SVD Notebook

In this notebook, you will learn explore the topic of SVD in terms of the bias/variance tradeoff.

In [22]:
```
from keras.datasets import mnist
from sklearn.linear_model import LinearRegression, Lasso
from keras.utils import to_categorical
from sklearn.decomposition import PCA
import numpy as np
import matplotlib.pyplot as plt
import nbconvert
```

In [23]:
```
numpy.random.seed(0)
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

SVD Refresher

Go back and review your slides/notes on SVD. Also, feel free to ask around. In short, an SVD (Singular Value Decomposition) of a matrix is a low-rank approximation of that matrix. What does this mean? Suppose the original matrix was of rank R, and we are trying to reduce it to rank K. SVD is the result of taking this list of R unique vectors and approximating them as a linear combination of K unique vectors.

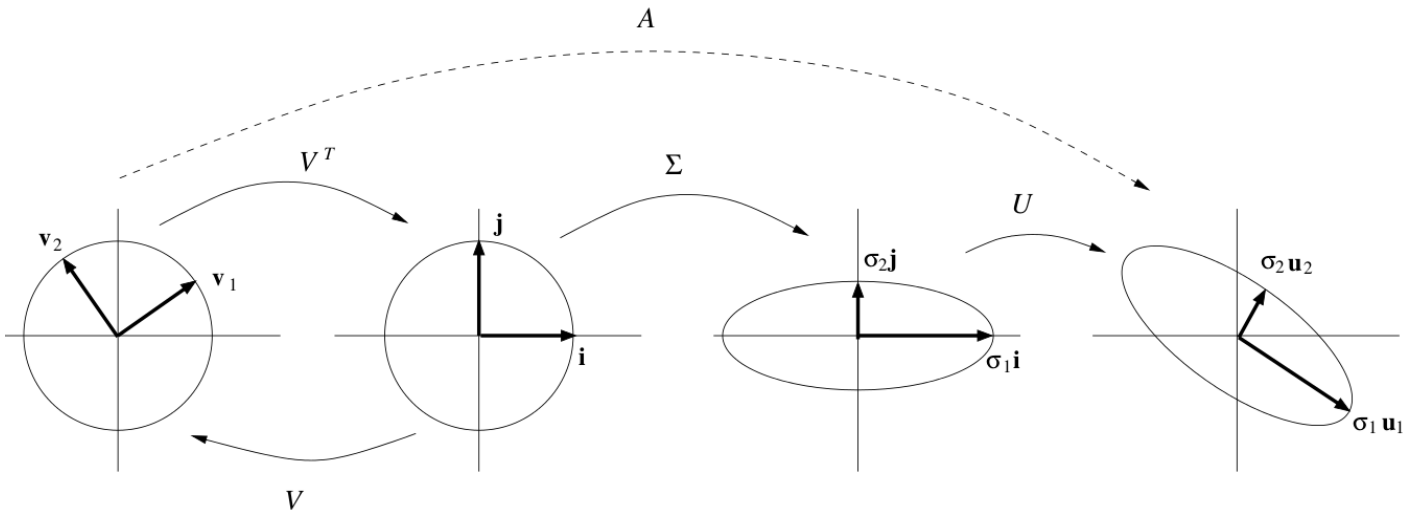
Another way to think about this is geometrically - in SVD:

$$A \approx U\Sigma V^T$$

Suppose we multiply a vector z with A.

$$Av \approx U\Sigma V^T z$$

We can think of this as a 3-step process: first, V^T hits z and *rotates* it into its eigenbasis (don't worry if you don't know what that is - all you should get out is that this first step is a rotation. Remember, U and V are orthonormal, so they don't change the magnitude of z, they only rotate). Then in step 2, Σ hits z and stretches it along the principal axes - remember, Σ is a diagonal matrix, so what's happening here is that each element of the newly rotated z is being scaled by a certain amount, the singular values. Finally in step 3, U hits z and rotates back into z's *original* basis. Here's a visual of this whole process:



MNIST Linear Regression

Let's see how SVD can help in terms of bias/variance. We'll be using the MNIST dataset. As a reminder, each image is a 28x28 black and white image labeled a number 0-9. Keep this information in the back of your mind, thinking especially about bias-variance, as we walk you through the effect of using SVD on this simple classification task.

We want to train a linear model. Reshape the X datasets appropriately for this task.

In [14]:
```
np.shape(x_train)
np.shape(x_test)
```

Out[14]: (10000, 28, 28)

Now create a linear regressor and fit it to the training set. Hint: make sure your y_train is properly featurized. What does it look like right now? How should we be encoding categorical data such as this?)

In [15]:
```
model = LinearRegression()
```

Print out the accuracies on both the train and test sets.

You may notice something curious - train accuracy is lower than test accuracy?! What is happening here is a severe case of underfitting - the model is not complex enough (indeed, for an image recognition task, a linear model is very simple). Thus, the scores will be very similar - however here the fact that test set performance is higher than training set performance is just an anomaly, due to variance. Proper cross validation would solve this issue.

Apply SVD

PCA is essentially the same as SVD. In PCA, the "principal components" are simply the singular vectors found in SVD. Apply PCA reduction to our data, keeping only the 100 most important dimensions.

In [19]:
```
from sklearn.decomposition import PCA
pca = PCA(n_components=3)

pca.fit(x_train)
```

```
---------------------------------------------------------------------
----
ValueError                                Traceback (most recent call l
ast)
<ipython-input-19-223ae42030c2> in <module>()
      2 pca = PCA(n_components=3)
      3
----> 4 pca.fit(x_train)

/anaconda3/lib/python3.6/site-packages/sklearn/decomposition/pca.py in
fit(self, X, y)
    327             Returns the instance itself.
    328         """
--> 329         self._fit(X)
    330         return self
    331

/anaconda3/lib/python3.6/site-packages/sklearn/decomposition/pca.py in
_fit(self, X)
    368
    369         X = check_array(X, dtype=[np.float64, np.float32], ensu
re_2d=True,
--> 370                         copy=self.copy)
    371
    372         # Handle n_components==None

/anaconda3/lib/python3.6/site-packages/sklearn/utils/validation.py in c
heck_array(array, accept_sparse, dtype, order, copy, force_all_finite,
 ensure_2d, allow_nd, ensure_min_samples, ensure_min_features, warn_on_
dtype, estimator)
    449         if not allow_nd and array.ndim >= 3:
    450             raise ValueError("Found array with dim %d. %s expec
ted <= 2."
--> 451                              % (array.ndim, estimator_name))
    452         if force_all_finite:
    453             _assert_all_finite(array)

ValueError: Found array with dim 3. Estimator expected <= 2.
```

Now with this new data, train a linear model.

And evaluate on train and test sets.

What did we notice? Well the train accuracy certainly decreased, but the test accuracy increased! (not by much - a different dataset would have proved this point better, but for ease of use we stick with a linear MNIST example). So let's recap: before we had 28x28=784 components, and after reducing our data to 100 components our test accuracy increased by train accuracy decreased. This is a very important lesson in bias-variance. In the section below, explain very clearly how this demonstrates the bias-variance tradeoff, and what influences increased bias and what influences increased variance.