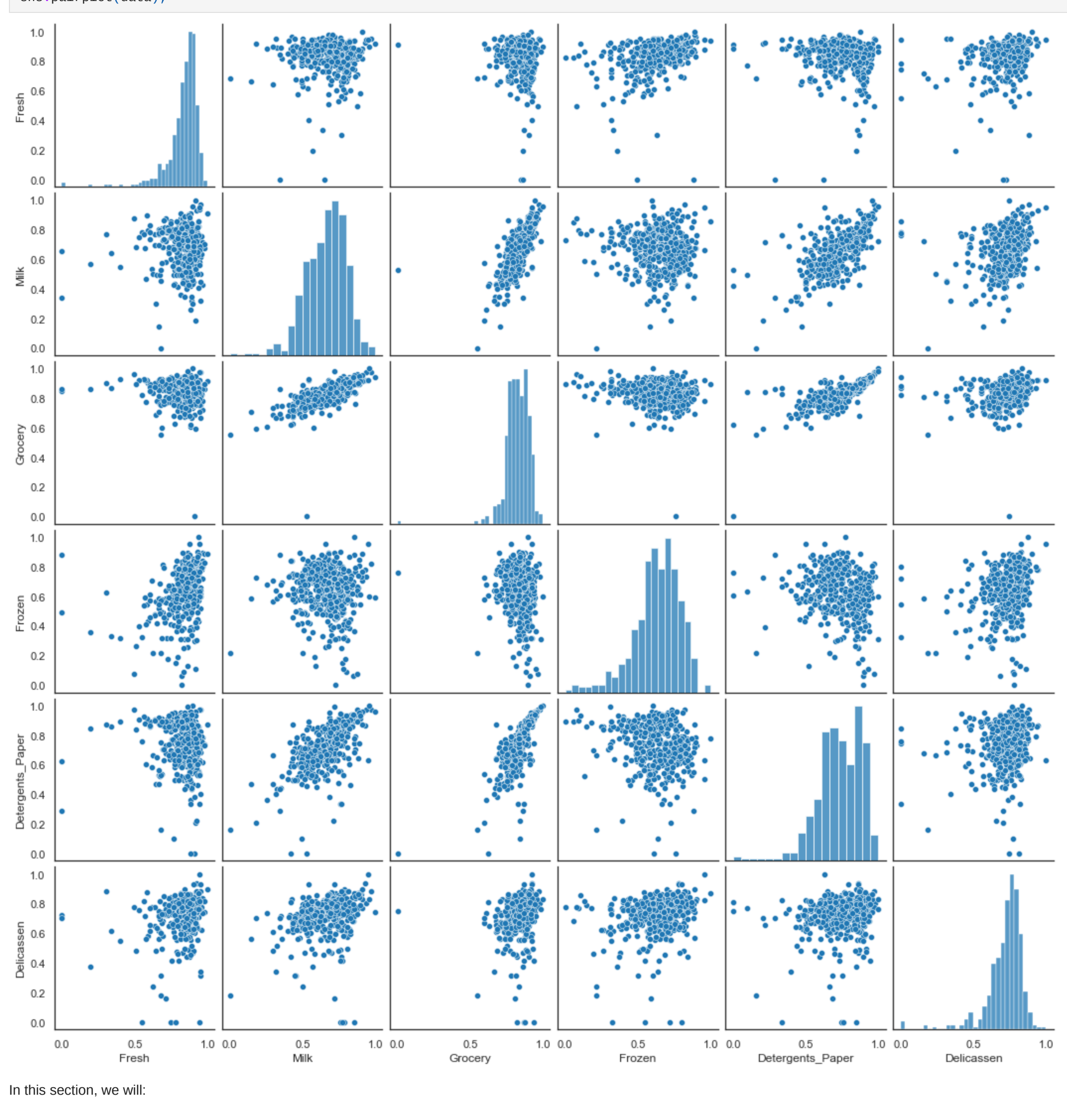
In [33]:
```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()

for col in data.columns:
    data[col] = mms.fit_transform(data[[col]]).squeeze()
```

Visualize the relationship between the variables.

