METIS

Day 9: PCA and Regularization

John Navarro
john.navarro@thisismetis.com
https://www.linkedin.com/in/johnnavarro/

METIS

Too many features?

- If you have more columns than rows, you might end up creating a model which is **overfit.**
- **How do you know which features are important?**

METIS

Too many features?

In general, not all of the columns in our dataset may be critical for predicting a given outcome variable because:

1. Several columns may be highly correlated with each other (provide redundant information) and with the outcome variable.

2. Columns may not be correlated at all with the outcome attribute (have no "predictive ability" in terms of what you're trying to predict).

The two strategies for dealing with both of these cases are called **Dimensionality Reduction** and **Regularization.** Both strategies can be used for supervised classification/regression problems, as well as for unsupervised learning and clustering.

METIS

Too many features?

By the end of this lesson you will be able to:

- use PCA (Principal Component Analysis) to reduce the size of a given dataset's features to a strictly smaller number of features
- understand how to interpret the eigenvalues and variance explained by a certain number of PCs after performing PCA
- use Lasso/L1 regularization for both feature selection and general regression/ classification problems
- use Ridge/L2 regularization for regression and classification problems
- explain the difference between L1/L2 regularization

METIS

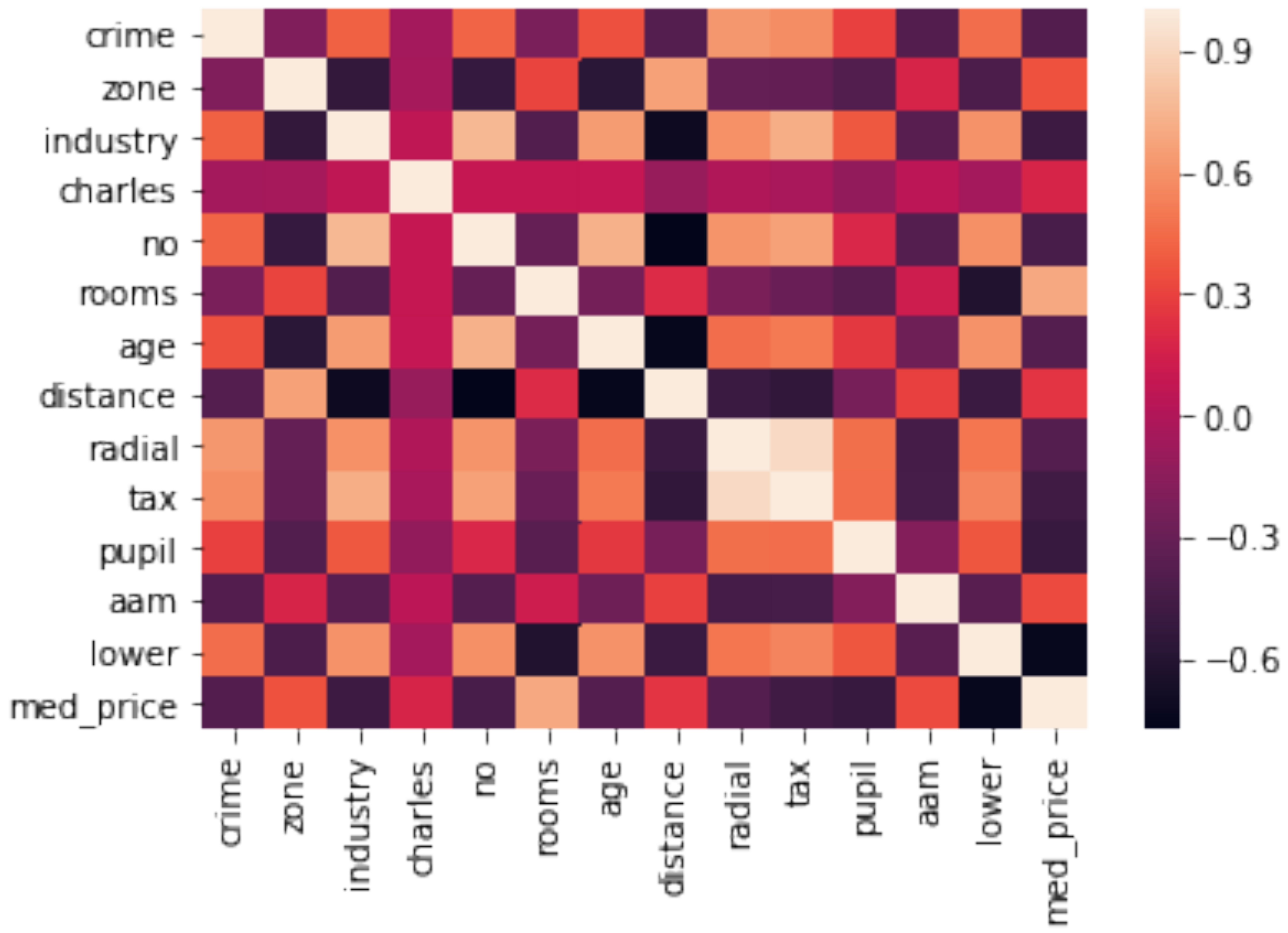
The Dataset

Here's the dataset schema:

1. **crime:** per capita crime rate by town
2. **zone:** proportion of residential land zoned for lots over 25,000 sq.ft.
3. **industry:** proportion of non-retail business acres per town
4. **charles:** Charles River dummy variable (1 if tract bounds river; 0 otherwise)
5. **no:** nitric oxides concentration (parts per 10 million)
6. **rooms:** average number of rooms per dwelling
7. **age:** proportion of owner-occupied units built prior to 1940
8. **distance:** weighted distances to five Boston employment centres
9. **radial:** index of accessibility to radial highways
10. **tax:** full-value property-tax rate per 10K dollars
11. **pupil:** pupil-teacher ratio by town
12. **aam:** where aam is the proportion of african americans by town
13. **lower:** percentage lower income status of the population
14. **med_price:** Median value of owner-occupied homes in $1000's

