Feature Engineering

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

In [1]: %load_ext autoreload
   %autoreload 2
   %matplotlib inline

In [2]: import pandas as pd
   import numpy as np
   import matplotlib.pyplot as plt

Feature Engineering

Underfitting, Again

Let's load the CorkB dataset and do some simple preprocessing:
# Use pandas to read the CSV file
```python
df = pd.read_csv("dataset-corkB.csv")
```

# Remove anomalies from flarea
```python
df = df[(df['flarea'].isnull()) | ((df['flarea'] > 10) & (df['flarea'] < 1000))]
df.reset_index(drop=True, inplace=True)
```

# For simplicity, rather than impute missing values, we'll simply remove examples with missing values
```python
df.dropna(subset=['flarea', 'bdrms', 'floors', 'devment', 'price'], inplace=True)
df.reset_index(drop=True, inplace=True)
```

# Encode nominal-valued features

# - devment is easy because it is binary-valued
```python
df.replace({'devment': {'SecondHand': 0, 'New': 1}}, inplace=True)
```

# - for type we use one-hot encoding
```python
one_hot = pd.get_dummies(df['type'], 'type', '_')
df.drop('type', axis=1, inplace=True)
df = pd.concat([df, one_hot], axis=1)
```

# - similarly for ber
```python
one_hot = pd.get_dummies(df['ber'], 'ber', '_')
df.drop('ber', axis=1, inplace=True)
df = pd.concat([df, one_hot], axis=1)
```

# - and similarly for location
```python
one_hot = pd.get_dummies(df['location'], 'location', '_')
df.drop('location', axis=1, inplace=True)
df = pd.concat([df, one_hot], axis=1)
```

# We're going to use scikit-learn's linear regressor, so we won't bother with any scaling

Let's consider only the floor area, number of bedrooms and number of bathrooms (hence this is more like the CorkA dataset) and plot a learning curve for linear regression on this data:
from sklearn.linear_model import LinearRegression
from sklearn.cross_validation import ShuffleSplit
from sklearn.metrics import mean_squared_error

def plot_learning_curve(estimator, X, y, num_iterations = 10, test_size = 0.3):
    max_training_set_size = int((1 - test_size) * len(y))
    training_set_sizes = range(1, max_training_set_size)
    mse_incremental_train = np.zeros((max_training_set_size, num_iterations))
    mse_incremental_test = np.zeros((max_training_set_size, num_iterations))
    ss = ShuffleSplit(n = len(y), n_iter = num_iterations, test_size = test_size,
                      random_state = np.random)
    for i, (train_indexes, test_indexes) in zip(range(num_iterations), ss):
        X_train = X[train_indexes]
        y_train = y[train_indexes]
        X_test = X[test_indexes]
        y_test = y[test_indexes]
        for j in training_set_sizes:
            estimator.fit(X_train[:j], y_train[:j])
            y_predicted = estimator.predict(X_train[:j])
            mse_incremental_train[j][i] = mean_squared_error(y_train
                    n[:j], y_predicted)
            y_predicted = estimator.predict(X_test)
            mse_incremental_test[j][i] = mean_squared_error(y_test,
                    y_predicted)
        mse_train = []
        mse_test = []
        for j in training_set_sizes:
            mse_train.append(np.mean(mse_incremental_train[j]))
            mse_test.append(np.mean(mse_incremental_test[j]))
        fig = plt.figure()
        plt.xlabel('num. training examples')
        plt.ylabel('MSE')
        plt.ylim(0, 100000)
        plt.plot(training_set_sizes, mse_train, label = 'training error',
                 color = 'purple')
        plt.plot(training_set_sizes, mse_test, label = 'test error', color = 'orange')
        plt.plot(training_set_sizes, np.ones_like(training_set_sizes), linestyle = 'dotted', color = 'gray')
        plt.legend()
        plt.show()

X = df[['flarea', 'bdrms', 'bthrms']].values
y = df['price'].values
estimator = LinearRegression()
plot_learning_curve(estimator, X, y)
As ever, the results are quite sensitive to the exact training/test split. But, in most runs, we see above the characteristic signs of underfitting: high training and test error, and very little sign of the lines converging.

