Reference

This is a DataCamp course.

Introduction

- This course involves a lot of natural language processing. The sklearn.feature_extraction.text is mainly used for many functions. such as:
  from sklearn.feature_extraction.text import HashingVectorizer
  from sklearn.feature_extraction.text import CountVectorizer
  Comparing to another course for natural language processing where nltk is mainly employed, and check their pros and cons of two packages.
- Explore a problem related to school district budgeting. By building a model to automatically classify items in a school's budget, it makes it easier and faster for schools to compare their spending with other schools.
- Classify line-items in a school budget based on what that money is being used for.
- The data set can be classified in many ways according to certain labels, e.g. Function: Career & Academic Counseling, or Position_Type: Librarian.
- Develop a model that predicts the probability for each possible label by relying on some correctly labeled examples.
- A **multi-class-multi-label** classification problem, because there are 9 broad categories that each take on many possible sub-label instances.
- First build a baseline model that is a simple, first-pass approach. Do some natural language processing to prepare the budgets for modeling.
- Check how competition winner was able to combine a number of expert techniques to build the most accurate model.
- Make schools more efficient by improving their budgeting decisions. Saves hundreds of hours each year that humans spent labeling line items. Spend more time on the decisions that really matter.

Exploring the raw data

Loading and summarizing the data

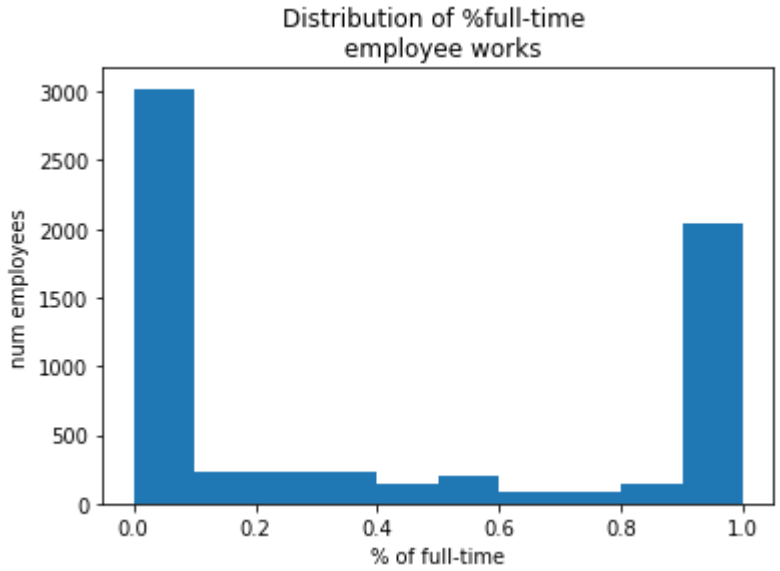
- Some of the column names correspond to features - descriptions of the budget items - such as the Job_Title_Description column. The values in this column tell us if a budget item is for a teacher, custodian, or other employee.

- Some columns correspond to the budget item labels you will be trying to predict with your model. For example, the Object_Type column describes whether the budget item is related classroom supplies, salary, travel expenses, etc.
- Two numeric columns, called FTE and Total. FTE: Stands for "full-time equivalent". If the budget item is associated to an employee, this number tells us the percentage of full-time that the employee works. A value of 1 means the associated employee works for the schooll full-time. A value close to 0 means the item is associated to a part-time or contracted employee.
- Total: Stands for the total cost of the expenditure. This number tells us how much the budget item cost.