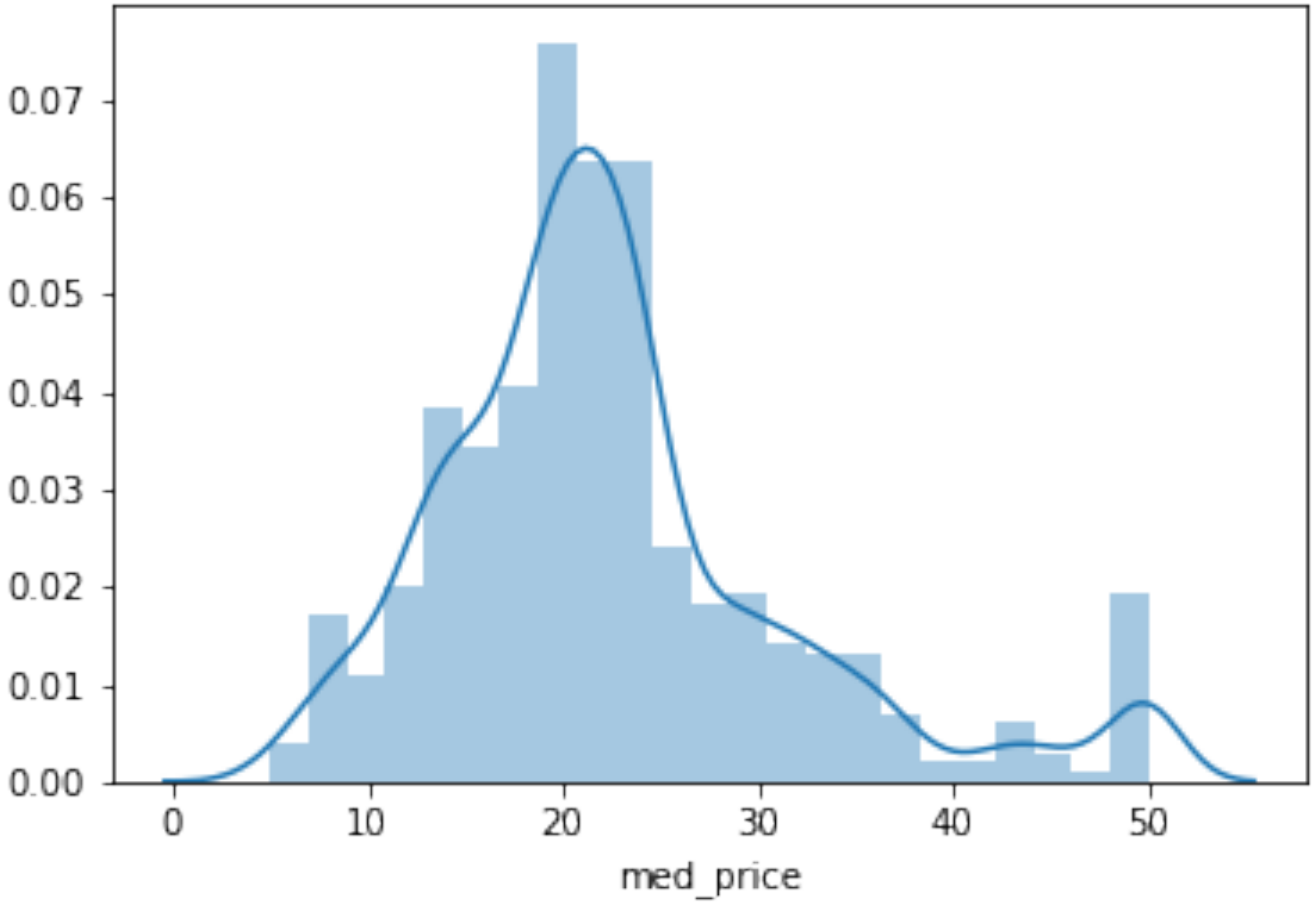
METIS

The Dataset



METIS

The Dataset



METIS

Build a Regression Model

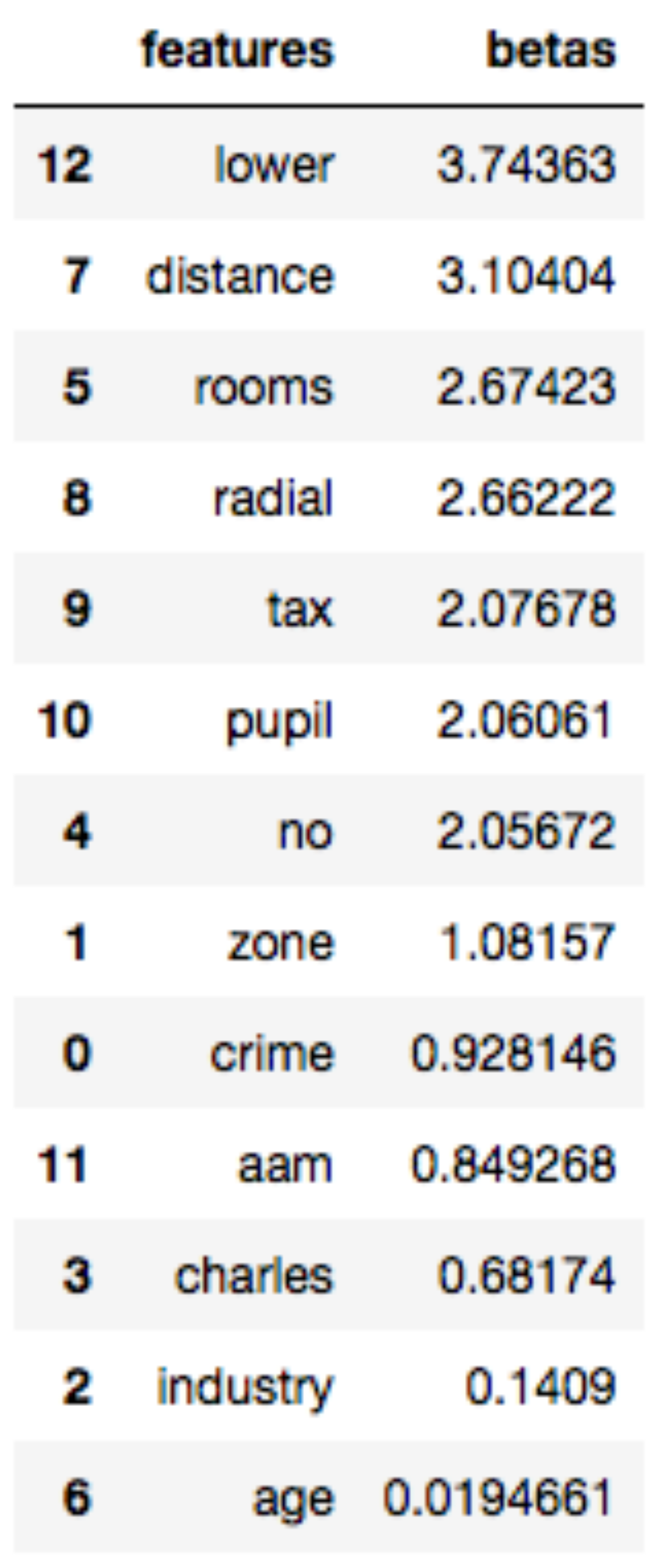
```
X = StandardScaler().fit_transform(housing_data[housing_features])
y = housing_data[housing_target]

lr = LinearRegression()

mean_squared_errors = np.abs(cross_val_score(lr,X,y,cv=50,scoring='neg_mean_squared_error'))
print("50-fold mean RMSE: ", np.mean( np.sqrt(mean_squared_errors)))
```

```
50-fold mean RMSE:  4.44466151015
```