We know that the possible solutions include (i) moving to a more complex model, (ii) increasing the complexity of this model, and (iii) adding more features. The first is always tempting — it’s always more fun to try out a sexier learning algorithm. The second isn’t really applicable here: we can’t make linear regression more complex, only less complex. (How?) Let’s try the third solution. That’s easy, because the CorkB dataset has further features. (There were 5 further features but now that we have used some one-hot encoding those 5 have become quite a few more).

Here’s the learning curve when we use more of the features:

```python
In [5]: X = df[['flarea', 'bdrms', 'bthrms', 'location_Banduff', 'location_SundaysWell']].values
    y = df['price'].values
    estimator = LinearRegression()
    plot_learning_curve(estimator, X, y)
```
As usual, it depends on train/test splits but now on some runs the training and test errors are lower, and it looks like the curves converge a little — there’s probably less underfitting.

This makes the point that more features (if they are predictive) can help in cases of underfitting. But you don’t always have to collect new features. The idea of feature engineering is to derive new features — ones that we hope are more predictive — from the ones we already have.

### Feature Engineering: A Classic Example

Here I create an artificial dataset. There are two features, $x_1$ and $x_2$. Both are numeric-valued, and both take values in the range 0 to 10. The task is binary classification. After creating the dataset, I use a scatter plot to visualize it: $x_1$ is on the horizontal axis; $x_2$ is on the vertical axis; red crosses are negative examples; green plus symbols are positive examples.

```python
In [6]: from sklearn.datasets import make_circles
X, y = make_circles(100)
fig = plt.figure(figsize=(4,4))
plt.title("Dataset")
plt.xlabel("x1")
plt.xlim(-1.1, 1.1)
plt.ylabel("x2")
plt.ylim(-1.1, 1.1)
for outcome, marker, colour in zip([0,1], "x+", "rg"):
    plt.scatter(X[:, 0][y == outcome], X[:, 1][y == outcome], c = colour, marker = marker)
plt.show()
```

![Dataset Scatter Plot](image)
Imagine training Logistic Regression on this dataset. We’ve seen that it fits a linear decision boundary, so it really cannot do well. It will try to fit a linear decision boundary \( \beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0 \) to the data.

**Question:**
- What can we say about this data — using terminology from our previous module?

But let’s do some feature engineering. Let’s add two new features:
- \( x_3 \) will simply be \( x_1^2 \)
- \( x_4 \) will simply be \( x_2^2 \)

And we’ll visualize these two new features:

\[
X_{\text{new}} = \text{np.ones}((100, 4))
\]
\[
X_{\text{new}}[:, 0:2] = X
\]
\[
X_{\text{new}}[:, 2] = X[:, 0] \times X[:, 0]
\]
\[
X_{\text{new}}[:, 3] = X[:, 1] \times X[:, 1]
\]

\[
\beta_0 = -1, \beta_1 = 0, \beta_2 = 0, \beta_3 = 1, \beta_4 = 1, \text{i.e.}
-1 \times 1 + 0 \times x_1 + 0 \times x_2 + 1 \times x_3 + 1 \times x_4 = 0
\]

which simplifies to
\[
x_3 + x_4 = 1
\]

and, given how we created \( x_3 \) and \( x_4 \), this is the same as
\[
x_1^2 + x_2^2 = 1
\]

which you might recognise as the equation of a unit circle centred at zero.
Feature Engineering: Derived Features

The previous section exemplifies a general point. It can be worth adding additional features that are derived from the existing features. You could consider adding:

- higher-order polynomial features, such as the square or cube of an existing feature
- interaction terms, such as the product or ratios of pairs of features
- other transformations, such as the square root or log of a feature

An interesting observation is that using higher-order polynomial features can give some of the effects of a more complex model, without moving to that more complex model. For example, the target function is not linear, but instead of moving to a learner that can learn non-linear hypotheses, we give a linear model higher-order features. This doesn't wholly throw off the shackles of your linear model: it is still learning additive functions of the features that you give it.