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

filePath = "C:/Users/ljyan/Desktop/courseNotes/dataScience/machineLearning/data/"
filename = "TrainingData.csv"
file = filePath+filename
df = pd.read_csv(file, index_col = 0)


print(len(df))
df = df.sample(frac=0.05, random_state=9) ## Sampling a small portion for testing.
print(len(df))
df1 = df['FTE'].dropna()
print(len(df1))

#Use a small sample for fast calculation.
```

```
400277
20014
6332
```

In [2]:
```
plt.hist(df1, bins = [0,0.1,0.2,0.4,0.5,0.6,0.7,0.8,0.9,1.0])
#Need specify bins parameter. Otherwise the default will give a single bar.

plt.title('Distribution of %full-time \n employee works')
plt.xlabel('% of full-time')
plt.ylabel('num employees')


plt.show()
```



Exploring datatypes in pandas

How many columns with dtype object are in the data? **dtype vs object**

In [3]:
```
print(df.dtypes.value_counts())
print('-------------------------')
print(df.info())
```

```
object     23
float64     2
dtype: int64
-------------------------
<class 'pandas.core.frame.DataFrame'>
Int64Index: 20014 entries, 437872 to 79429
Data columns (total 25 columns):
Function                  20014 non-null object
Use                       20014 non-null object
Sharing                   20014 non-null object
Reporting                 20014 non-null object
Student_Type              20014 non-null object
Position_Type             20014 non-null object
Object_Type               20014 non-null object
Pre_K                     20014 non-null object
Operating_Status          20014 non-null object
Object_Description        18797 non-null object
Text_2                    4389 non-null object
SubFund_Description       15331 non-null object
Job_Title_Description     14762 non-null object
Text_3                    5405 non-null object
Text_4                    2737 non-null object
Sub_Object_Description    4695 non-null object
Location_Description      8183 non-null object
FTE                       6332 non-null float64
Function_Description      17184 non-null object
Facility_or_Department    2799 non-null object
Position_Extra            13303 non-null object
Total                     19785 non-null float64
Program_Description       15225 non-null object
Fund_Description          10200 non-null object
Text_1                    14773 non-null object
dtypes: float64(2), object(23)
memory usage: 4.0+ MB
None
```

Encode the labels as categorical variables

- Ultimate goal is to predict the probability that a certain label is attached to a budget line item.
- Many columns in the data are the inefficient object type. Does this include the labels we're trying to predict?
- There are 9 columns of labels in the dataset. Each of these columns is a category that has many possible values it can take.
- Every label is encoded as an object datatype. Because category datatypes are much more efficient, we convert the labels to category types using the .astype() method.
- Note: .astype() only works on a pandas Series. Since we are working with a pandas DataFrame, we need to use the .apply() method and provide a lambda function called categorize_label that applies .astype() to each column, x.

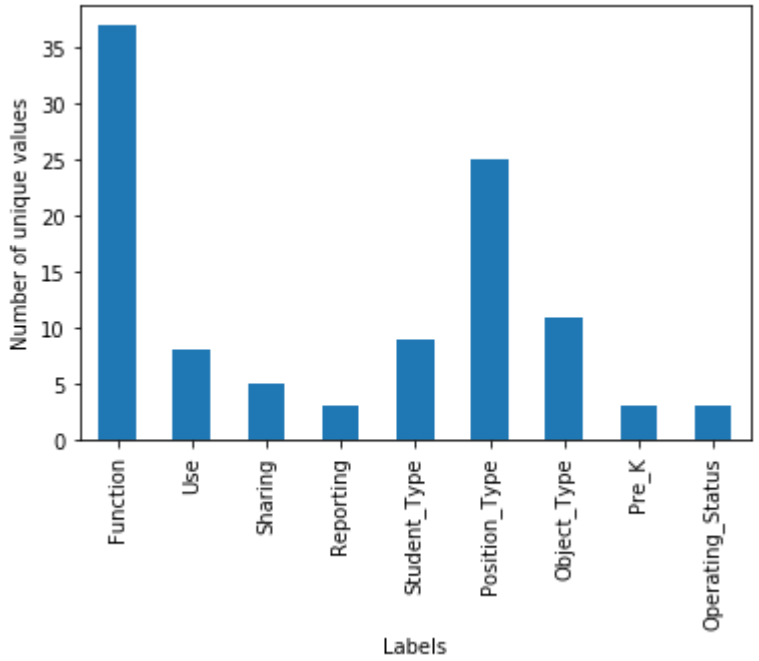
In [4]:
```
categorize_label = lambda x: x.astype('category')