METIS

Build a Regression Model

```
coeffs = LinearRegression().fit(X,y).coef_
coeff_df =
pd.DataFrame(list(zip(housing_features,np.abs(coeffs))),columns=["features","betas"])
coeff_df.sort_values("betas",ascending=False,inplace=True)
```



	features	betas
12	lower	3.74363
7	distance	3.10404
5	rooms	2.67423
8	radial	2.66222
9	tax	2.07678
10	pupil	2.06061
4	no	2.05672
1	zone	1.08157
0	crime	0.928146
11	aam	0.849268
3	charles	0.68174
2	industry	0.1409
6	age	0.0194661

METIS

PCA

PCA, short for Principal Components Analysis is a method for transforming your initial features into a new feature space such that:

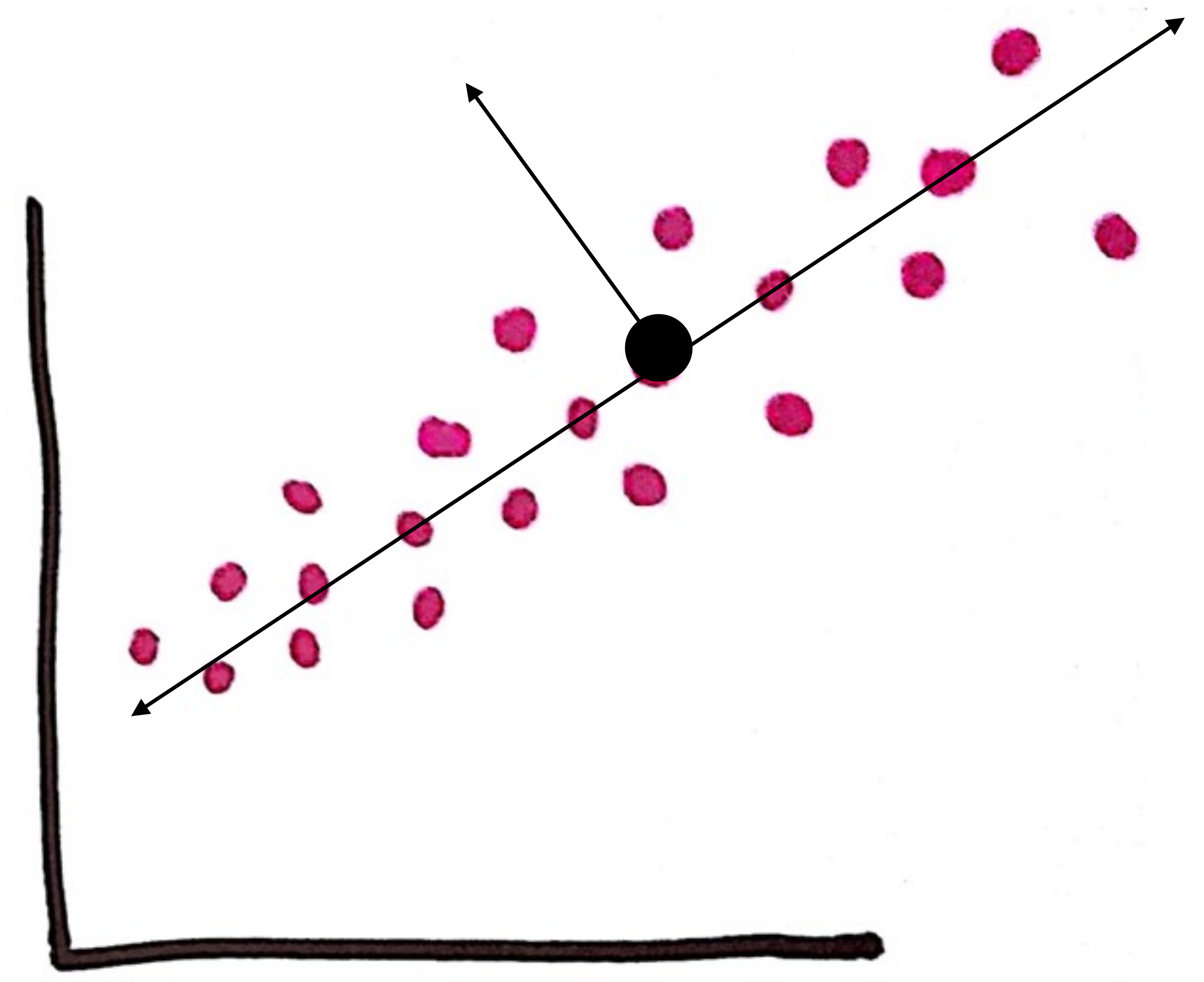
- All of the new columns are uncorrelated with each other, but are linearly-blended versions of the original columns (features)

- Each succesive column is less important in terms of how much of the variance (information) in the original data it explains. So, the first column is the "most important" (in a specific statistical sense) column, the 2nd column is "2nd most important" and so on.

We can generate a PCA-based transformation of our features.

METIS

PCA



- PCA finds a new coordinate system obtained from the old one by translation and rotation only and moves center of coordinate system to center of data.
- It moves the x axis onto the principal axis of variation, where you see the most variation relative to other data points
- It moves further axes down the road into orthogonal, less important directions of variation
- PCA finds for you these axes and also tells you how important these axes are. This allows you to use only the most important principal components when building your model.