Often better still, is to derive these new features using domain-specific knowledge (e.g. with the help of your domain expert). Here are a couple of examples:

- The first example is from this blog post (http://blog.bigml.com/2013/02/21/everything-you-wanted-to-know-about-machine-learning-but-were-too-afraid-to-ask-part-two/). A dataset has four features: the longitude and latitude of two cities. The dependent variable classifies examples according to whether it is reasonable to drive from one to the other, e.g.:

<table>
<thead>
<tr>
<th>City 1 Lat.</th>
<th>City 1 Lng.</th>
<th>City 2 Lat.</th>
<th>City 2 Lng.</th>
<th>Drivable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.24</td>
<td>46.71</td>
<td>121.33</td>
<td>47.34</td>
<td>Yes</td>
</tr>
<tr>
<td>123.24</td>
<td>56.91</td>
<td>121.33</td>
<td>55.23</td>
<td>Yes</td>
</tr>
<tr>
<td>123.24</td>
<td>46.71</td>
<td>121.33</td>
<td>55.34</td>
<td>No</td>
</tr>
<tr>
<td>123.24</td>
<td>46.71</td>
<td>130.99</td>
<td>47.34</td>
<td>No</td>
</tr>
</tbody>
</table>

  No simple learner (and certainly no learner that learns linear models) will do well at classifying examples like these. What we're expecting it to work out is the distance between the two cities. In the case of longitude and latitude (and distances between points on a sphere in general), this is given by the Haversine formula. Even a learner whose hypothesis space is rich enough to include this formula is unlikely ever to discover it from a set of examples like those above. But if you engage in feature engineering, it might occur to you to include distance as an extra feature:

<table>
<thead>
<tr>
<th>City 1 Lat.</th>
<th>City 1 Lng.</th>
<th>City 2 Lat.</th>
<th>City 2 Lng.</th>
<th>Distance (mi.)</th>
<th>Drivable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.24</td>
<td>46.71</td>
<td>121.33</td>
<td>47.34</td>
<td>14</td>
<td>Yes</td>
</tr>
<tr>
<td>123.24</td>
<td>56.91</td>
<td>121.33</td>
<td>55.23</td>
<td>28</td>
<td>Yes</td>
</tr>
<tr>
<td>123.24</td>
<td>46.71</td>
<td>121.33</td>
<td>55.34</td>
<td>705</td>
<td>No</td>
</tr>
<tr>
<td>123.24</td>
<td>46.71</td>
<td>130.99</td>
<td>47.34</td>
<td>2431</td>
<td>No</td>
</tr>
</tbody>
</table>

  And now it is easy to learn a hypothesis that does well — even for a learner that learns only linear models.

- What possibly useful feature might you derive from the alcohol dataset? To remind you, its original features are:
  - age, height, weight, duration, elapsed, sex, last meal
Case Study: Titanic Dataset

The Titanic dataset is available from Kaggle (http://www.kaggle.com). It records data about 891 Titanic passengers, and the job is to classify them according to whether they died (0) or survived (1). Here are its features and notes about simple data preparation:

- **PassengerId**
  - There was a unique id, which I have removed from my copy of the csv file.

- **Survived**
  - This is the class: 0 = died, 1 = survived.

- **Pclass**
  - This is the class of travel: first (1), second (2) or third (3).

- **Name**
  - This is the passenger's name. Ordinarily, we might delete this feature. Names are (nearly) unique and so unlikely to be predictive. However, I have not deleted it, and we will use it for some feature engineering later.

- **Sex**
  - This is recorded as "female" or "male", and so we must binarize it.

- **Age**
  - According to Kaggle, this is the age in years. But if a passenger's age is less than one, a float is used (e.g. 0.83). Some people's ages are already estimates and these too are shown in non-integer form (e.g. 28.5). There are 177 missing values. Rather than build a pipeline that imputes mean age, we'll simply delete examples where the age is missing — this is done just to make the code shorter and simpler for lecture purposes. In reality, we are deleting too many examples.

- **SibSp**
  - This is how many of the passenger's siblings (brothers, sisters, stepbrothers or stepsisters) are aboard, plus one if the passenger's spouse (husband or wife) is aboard.

