**Linear Discriminant Analysis**

* **Supervised Learning + Dimensionality Reduction**

**Introduction**

In machine learning, the performance of a model only benefits from more features up until a certain point. The more features are fed into a model, the more the dimensionality of the data increases. As the dimensionality increases, overfitting becomes more likely.

There are multiple techniques that can be used to fight overfitting, but dimensionality reduction is one of the most effective techniques. Dimensionality reduction selects the most important components of the feature space, preserving them and dropping the other components.

**Why is Dimensionality Reduction Needed?**

There are a few reasons that dimensionality reduction is used in machine learning: to combat computational cost, to control overfitting, and to visualize and help interpret high dimensional data sets.

Often in machine learning, the more features that are present in the dataset the better a classifier can learn. However, more features also means a higher computational cost. Not only can high dimensionality lead to long training times, more features often lead to an algorithm overfitting as it tries to create a model that explains all the features in the data.

Because dimensionality reduction reduces the overall number of features, it can reduce the computational demands associated with training a model but also helps combat overfitting by keeping the features that will be fed to the model fairly simple.

Dimensionality reduction can be used in both supervised and [unsupervised learning contexts](https://scikit-learn.org/stable/modules/unsupervised_reduction.html). In the case of unsupervised learning, dimensionality reduction is often used to preprocess the data by carrying out feature selection or feature extraction.

The primary algorithms used to carry out dimensionality reduction for unsupervised learning are [Principal Component Analysis](https://stackabuse.com/dimensionality-reduction-in-python-with-scikit-learn/#principalcomponentanalysis) (PCA) and [Singular Value Decomposition](https://stackabuse.com/dimensionality-reduction-in-python-with-scikit-learn/#singularvaluedecomposition) (SVD).

In the case of supervised learning, dimensionality reduction can be used to simplify the features fed into the machine learning classifier. The most common methods used to carry out dimensionality reduction for supervised learning problems is [Linear Discriminant Analysis](https://stackabuse.com/dimensionality-reduction-in-python-with-scikit-learn/#lineardiscriminantanalysis) (LDA) and PCA, and it can be utilized to predict new cases.

### Linear Discriminant Analysis

[Linear Discriminant Analysis](https://stackabuse.com/implementing-lda-in-python-with-scikit-learn/) operates by projecting data from a multidimensional graph onto a linear graph. The easiest way to conceive of this is with a graph filled up with data points of two different classes. Assuming that there is no line that will neatly separate the data into two classes, the two dimensional graph can be reduced down into a 1D graph. This 1D graph can then be used to hopefully achieve the best possible separation of the data points.

When LDA is carried out there are two primary goals: minimizing the variance of the two classes and maximizing the distance between the means of the two data classes.

In order to achieve this, a new axis will be plotted in the 2D graph. This new axis should separate the two data points based on the previously mentioned criteria. Once the new axis has been created the data points within the 2D graph are redrawn along the new axis.

LDA carries out three different steps to move the original graph to the new axis. First, the separability between the classes has to be calculated, and this is based on the distance between the class means or the between-class variance. In the next step, the within class variance must be calculated, which is the distance between the mean and sample for the different classes. Finally, the lower dimensional space that maximizes the between class variance has to be constructed.

LDA works best when the means of the classes are far from each other. If the means of the distribution are shared it won't be possible for LDA to separate the classes with a new linear axis.

#### LDA Implementation Example

Finally, let's see how LDA can be used to carry out dimensionality reduction. Note that LDA can be used as a classification algorithm in addition to carrying out dimensionality reduction.

We'll be using the [Titanic](https://www.kaggle.com/c/titanic) dataset for the following example.

Let's start off by making all our necessary imports:

import pandas as pd

import numpy as np

from sklearn.metrics import accuracy\_score, f1\_score

from sklearn.preprocessing import LabelEncoder, StandardScaler

from sklearn.discriminant\_analysis import LinearDiscriminantAnalysis as LDA

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

We'll now load in our training data, which we'll divide into training and validation sets.