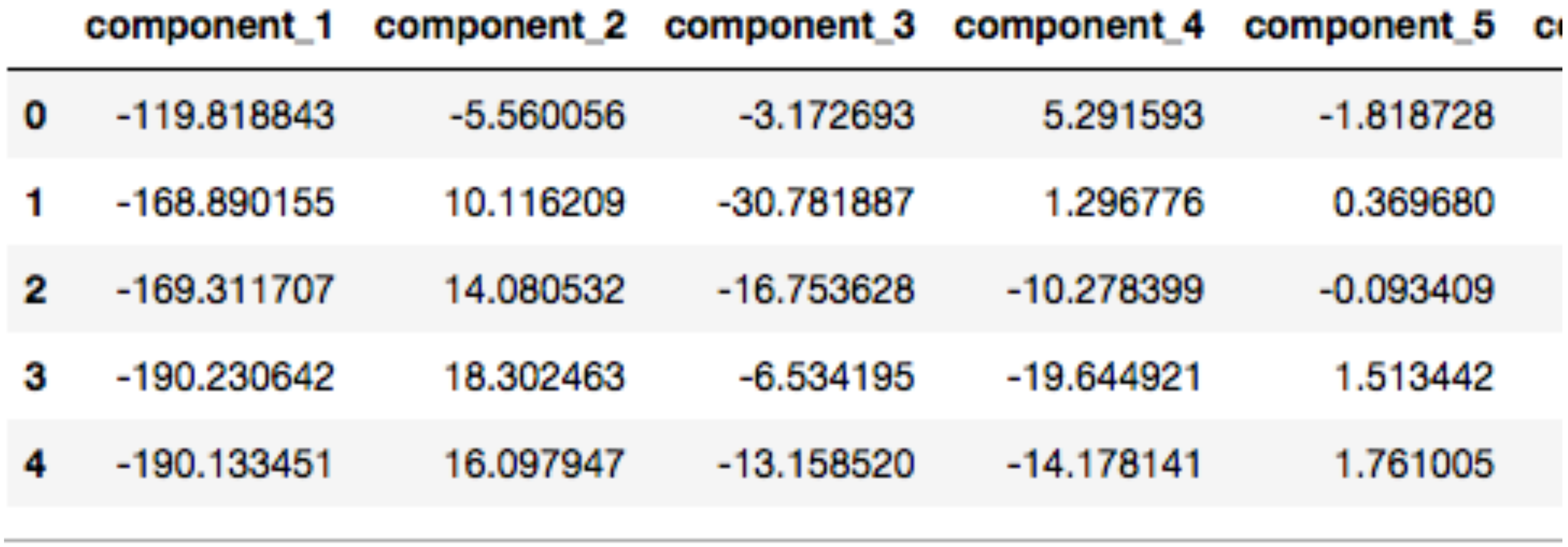
METIS

PCA

```
pca = PCA()
transformed_pca_x = pca.fit_transform(housing_data[housing_features])

#create component indices
component_names = ["component_"+str(comp) for comp in range(1,
len(pca.explained_variance_)+1)]

#generate new component dataframe
transformed_pca_x =
pd.DataFrame(transformed_pca_x,columns=component_names)
transformed_pca_x.head()
```

METIS

PCA



	component_1	component_2	component_3	component_4	component_5	c
0	-119.818843	-5.560056	-3.172693	5.291593	-1.818728	
1	-168.890155	10.116209	-30.781887	1.296776	0.369680	
2	-169.311707	14.080532	-16.753628	-10.278399	-0.093409	
3	-190.230642	18.302463	-6.534195	-19.644921	1.513442	
4	-190.133451	16.097947	-13.158520	-14.178141	1.761005	

METIS

PCA

Once PCA is performed on a given dataset, we get several pieces of information:

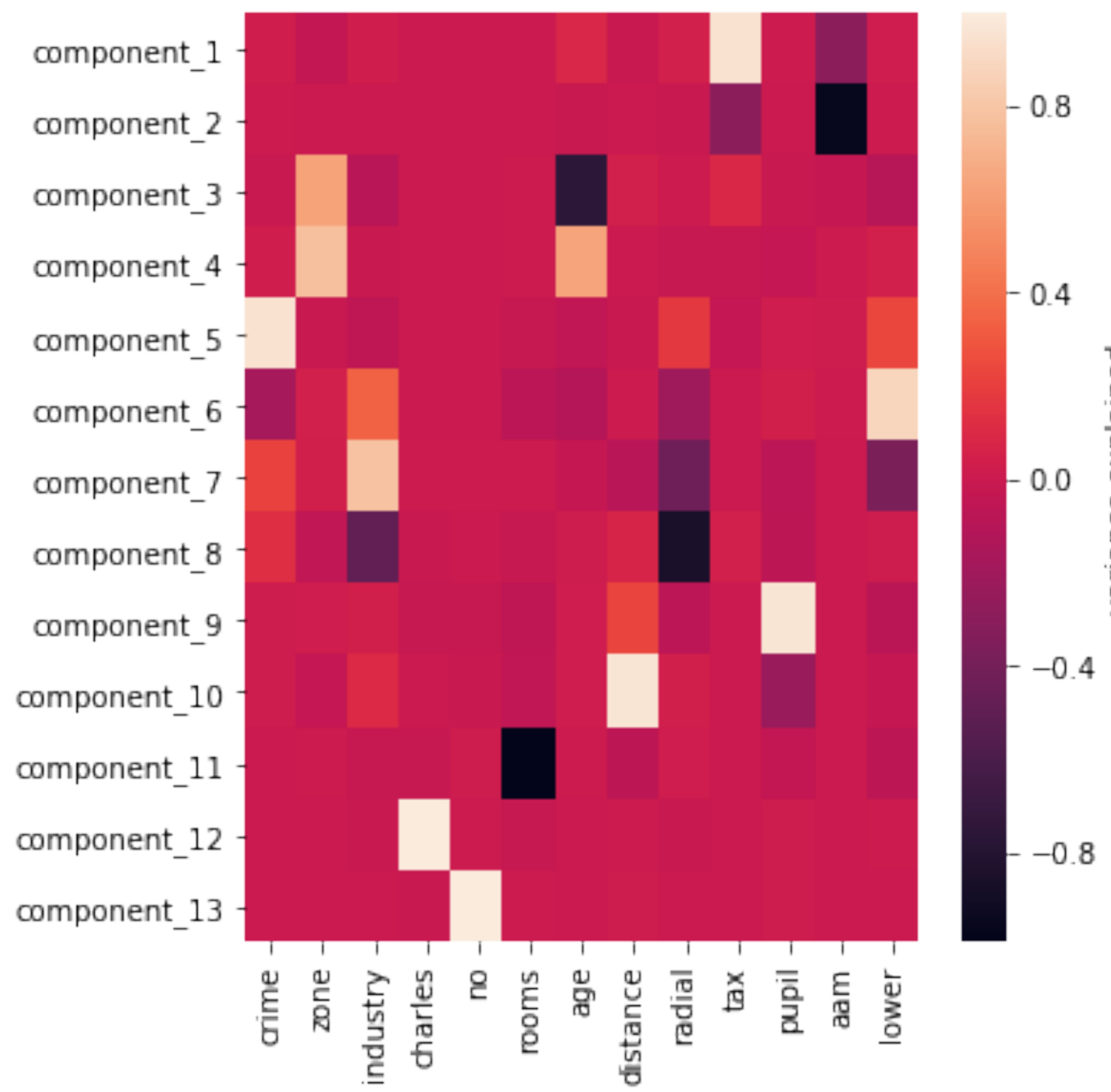
- We get a PCA decomposition of the original features into PCA components, into what is called a component matrix. This matrix tells us how to generate the new components (by blending the original features/columns) and is ordered from most-important to least important

- We get what are called the **eigenvectors** of each newly-generated component (column), which tells us how important that column is in terms of the amount of the "information" in the original dataset it explains. From the eigenvectors, we can compute the total amount of variance explained per-component as a fraction of the total amount of variance in the original dataset.