In [34]:
```
sns.set_context('notebook')
sns.set_style('white')
sns.pairplot(data);
```



In this section, we will:

- Use Scikit-learn's pipeline function, recreate the data pre-processing scheme above (transformation and scaling) using a pipeline. If you used a non-Scikit learn function to transform the data (e.g. NumPy's log function), checkout the custom transformer class called `FunctionTransformer`.
- Use the pipeline to transform the original data that was stored at the earlier.
- Compare the results to the original data to verify that everything worked.

Note: Scikit-learn has a more flexible `Pipeline` function and a shortcut version called `make_pipeline`. Either can be used. Also, if different transformations need to be performed on the data, a `FeatureUnion` can be used.

In [37]:
```
from sklearn.preprocessing import FunctionTransformer
from sklearn.pipeline import Pipeline

# The custom NumPy log transformer
log_transformer = FunctionTransformer(np.log1p)

# The pipeline
estimators = [('log1p', log_transformer), ('minmaxscale', MinMaxScaler())]
pipeline = Pipeline(estimators)

# Convert the original data
data_pipe = pipeline.fit_transform(data_orig)
```

The results are identical. Note that machine learning models and grid searches can also be added to the pipeline (and in fact, usually are.)

In [39]:
```
#data pipe should equal that data that we just transformed.
#So we're going to check that using NumPy.allclose,
#which is just going to check that each value within each of our arrays are exactly the same,

np.allclose(data_pipe, data)
```

Out[39]:
```
False
```

PCA

- Performing PCA with `n_components` ranging from 1 to 5.
- Storing the amount of explained variance for each number of dimensions.
- Also store the feature importance for each number of dimensions. *Hint:* PCA doesn't explicitly provide this after a model is fit, but the `components_` properties can be used to determine something that approximates importance. How you decided to do so is entirely up to you.
- Plot the explained variance and feature importances.

In [44]:
```
from sklearn.decomposition import PCA
n = 2
pd.Series({'n':n, 'model':PCAmod,
                                 'var': PCAmod.explained_variance_ratio_.sum()})
```

Out[44]:
```
n                         2
model    PCA(n_components=2)
var                0.695652
dtype: object
```

In [50]:
```
PCAmod.explained_variance_ratio_
```

Out[50]:
```
array([0.40631534, 0.28933681, 0.12372076, 0.09573864, 0.0655913 ])
```

In [52]:
```
PCAmod.explained_variance_ratio_.sum()
```

Out[52]:
```
0.9807028483254034
```

In [55]:
```
from sklearn.decomposition import PCA
n = 2
pd.Series({'n':n, 'model':PCAmod,
                                 'var': PCAmod.explained_variance_ratio_.sum()})
weights = PCAmod.explained_variance_ratio_.reshape(-1,1)/PCAmod.explained_variance_ratio_.sum()
weights
```

Out[55]:
```
array([[0.41431035],
       [0.29503005],
       [0.1261552 ],
       [0.09762247],
       [0.06688193]])
```

In [58]:
```
from sklearn.decomposition import PCA
n = 2
pd.Series({'n':n, 'model':PCAmod,
                                 'var': PCAmod.explained_variance_ratio_.sum()})
weights = PCAmod.explained_variance_ratio_.reshape(-1,1)/PCAmod.explained_variance_ratio_.sum()
overall_contribution = np.abs(PCAmod.components_)*weights
overall_contribution
```

Out[58]:
```
array([[0.03599388, 0.23717225, 0.13522089, 0.10452681, 0.28366318,
        0.06656939],
       [0.10818118, 0.0585261 , 0.00974981, 0.23215918, 0.01446096,
        0.13308716],
       [0.02851832, 0.00452192, 0.01154961, 0.06543015, 0.03408488,
        0.09749418],
       [0.08738228, 0.00401993, 0.00147339, 0.02125841, 0.00918582,
        0.03660402],
       [0.00351675, 0.05111478, 0.00046533, 0.0017493 , 0.04120455,
        0.01212489]])
```