- **Parch**
  - This is how many of the passenger's parents are aboard plus how many of the passenger's children (sons, daughters, stepsons or stepdaughters) are aboard.

- **Ticket**
  - This is the ticket number. Most are numeric but some are peculiar strings. Several people can have the same ticket number, e.g. family, friends or servants traveling together. We won't use this but I left it in because you can imagine that, suitably encoded, it could be predictive: probably people traveling together share the same outcome. (I haven't checked this.)

- **Fare**
  - Obviously, this is numeric. As noted in this on-line tutorial (http://triangleinequality.wordpress.com/2013/05/19/machine-learning-with-python-first-steps-munging/), some fares are 0. This might be understandable if the passenger was a baby. But that's not the case. Otherwise, the smallest fare is 4, so the zeros are either anomalies or missing values. The best thing might be to impute the mean for the passenger's Pclass. But, again, in the interests of shorter code for lecture purposes, we'll delete these examples.

- **Cabin**
  - This is the cabin number, e.g. C85. Only first class passengers have cabin numbers, so 687 are missing. We could binarize this feature. But perhaps we should delete it on the basis
that Pclass covers the same information. However, we will leave this feature untouched for now (leaving the missing values as empty strings), and we will use it for some feature engineering.

- **Embarked**
  - This is the port of embarkation: "C" = Cherbourg; "Q" = Queenstown; "S" = Southampton. You might think it is not predictive: why would your chances of survival depend on where you embarked? But, in fact, Southampton and Queenstown embarkees were more likely to die. There might be a correlation here with Pclass: hardly any Queenstown embarkees were first- or second-class. We will retain the feature. Since it is nominal-valued, we will one-hot encode it. Before that, we will delete the two examples that have missing values for this feature.

```
In [8]: # Use pandas to read the CSV file
df = pd.read_csv("dataset-titanic.csv")

print(df.shape)
df.describe()
```

```
Out[8]:

<table>
<thead>
<tr>
<th></th>
<th>Survived</th>
<th>Pclass</th>
<th>Age</th>
<th>SibSp</th>
<th>Parch</th>
<th>Fare</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>891.000000</td>
<td>891.000000</td>
<td>714.000000</td>
<td>891.000000</td>
<td>891.000000</td>
<td>891.000000</td>
</tr>
<tr>
<td>mean</td>
<td>0.383838</td>
<td>2.308642</td>
<td>29.699118</td>
<td>0.523008</td>
<td>0.381594</td>
<td>32.204208</td>
</tr>
<tr>
<td>std</td>
<td>0.486592</td>
<td>0.836071</td>
<td>14.526497</td>
<td>1.102743</td>
<td>0.806057</td>
<td>49.693429</td>
</tr>
<tr>
<td>min</td>
<td>0.000000</td>
<td>1.000000</td>
<td>0.420000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>25%</td>
<td>0.000000</td>
<td>2.000000</td>
<td>20.125000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>7.910400</td>
</tr>
<tr>
<td>50%</td>
<td>0.000000</td>
<td>3.000000</td>
<td>28.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>14.454200</td>
</tr>
<tr>
<td>75%</td>
<td>1.000000</td>
<td>3.000000</td>
<td>38.000000</td>
<td>1.000000</td>
<td>0.000000</td>
<td>31.000000</td>
</tr>
<tr>
<td>max</td>
<td>1.000000</td>
<td>3.000000</td>
<td>80.000000</td>
<td>8.000000</td>
<td>6.000000</td>
<td>512.329200</td>
</tr>
</tbody>
</table>

Out[9]:

```
<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Sex</th>
<th>Ticket</th>
<th>Cabin</th>
<th>Embarked</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>891</td>
<td>891</td>
<td>891</td>
<td>204</td>
<td>889</td>
</tr>
<tr>
<td>unique</td>
<td>891</td>
<td>2</td>
<td>681</td>
<td>147</td>
<td>3</td>
</tr>
<tr>
<td>top</td>
<td>Rugg, Miss. Emily</td>
<td>male</td>
<td>347082</td>
<td>B96 B98</td>
<td>S</td>
</tr>
<tr>
<td>freq</td>
<td>1</td>
<td>577</td>
<td>7</td>
<td>4</td>
<td>644</td>
</tr>
</tbody>
</table>
```
In [10]:
# Sex
df.replace({'Sex': {'female': 0, 'male': 1}}, inplace=True)

# Age, Fare, Embarked
# Delete examples whose Age, Fare or Embarked is NaN
df.dropna(subset=['Age', 'Fare', 'Embarked'], inplace=True)
df.reset_index(drop=True, inplace=True)

# Fare again
# One-hot encode it
one_hot = pd.get_dummies(df['Embarked'], 'Embarked', '_')
df.drop('Embarked', axis=1, inplace=True)
df = pd.concat([df, one_hot], axis=1)

Higher-Order Features

Now for some feature engineering. We'll create a few new features in a mindless way: just squaring and
cubing the numeric features (Age, SibSp, Parch and Fare):

In [11]:
# Age
age_values = df['Age'].values
age_squared = age_values * age_values
age_cubed = age_squared * age_values
df['AgeSquared'] = age_squared
df['AgeCubed'] = age_cubed

# SibSp
sibsp_values = df['SibSp'].values
sibsp_squared = sibsp_values * sibsp_values
sibsp_cubed = sibsp_squared * sibsp_values
df['SibSpSquared'] = sibsp_squared
df['SibSpCubed'] = sibsp_cubed

# Parch
parch_values = df['Parch'].values
parch_squared = parch_values * parch_values
parch_cubed = parch_squared * parch_values
df['ParchSquared'] = parch_squared
df['ParchCubed'] = parch_cubed

# Fare
fare_values = df['Fare'].values
fare_squared = fare_values * fare_values
fare_cubed = fare_squared * fare_values
df['FareSquared'] = fare_squared
df['FareCubed'] = fare_cubed

Domain-Inspired Feature Engineering
And now some feature engineering that is driven a little more by intuitions about the domain. All of these are based on this on-line tutorial (http://triangleinequality.wordpress.com/2013/09/08/basic-feature-engineering-with-the-titanic-data/).

- **Title**
  - We can extract a person's title from the Name feature. The titles to be found are: 'Mrs', 'Mr', 'Master', 'Miss', 'Major', 'Rev', 'Dr', 'Ms', 'Mlle', 'Col', 'Capt', 'Mme', 'Countess', 'Don', and 'Jonkheer'. There will be some missing values, which we will leave as they are. Then we'll one-hot encode this. We'll replace the Name feature by these new features.

```python
# Title
def find_one(substrs, superstr):
    for substr in substrs:
        if superstr.find(substr) != -1:
            return substr
    return ''

name_values = df['Name'].values
title_values = []
for name_value in name_values:
    title_values.append(find_one(['Mrs', 'Mr', 'Master', 'Miss', 'Major', 'Rev', 'Dr', 'Ms', 'Mlle', 'Col', 'Capt', 'Mme', 'Countess', 'Don', 'Jonkheer'], name_value))
one_hot = pd.get_dummies(title_values, 'Title', '_')
df.drop('Name', axis=1, inplace=True)
df = pd.concat([df, one_hot], axis=1)
```

- **Deck**
  - For first-class passengers, the Cabin feature records the cabin number(s), e.g. C85. The first character is the deck, and this may well be predictive. So we will extract it, like we did Title. Missing values will be turned into empty strings. Then we'll one-hot encode. We'll replace the Deck feature by these new features.

```python
# Deck
cabin_values = df['Cabin'].values
deck_values = []
for cabin_value in cabin_values:
    if str(cabin_value) == 'nan':
        deck_values.append('')
    else:
        deck_values.append(find_one(['A', 'B', 'C', 'D', 'E', 'F', 'T', 'G'], cabin_value))
one_hot = pd.get_dummies(deck_values, 'Deck', '_')
df.drop('Cabin', axis=1, inplace=True)
df = pd.concat([df, one_hot], axis=1)
```

**Interaction Features**
Let's throw in some examples of interaction features: sums, products and ratios.

- **FamSize**
  - We can create a feature from the sum of SibSp and Parch. It might be predictive: perhaps people traveling alone or in smaller groups did better by wasting less time trying to help other members of their party. Of course, for logistic regression, we don't really need to create new additive features, since its hypotheses are already based on (weighted) sums of the features. But this feature might be helpful to other classifiers.

- **AgePclass**
  - This interaction feature is not additive: it is the product of the Age and Pclass. The (somewhat tenuous) rationale is that young rich people (small age, small Pclass, giving small product) are more likely to survive than old poor people (high age, high Pclass, giving high product), so low values will correlate with survival.

- **FarePerPerson**
  - This is Fare divided by FamSize.