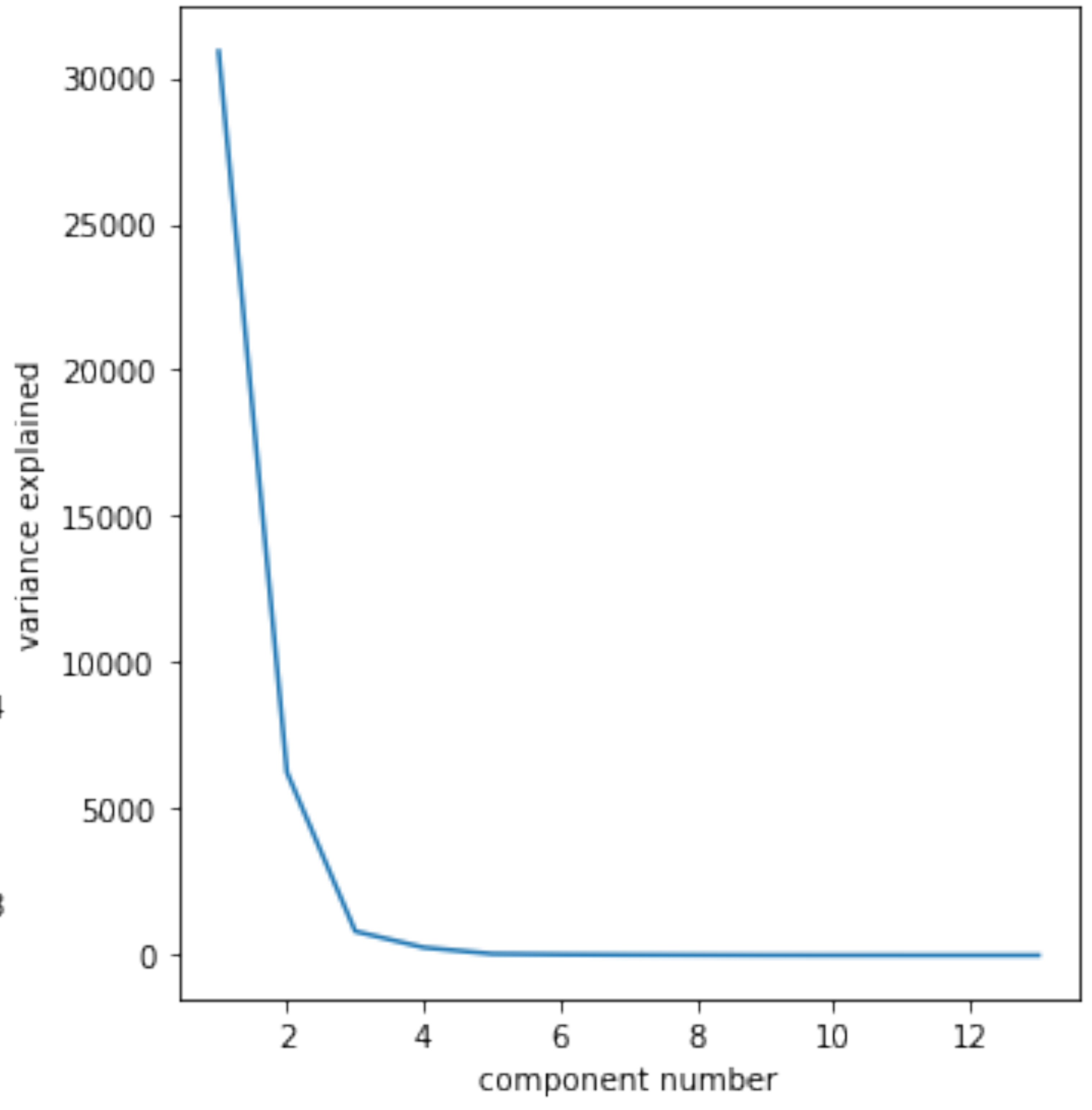
In general, PCA is a useful method for reducing your data to a smaller dimensional representation that you can look at, not necessarily that you can then successfully predict from.