The larger these absolute values are, the more they contributed to each component and the more important that feature is. What we had here before is we took the absolute value, because we don't care about whether it's positive or negative. We just care about how much it affected that principal component. Then we're weighting it according to these weights. weights are going to be how important each one the principal components are. This first one is going to be multiplied by 0.41, and this second one is going to be multiplied by 0.29, so that we don't put on too much weight. We see here that we use 70 percent of whatever feature this is; this is the fifth feature. Then we use 70 percent here in the second feature. The second PCA for a different feature, we want to ensure that these do not get equal weights. This should get a higher weight than this one since this is part of the first principal component.

In [59]:
```
weights
```

Out[59]:
```
array([[0.41431035],
       [0.29503005],
       [0.1261552 ],
       [0.09762247],
       [0.06688193]])
```

In [60]:
```
from sklearn.decomposition import PCA
n = 2
pd.Series({'n':n, 'model':PCAmod,
                                 'var': PCAmod.explained_variance_ratio_.sum()})
weights = PCAmod.explained_variance_ratio_.reshape(-1,1)/PCAmod.explained_variance_ratio_.sum()
overall_contribution = np.abs(PCAmod.components_)*weights
overall_contribution.sum(axis = 0)
# how much each one of these different features with their weightings were able to comprise these
#principal components that we have.
```

Out[60]:
```
array([0.26359241, 0.35535498, 0.15845903, 0.42512385, 0.3825994 ,
       0.34587963])
```

4th feature(0.42512385) was the most important in the first two components if you add up the weights of the first two components

In [66]:
```
from sklearn.decomposition import PCA

pca_list = list()
feature_weight_list = list()

# Fit a range of PCA models

for n in range(1, 6):

    # Create and fit the model
    PCAmod = PCA(n_components=n)
    PCAmod.fit(data)

    # Store the model and variance
    pca_list.append(pd.Series({'n':n, 'model':PCAmod,
                               'var': PCAmod.explained_variance_ratio_.sum()}))

    # Calculate and store feature importances
    weights = PCAmod.explained_variance_ratio_.reshape(-1,1)/PCAmod.explained_variance_ratio_.sum()
    overall_contribution = np.abs(PCAmod.components_)*weights
    abs_feature_values = np.abs(PCAmod.components_).sum(axis=0)
    feature_weight_list.append(pd.DataFrame({'n':n,
                                             'features': data.columns,
                                             'values':abs_feature_values/abs_feature_values.sum()}))

pca_df = pd.concat(pca_list, axis=1).T.set_index('n')
pca_df
```

Out[66]:

	model	var
n		
1	PCA(n_components=1)	0.406315
2	PCA(n_components=2)	0.695652
3	PCA(n_components=3)	0.819373
4	PCA(n_components=4)	0.915112
5	PCA(n_components=5)	0.980703

In [69]:
```
feature_weight_list[1]
```

Out[69]:

	n	features	values
0	2	Fresh	0.114290
1	2	Milk	0.194238
2	2	Grocery	0.090570
3	2	Frozen	0.261864
4	2	Detergents_Paper	0.184878
5	2	Delicassen	0.154159

Create a table of feature importances for each data column.

In [67]:
```
features_df = (pd.concat(feature_weight_list)
               .pivot(index='n', columns='features', values='values'))

features_df
```