Though, we need to do a little data preprocessing first. Let's drop the Name, Cabin, and Ticket columns as they don't carry a lot of useful info. We also need to fill in any missing data, which we'll replace with median values in the case of the Age feature and an S in the case of the Embarked feature:

training\_data = pd.read\_csv("train.csv")

# Let's drop the cabin and ticket columns

training\_data.drop(labels=['Cabin', 'Ticket'], axis=1, inplace=True)

training\_data["Age"].fillna(training\_data["Age"].median(), inplace=True)

training\_data["Embarked"].fillna("S", inplace=True)

We also need to encode the non-numerical features. We'll encode both the Sex and Embarked columns. Let's drop the Name column as well, since it seems unlikely to be useful in classification:

encoder\_1 = LabelEncoder()

# Fit the encoder on the data

encoder\_1.fit(training\_data["Sex"])

# Transform and replace the training data

training\_sex\_encoded = encoder\_1.transform(training\_data["Sex"])

training\_data["Sex"] = training\_sex\_encoded

encoder\_2 = LabelEncoder()

encoder\_2.fit(training\_data["Embarked"])

training\_embarked\_encoded = encoder\_2.transform(training\_data["Embarked"])

training\_data["Embarked"] = training\_embarked\_encoded

# Assume the name is going to be useless and drop it

training\_data.drop("Name", axis=1, inplace=True)

We need to scale the values, but the Scaler tool takes arrays, so the values we want to reshape need to be turned into arrays first. After that, we can scale the data:

# Remember that the scaler takes arrays

ages\_train = np.array(training\_data["Age"]).reshape(-1, 1)

fares\_train = np.array(training\_data["Fare"]).reshape(-1, 1)

scaler = StandardScaler()

training\_data["Age"] = scaler.fit\_transform(ages\_train)

training\_data["Fare"] = scaler.fit\_transform(fares\_train)

# Now to select our training and testing data

features = training\_data.drop(labels=['PassengerId', 'Survived'], axis=1)

labels = training\_data['Survived']

We can now select the training features and labels and use train\_test\_split to make our training and validation data. It's easy to do classification with LDA, you handle it just like you would any other classifier in Scikit-Learn.

Just fit the function on the training data and have it predict on the validation/testing data. We can then print metrics for the predictions against the actual values:

X\_train, X\_val, y\_train, y\_val = train\_test\_split(features, labels, test\_size=0.2, random\_state=27)

model = LDA()

model.fit(X\_train, y\_train)

preds = model.predict(X\_val)

acc = accuracy\_score(y\_val, preds)

f1 = f1\_score(y\_val, preds)

print("Accuracy: {}".format(acc))

print("F1 Score: {}".format(f1))

Here's the print out:

Accuracy: 0.8100558659217877

F1 Score: 0.734375

When it comes to transforming the data and reducing dimensionality, let's run a Logistic Regression classifier on the data first so we can see what our performance is prior to dimensionality reduction:

logreg\_clf = LogisticRegression()

logreg\_clf.fit(X\_train, y\_train)

preds = logreg\_clf.predict(X\_val)

acc = accuracy\_score(y\_val, preds)

f1 = f1\_score(y\_val, preds)

print("Accuracy: {}".format(acc))

print("F1 Score: {}".format(f1))

Here's the results:

Accuracy: 0.8100558659217877

F1 Score: 0.734375

Now we will transform the data features by specifying a number of desired components for LDA and fitting the model on the features and labels. We then just transform the features and save it into a new variable. Let's print out the original and reduced number of features:

LDA\_transform = LDA(n\_components=1)

LDA\_transform.fit(features, labels)

features\_new = LDA\_transform.transform(features)

# Print the number of features

print('Original feature #:', features.shape[1])

print('Reduced feature #:', features\_new.shape[1])

# Print the ratio of explained variance

print(LDA\_transform.explained\_variance\_ratio\_)

Here's the print out for the above code:

Original feature #: 7

Reduced feature #: 1

[1.]

We now just have to do train/test split again with the new features and run the classifier again to see how performance changed:

X\_train, X\_val, y\_train, y\_val = train\_test\_split(features\_new, labels, test\_size=0.2, random\_state=27)

logreg\_clf = LogisticRegression()

logreg\_clf.fit(X\_train, y\_train)

preds = logreg\_clf.predict(X\_val)

acc = accuracy\_score(y\_val, preds)

f1 = f1\_score(y\_val, preds)

print("Accuracy: {}".format(acc))

print("F1 Score: {}".format(f1))

Accuracy: 0.8212290502793296

F1 Score: 0.7500000000000001