```python
In [14]:
# FamSize
sibsp_values = df['SibSp'].values
parch_values = df['Parch'].values
famsise = sibsp_values + parch_values + 1
df['FamSize'] = famsize

# AgePclass
age_values = df['Age'].values
pclass_values = df['Pclass'].values
agepclass = age_values * pclass_values
df['AgePclass'] = agepclass

# FarePerPerson
fare_values = df['Fare'].values
fareperperson = fare_values / famsize
df['FarePerPerson'] = fareperperson
```

**Trying Out the New Features**

So now let's do some error estimation. We'll use logistic regression (but with $l_1$-norm regularization) and 10-fold cross validation. We'll try it (a) with just the original features (after preprocessing them and excluding the ones we left in but didn't intend to use such as Name, Ticket and Cabin), (b) with the originals plus all the higher-order polynomial terms, (c) with the originals plus the domain-specific ones we engineered but without the higher-order polynomial terms, and (d) with all of them.
from sklearn.linear_model import LogisticRegression
from sklearn.cross_validation import cross_val_score

def run(X, y):
    estimator = LogisticRegression(penalty = 'l1', C = 1)
    accuracies_test = cross_val_score(estimator, X, y, scoring = 'accuracy', cv = 10)
    return np.mean(accuracies_test)

X = df[['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked_C', 'Embarked_Q', 'Embarked_S']].values
y = df['Survived'].values
print "Original: ", run(X, y)

X = df[['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked_C', 'Embarked_Q', 'Embarked_S', 'AgeSquared', 'AgeCubed', 'SibSpSquared', 'SibSpCubed', 'ParchSquared', 'ParchCubed', 'FareSquared', 'FareCubed']].values
print "Original + Higher-order Polynomials: ", run(X, y)

X = df[['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked_C', 'Embarked_Q', 'Embarked_S', 'Title_Mrs', 'Title_Mr', 'Title_Master', 'Title_Miss', 'Title_Major', 'Title_Rev', 'Title_Doctor', 'Title_Ms', 'Title_Mlle', 'Title_Col', 'Title_Capt', 'Title_Mme', 'Title_Countess', 'Title_Don', 'Title_Jonkheer', 'Deck', 'Deck_A', 'Deck_B', 'Deck_C', 'Deck_D', 'Deck_E', 'Deck_F', 'Deck_T', 'Deck_G', 'FamSize', 'AgePclass', 'FarePerPerson']].values
print "Original + Domain-Specific: ", run(X, y)

X = df[['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked_C', 'Embarked_Q', 'Embarked_S', 'AgeSquared', 'AgeCubed', 'SibSpSquared', 'SibSpCubed', 'ParchSquared', 'ParchCubed', 'FareSquared', 'FareCubed', 'Title_Mrs', 'Title_Mr', 'Title_Master', 'Title_Miss', 'Title_Major', 'Title_Rev', 'Title_Doctor', 'Title_Ms', 'Title_Mlle', 'Title_Col', 'Title_Capt', 'Title_Mme', 'Title_Countess', 'Title_Don', 'Title_Jonkheer', 'Deck', 'Deck_A', 'Deck_B', 'Deck_C', 'Deck_D', 'Deck_E', 'Deck_F', 'Deck_T', 'Deck_G', 'FamSize', 'AgePclass', 'FarePerPerson']].values
print "All: ", run(X, y)

Original:  0.709776995305
Original + Higher-order Polynomials:  0.71957690588
Original + Domain-Specific:  0.725488486474
All:  0.719973731277