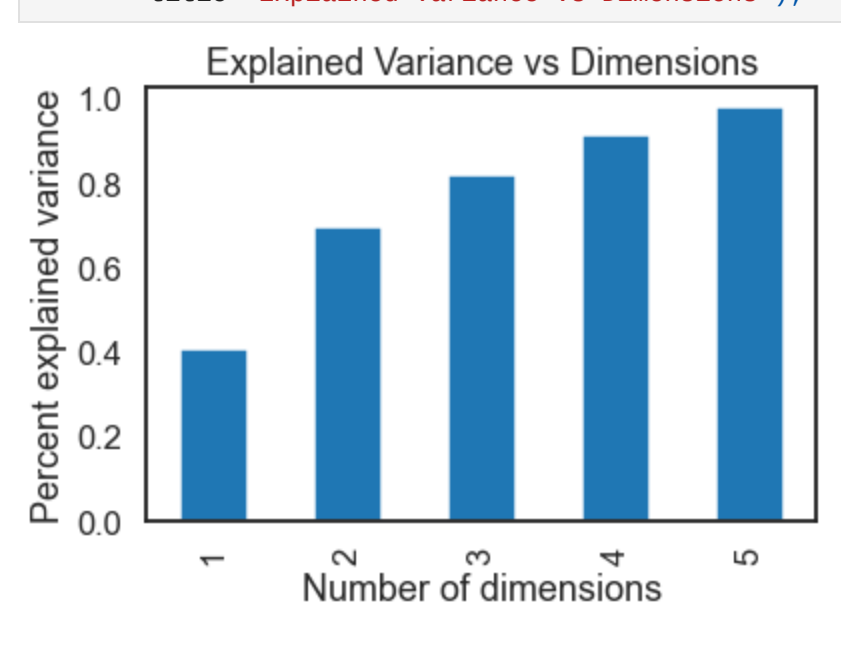
Out[67]:

features	Delicassen	Detergents_Paper	Fresh	Frozen	Grocery	Milk
n						
1	0.077124	0.328639	0.041701	0.121100	0.156660	0.274776
2	0.154159	0.184878	0.114290	0.261864	0.090570	0.194238
3	0.235332	0.170622	0.115511	0.264779	0.076650	0.137106
4	0.233928	0.145971	0.209356	0.236063	0.061963	0.112720
5	0.211672	0.186937	0.177477	0.196504	0.051589	0.175820

Create a plot of explained variances.

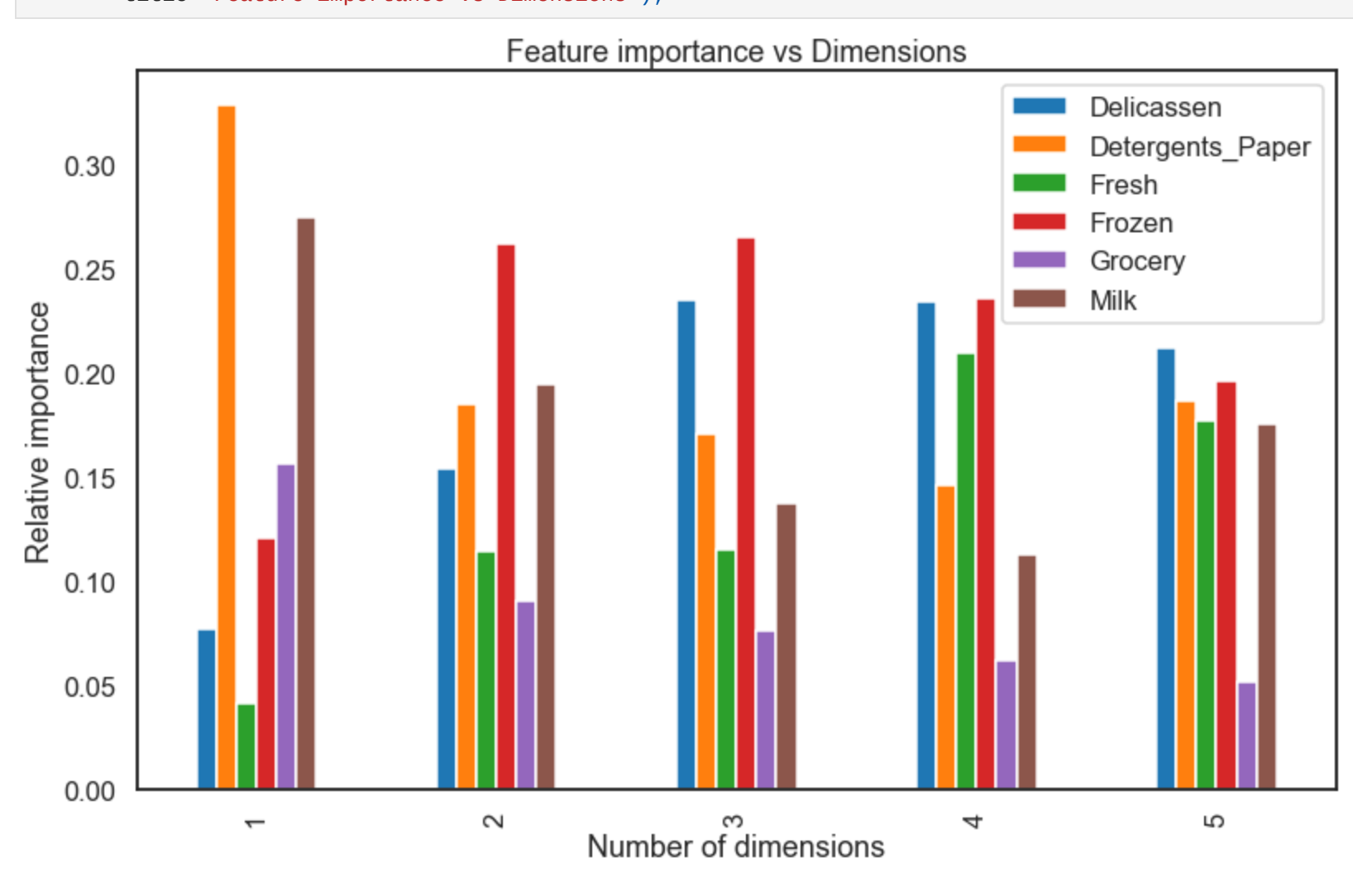
In [71]:
```
sns.set_context('talk')
ax = pca_df['var'].plot(kind='bar')

ax.set(xlabel='Number of dimensions',
       ylabel='Percent explained variance',
       title='Explained Variance vs Dimensions');
```