METIS

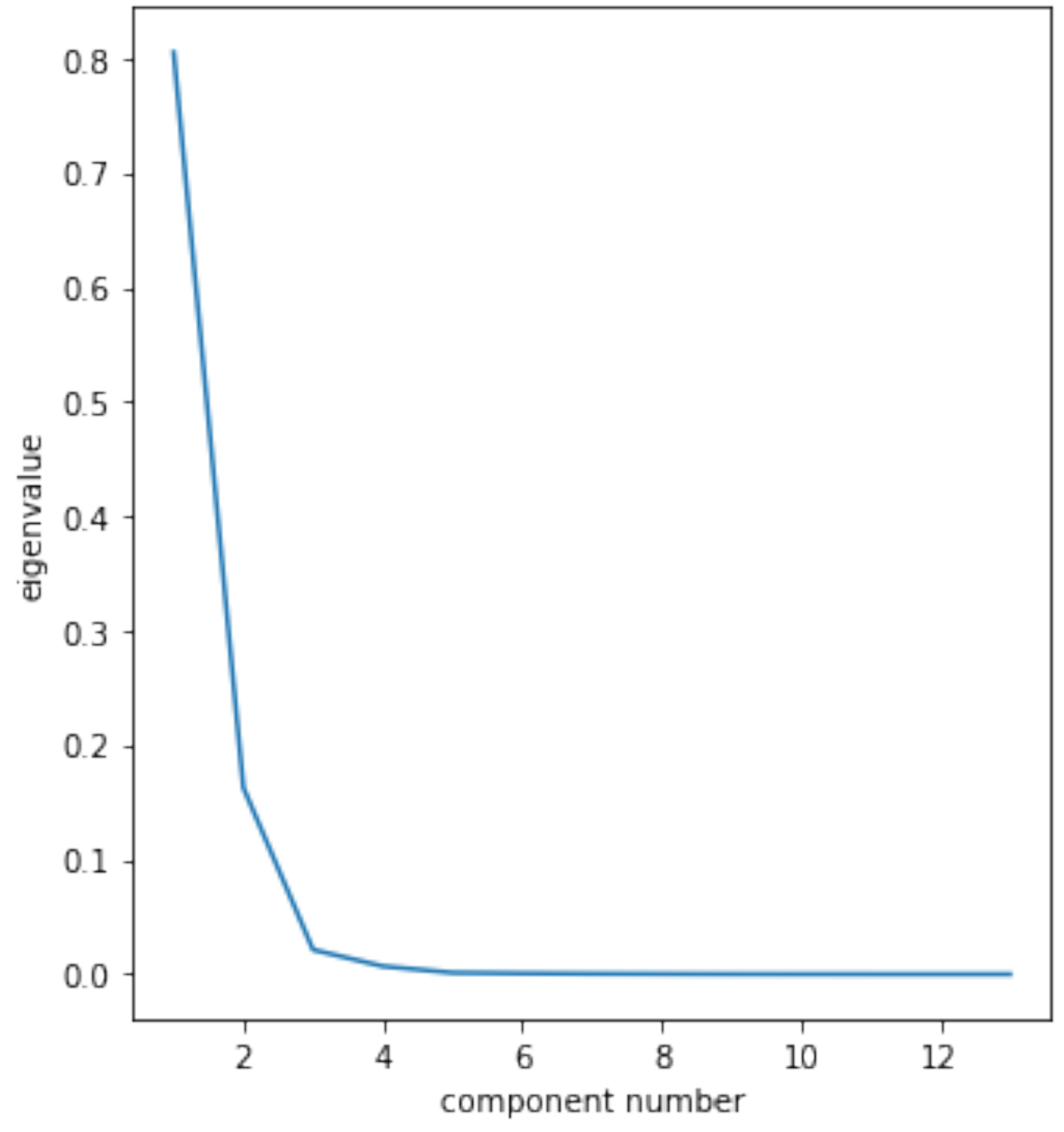
PCA



component_1
component_2
component_3
component_4
component_5
component_6
component_7
component_8
component_9
component_10
component_11
component_12
component_13
crime
zone
industry
charles
no
rooms
age
distance
radial
tax
pupil
aam
lower
0.8
0.4
0.0
−0.4
−0.8



variance explained
component number
30000
25000
20000
15000
10000
5000
0
2
4
6
8
10
12



eigenvalue
component number
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
2
4
6
8
10
12

METIS

PCA

In essence, PCA is fundamentally a linear method for projecting the original data onto a smaller-dimensional space (fewer columns) that keeps as much of the variance (distribution) of the original data as possible. What fraction of the original variance is explained by using just the first two PCA components?

Cumulative Sums of Variances of Components

component_1	0.805823
component_2	0.968875
component_3	0.990224
component_4	0.997181
component_5	0.998481
component_6	0.999208
component_7	0.999627
component_8	0.999875
component_9	0.999961
component_10	0.999992
component_11	0.999998
component_12	1
component_13	1

So it looks like just the first 2 components explain ~97% of the total variance in the dataset.

METIS

PCA

```
mean_squared_errors = np.abs(cross_val_score(lr,transformed_pca_x.iloc[:,:
2],y,cv=50,scoring='neg_mean_squared_error'))

root_mean_squared_errors = np.sqrt(mean_squared_errors)
```

Table 1-1

50-fold	**mean**	**RMSE:**	**7.07317986785**
50-fold	std	RMSE:	4.17602902835

METIS

Exercise

How many components do we need to keep until we achieve very similar performance?

Exercise

How many components do we need to keep until we achieve very similar performance?

Perform PCA on the `iris` dataset. Attempt to classify the 3 types of irises from each other using just the first 2 components of a PCA-transformed dataset, and using the full untransformed dataset. What is the classification difference?

METIS

Improving RMSE

Now, let's try something else with the Boston dataset in order to improve the overall predictive accuracy.

Let's generate additional polynomial features up to degree 2 (all pairwise interactions among our original columns), and see if we can't improve the overall RMSE (by adding non-linearities that might be predictive apart from the linear features we've had so far):

```
pf_2 = PolynomialFeatures(degree=2,interaction_only=True)
pf_2_data = pf_2.fit_transform(housing_data[housing_features])[:,1:]#ignore constant column
lr = LinearRegression()
absolute_errors = np.abs(cross_val_score(lr,pf_2_data,housing_data[housing_target],cv=10))
rmses = np.sqrt(absolute_errors)
print("10-fold mean RMSE for degree-2 case: ",np.mean(rmses))
```

```
10-fold mean RMSE for degree-3 case:  14.9645916144
```

METIS

Improving RMSE

```
10-fold mean RMSE for degree-3 case:  14.9645916144
```

Why did this get so much worse?

We have officially horribly overfit our data. Is there anything that can be done in this case?

YES. We can apply what is called REGULARIZATION!

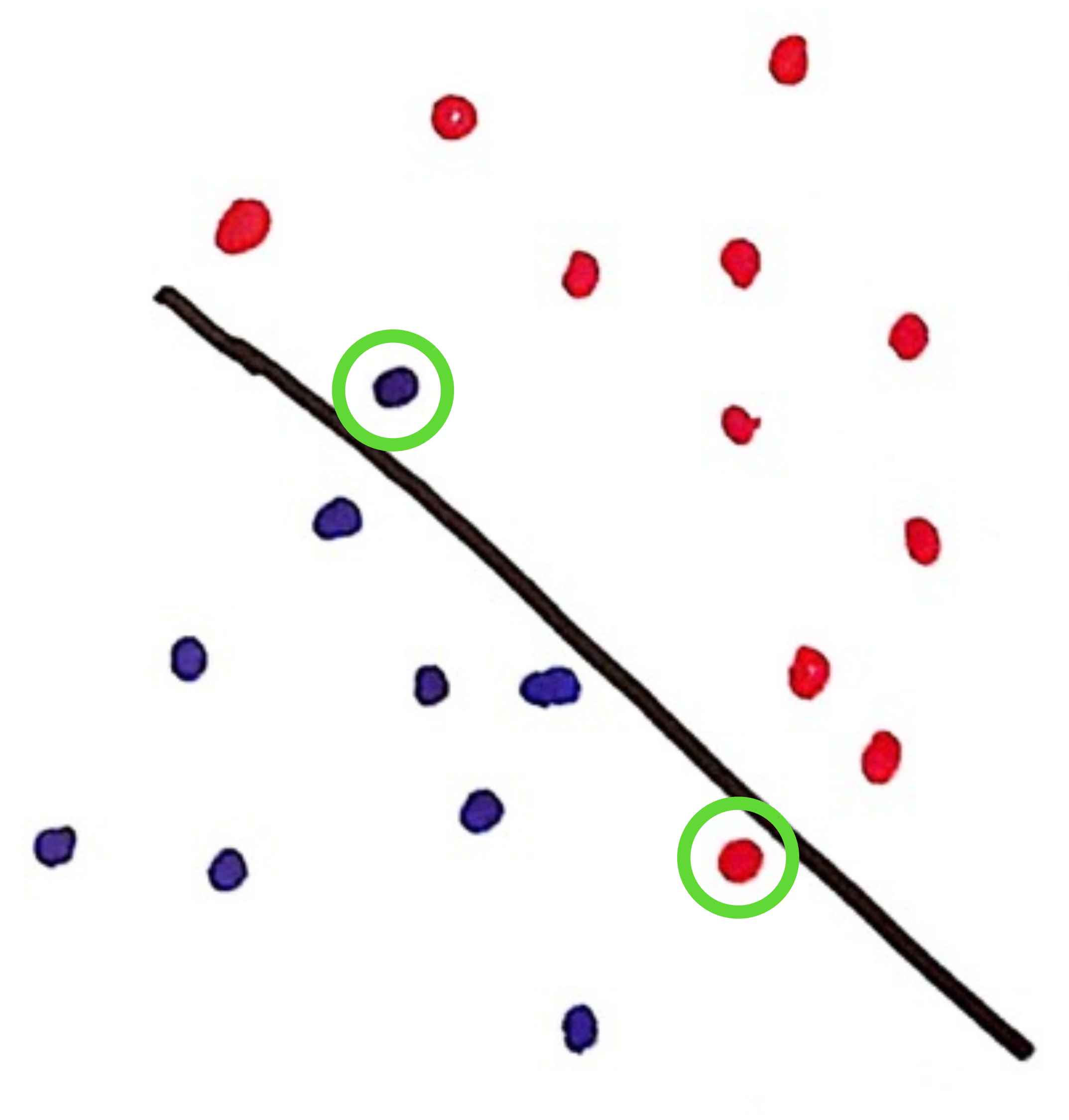
Regularization is a method for penalizing overly complicated models, while minimizing out-of-sample (test-set) error. We will talk about 2 kinds of regularization, which have different names, depending on whether they are being used for regression or classification:

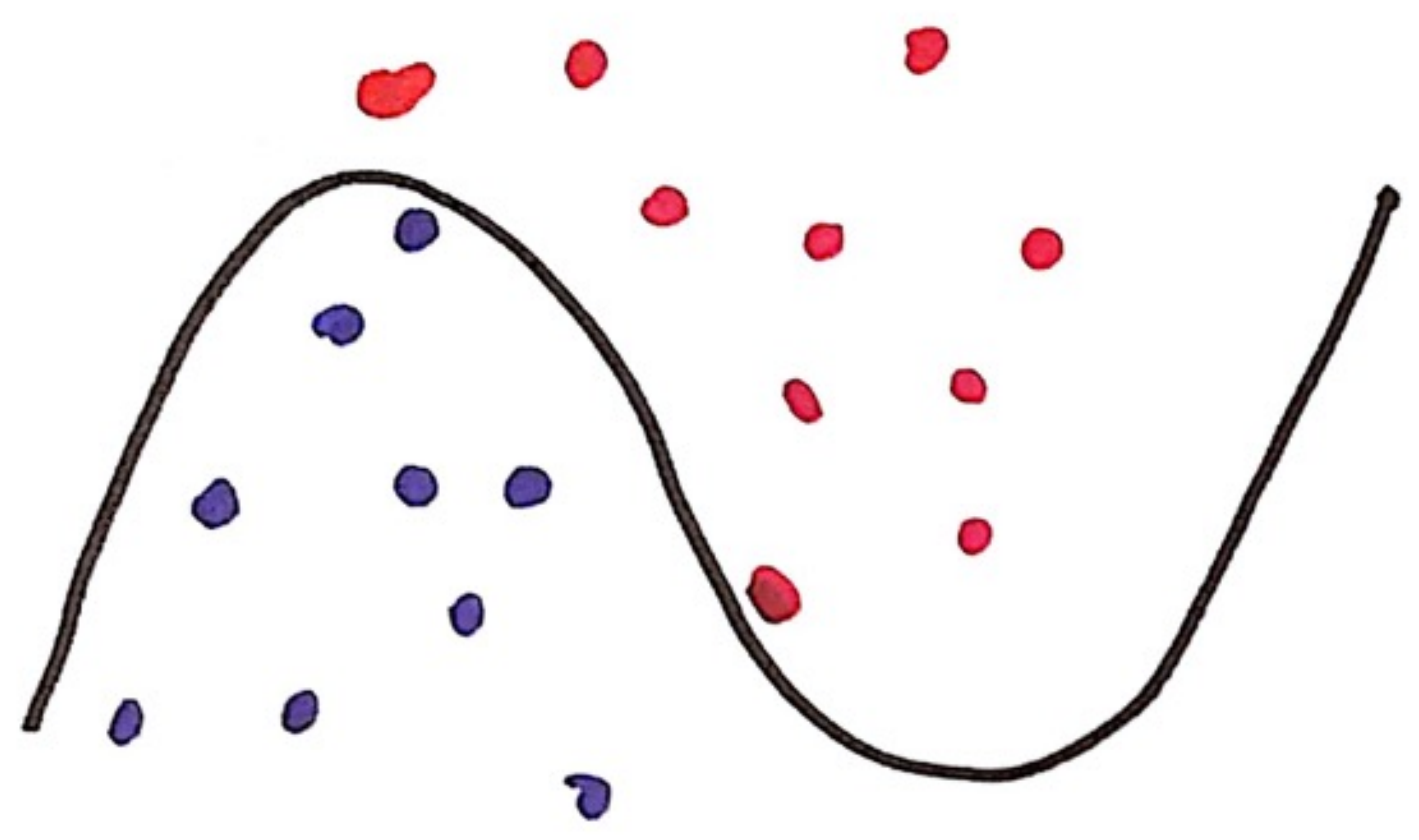
- Lasso/L1 Regularization
- Ridge/L2 Regularization

In essence, regularization is a method for adding additional constraints or penalties to a model, with the goal of preventing overfitting and improving generalization.

METIS

Regularization Summary





- Two Errors

- No errors

So is this model better?

Well, the one on the left makes a couple errors, while the model on the right has a lower error. So, if one were to choose a model based on the lowest error, the model on the right would win!

However, that model is also much more complicated!

If you take all the coefficients of the equations and add them to the error (which we'll get into shortly), you'll see that the model on the left actually has a smaller combined error.

METIS

Simple vs Complex

So, when is simple better over complex and vice versa?

We might be ok with some complexity if we're building a model that has little room for error (like a medical model or a model to send a rocket into space). Models that predict what movies you might like or who you might be friends with on Snapchat, however, have more room for experimenting and need to be simpler and faster to run on big data, so we're ok with some errors. Errors here are also not a matter of life or death.