And here's a plot of feature importances.

In [73]:
```
ax = features_df.plot(kind='bar', figsize=(13,8))
ax.legend(loc='upper right')
ax.set(xlabel='Number of dimensions',
       ylabel='Relative importance',
       title='Feature importance vs Dimensions');
```



Kernel PCA

- Fit a `KernelPCA` model with `kernel='rbf'`. You can choose how many components and what values to use for the other parameters (`rbf` refers to a Radial Basis Function kernel, and the `gamma` parameter governs scaling of this kernel and typically ranges between 0 and 1). Several other kernels can be tried, and even passed ss cross validation parameters (see this example).
- If you want to tinker some more, use `GridSearchCV` to tune the parameters of the `KernelPCA` model.

The second step is tricky since grid searches are generally used for supervised machine learning methods and rely on scoring metrics, such as accuracy, to determine the best model. However, a custom scoring function can be written for `GridSearchCV`, where larger is better for the outcome of the scoring function.

What would such a metric involve for PCA? What about percent of explained variance? Or perhaps the negative mean squared error on the data once it has been transformed and then inversely transformed?

In [76]:
```
from sklearn.decomposition import KernelPCA
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error

# Custom scorer--use negative rmse of inverse transform
def scorer(pcamodel, X, y=None):

    try:
        X_val = X.values
    except:
        X_val = X

# try and accept is just we want to ensure that we are working with NumPy array rather than working with Pandas dataframe.
#So if this x is equal to a pandas data frame, we call that values and we're working with the array.
#If it's already an array, then they'll just say x val to that array.

    # Calculate and inverse transform the data
    data_inv = pcamodel.fit(X_val).transform(X_val)
    data_inv = pcamodel.inverse_transform(data_inv)

    # The error calculation
    mse = mean_squared_error(data_inv.ravel(), X_val.ravel())

    # Larger values are better for scorers, so take negative value
    return -1.0 * mse

# The grid search parameters
param_grid = {'gamma':[0.001, 0.01, 0.05, 0.1, 0.5, 1.0],
              'n_components': [2, 3, 4]}

# The grid search
kernelPCA = GridSearchCV(KernelPCA(kernel='rbf', fit_inverse_transform=True),
                         param_grid=param_grid,
                         scoring=scorer,
                         n_jobs=-1)


kernelPCA = kernelPCA.fit(data)

kernelPCA.best_estimator_
```

Out[76]:
```
KernelPCA(fit_inverse_transform=True, gamma=1.0, kernel='rbf', n_components=4)
```

Let's explore how our model accuracy may change if we include a `PCA` in our model building pipeline. Let's plan to use sklearn's `Pipeline` class and create a pipeline that has the following steps:

1. A scaler
2. $PCA(n_compo \neq nts = n)$
3. $\mathrm{Log}isticRegression$

- Write a function that takes in a value of `n` and makes the above pipeline, then predicts the "Activity" column over a 5-fold StratifiedShuffleSplit, and returns the average test accuracy
- For various values of n, call the above function and store the average accuracies.
- Plot the average accuracy by number of dimensions.

In [81]:
```
data = pd.read_csv("C:\\Users\\rsnen\\Desktop\\IBM\\Human_Activity_Recognition_Using_Smartphones_Data.csv", sep=',')
```

In [83]:
```
data.columns
```

Out[83]:
```
Index(['tBodyAcc-mean()-X', 'tBodyAcc-mean()-Y', 'tBodyAcc-mean()-Z',
       'tBodyAcc-std()-X', 'tBodyAcc-std()-Y', 'tBodyAcc-std()-Z',
       'tBodyAcc-mad()-X', 'tBodyAcc-mad()-Y', 'tBodyAcc-mad()-Z',
       'tBodyAcc-max()-X',
       ...
       'fBodyBodyGyroJerkMag-skewness()', 'fBodyBodyGyroJerkMag-kurtosis()',
       'angle(tBodyAccMean,gravity)', 'angle(tBodyAccJerkMean),gravityMean)',
       'angle(tBodyGyroMean,gravityMean)',
       'angle(tBodyGyroJerkMean,gravityMean)', 'angle(X,gravityMean)',
       'angle(Y,gravityMean)', 'angle(Z,gravityMean)', 'Activity'],
      dtype='object', length=562)
```

In [84]:
```
data.shape
```

Out[84]:
```
(10299, 562)
```

lets reduce number of columns with PCA

In [85]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

X = data.drop('Activity', axis=1)
y = data.Activity
sss = StratifiedShuffleSplit(n_splits=5, random_state=42)

def get_avg_score(n):
    pipe = [
        ('scaler', StandardScaler()),
        ('pca', PCA(n_components=n)),
        ('estimator', LogisticRegression(solver='liblinear'))
    ]

    pipe = Pipeline(pipe)
    scores = []
    for train_index, test_index in sss.split(X, y):
        X_train, X_test = X.loc[train_index], X.loc[test_index]
        y_train, y_test = y.loc[train_index], y.loc[test_index]
        pipe.fit(X_train, y_train)
        scores.append(accuracy_score(y_test, pipe.predict(X_test)))
    return np.mean(scores)


ns = [10, 20, 50, 100, 150, 200, 300, 400]
score_list = [get_avg_score(n) for n in ns]
```

In [88]:
```
score_list
```

Out[88]:
```
[0.8444660194174757,
 0.8836893203883495,
 0.9368932038834951,
 0.965242718446602,
 0.9716504854368931,
 0.9786407766990293,
 0.9842718446601942,
 0.9852427184466019]
```

In [90]:
```
sns.set_context('talk')

ax = plt.axes()
ax.plot(ns, score_list)
ax.set(xlabel='Number of Dimensions',
       ylabel='Average Accuracy',
       title='LogisticRegression Accuracy vs Number of dimensions on the Human Activity Dataset')
ax.grid(True)
```