Case 1

- requires low error
- ok if it's complex
- punishment on the complexity should be small

Case 2

- requires simplicity
- ok with errors
- punishment on the complexity should be large

METIS

Lasso Regression

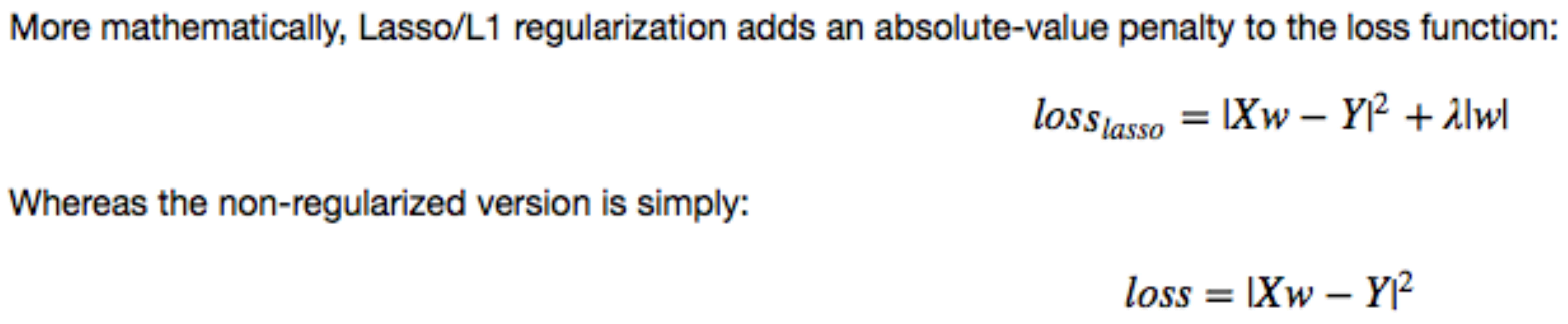
In traditional regression/classification, we try to minimize some kind of loss function (we try to make the loss be as close to 0 as possible). In regression, this is usually the **mean squared error** (or root mean squared error). However, with regularization, instead of just minimizing the loss based on the error of the model, we can also add a term that effectively "punishes" more complicated models.

What this means is that as you add more terms (features/columns) to your model, you suffer more added "loss due to complexity" in addition to "loss due to model performance."

So, how can regularization be used in linear regression to effectively make models simpler?

You reduce the coefficients for specific columns in your model until they go to zero (or very close to zero) for your model, which has the net effect of not using those features (or using them very sparingly)

Lasso Regression



More mathematically, Lasso/L1 regularization adds an absolute-value penalty to the loss function:

$$loss_{lasso} = |Xw - Y|^2 + \lambda|w|$$

Whereas the non-regularized version is simply:

$$loss = |Xw - Y|^2$$

So, L1 regularization adds a penalty (based on λ) to the regular loss function. Since each non-zero coefficient adds to the penalty, it forces weak features to have zero as coefficients. Thus L1 regularization produces sparse solutions, inherently performing feature selection.

For regression, Scikit-learn offers the `Lasso` and `LassoCV` functions, and for Logistic regression it offers the `l1` penalty parameter for classification. Let's use the `Lasso` with full regularization (when λ=0) on the degree-3 case:

METIS

Lasso Regression

```
lasso = Lasso(max_iter=2000)mean_squared_errors_poly3_lasso =
np.abs(cross_val_score(lasso,pf_3_data,housing_data[housing_target],cv=10))
rmses_lasso_poly3 = np.sqrt(mean_squared_errors_poly3_lasso)
```

```
10-fold mean RMSE for degree-3 case, strongest lasso
regularization:  0.871966016477
```

So, with full lasso regularization, we get a model that uses less than 40% of the original columns, and performs better than the model that only uses the original features or the combination of the original features and the pairwise interactions!

METIS

Exercise

- Try the lasso method but generating the 4th order polynomial features (use `PolynomialFeatures(degree=4,interactions_only=True)` and the same `Lasso()` and `cross_val_score` functions). Does our RMSE improve yet again?

- Try the lasso method for the poly3 case, but change the λ (called `alpha` in sklearn) parameter to `0.2`; What happens to the RMSE? What happens to the fraction of non-zero features in the model trained on all the data?

METIS

Ridge Regression

Lasso looks really excellent to use a regularization method. However, as I'd mentioned before, it is difficult to use across the board for all regression/classification problems, because it can create non-unique solutions, or solutions don't always converge.

So, what can be done? We can create a penalty term that is smooth and therefore differentiable everywhere. Specifically, we can use what is called **Ridge/L2 regularization.** Here, the loss is slightly different:

$$loss_{ridge} = |Xw - Y|^2 + \lambda |w|^2$$

Ridge Regression

The act of squaring the weight features makes the penalty differentiable (because the values now smoothly transition to 0). So, squaring the L1 penalty turns it into an L2 penalty Square the Lasso and get a Ridge.

Since the coefficients are squared in the penalty expression, it has a different effect from L1-norm, namely it forces the coefficient values to be spread out more equally. For correlated features, it means that they tend to get similar coefficients (whereas in L1 regularization, one of the terms will be forced to 0).

The effect of this is that models are much more stable. So while Ridge/L2 regularization does not perform feature selection the same way as L1 does, it is much more useful for feature *interpretation*; a predictive feature will get a non-zero coefficient, which is often not the case with L1.

Ridge Regression

```
ridge = Ridge()
ridge_cv_squared_errors =
np.abs(cross_val_score(ridge,pf_3_data,housing_data[housing_target],cv=10))
rmse_ridge_cv = map(np.sqrt,ridge_cv_squared_errors)
```

METIS

Exercise

- Try to use both lasso and ridge on the degree-2 polynomial version of the data, do either of them reduce the RMSE? What does our result tell us about the full degree-2 model?

METIS

Lasso vs. Ridge

Lasso:

- produces sparse models
- is useful for strong feature selection in order to improve model performance, or to minimize the number of explanatory variables.
- can produce non-unique solutions (when some features are very strongly correlated)
- can produce very different solutions depending on slight changes in features (because of non-uniqueness)

Ridge:

- produces stable models with smooth non-zero coefficients across features.
- is useful for data interpretation, understanding what features, even when correlated, may be used in combination to predict the response.
- may tell you something about how the data itself was generated.

You can combine both Lasso and Ridge models into a single penalized model (that uses a weighted combination of Lasso and Ridge regression). This is called the `ElasticNet`.

METIS

Conclusion

- PCA is more useful for explanation/data exploration than it is for classification/ regression when you have few (< 100) features.
- When generating polynomial features, generate the lowest-order polynomial interactions first, drop the constant term (the first column in the matrix), and try to use the full model for regression/classification.
- If you are trying for data explanation, use Ridge/L2 regularization
- If you are going for sparse models, use Lasso/L1, but realize that the solution may be non-unique. In general, Lasso doesn't always perform better than Ridge.
- `ElasticNet` requires significantly more tuning than either method, but can lead to the highest performing "linear" models. **As always, with great power comes great responsibility, so use all of these methods wisely.**