# Convert df[LABELS] to a categorical type
LABELS = ['Function','Use','Sharing','Reporting','Student_Type','Position_Type',
          'Object_Type', 'Pre_K', 'Operating_Status']

df[LABELS] = df[LABELS].apply(categorize_label,axis = 0)

print(df[LABELS].dtypes)
```

```
Function            category
Use                 category
Sharing             category
Reporting           category
Student_Type        category
Position_Type       category
Object_Type         category
Pre_K               category
Operating_Status    category
dtype: object
```

Counting unique labels

Count and plot the number of unique values for each category of label. There are over 100 unique labels.

In [5]:
```
import matplotlib.pyplot as plt

num_unique_labels = df[LABELS].apply(pd.Series.nunique)
#count 9 columns simultaneously.

num_unique_labels.plot(kind = 'bar')

plt.xlabel('Labels')
plt.ylabel('Number of unique values')
plt.show()
```



Penalizing highly confident wrong answers

- log loss provides a steep penalty for predictions that are both wrong and confident, i.e., a high probability is assigned to the incorrect class. See definition of cross-entropy in statistics notes.

Computing log loss with NumPy

- Show how log loss metric handles the trade-off between accuracy and confidence.
- 5 one-dimensional numeric arrays simulating different types of predictions have been pre-loaded: actual_labels, correct_confident, correct_not_confident, wrong_not_confident, and wrong_confident.

In [6]:
```
import numpy as np
def compute_log_loss(predicted, actual, eps = 1e-14):
    """ Compute the logarithmic loss between predicted and actual
        when these are 1D arrays
    """
    predicted = np.clip(predicted,eps,1-eps)
    loss = -1 * np.mean(actual * np.log(predicted) \
            +(1-actual) * np.log(1-predicted))
    return loss
```

In [7]:
```
actual_labels = np.array([ 1.,  1.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.])
correct_confident = np.array([ 0.95,  0.95,  0.95,  0.95,  0.95,  0.05,  0.05,  0.05,  0.05,  0.05])
correct_not_confident = np.array([ 0.65,  0.65,  0.65,  0.65,  0.65,  0.35,  0.35,  0.35,  0.35,  0.35])
wrong_not_confident = np.array([ 0.35,  0.35,  0.35,  0.35,  0.35,  0.65,  0.65,  0.65,  0.65,  0.65])
wrong_confident= np.array([ 0.05,  0.05,  0.05,  0.05,  0.05,  0.95,  0.95,  0.95,  0.95,  0.95])

# Compute and print log loss for 1st case
correct_confident = compute_log_loss(correct_confident, actual_labels)
print("Log loss, correct and confident: {}".format(correct_confident))

# Compute log loss for 2nd case
correct_not_confident = compute_log_loss(correct_not_confident, actual_labels)
print("Log loss, correct and not confident: {}".format(correct_not_confident))

# Compute and print log loss for 3rd case
wrong_not_confident = compute_log_loss(wrong_not_confident, actual_labels)
print("Log loss, wrong and not confident: {}".format(wrong_not_confident))

# Compute and print log loss for 4th case
wrong_confident = compute_log_loss(wrong_confident, actual_labels)
print("Log loss, wrong and confident: {}".format(wrong_confident))

# Compute and print log loss for actual labels
actual_labels = compute_log_loss(actual_labels, actual_labels)
print("Log loss, actual labels: {}".format(actual_labels))
```

```
Log loss, correct and confident: 0.05129329438755058
Log loss, correct and not confident: 0.4307829160924542
Log loss, wrong and not confident: 1.049822124498678
Log loss, wrong and confident: 2.9957322735539904
Log loss, actual labels: 9.99200722162646e-15
```

Creating a simple first model

Setting up a train-test split in scikit-learn

- Split the data into a training set and a test set. Some labels don't occur very often, but we want to make sure that they appear in both the training and the test sets. We provide a function that will make sure at least min_count examples of each label appear in each split: multilabel_train_test_split.

- Start with a simple model that uses just the numeric columns of the DataFrame when calling multilabel_train_test_split.

In [8]:

```
from warnings import warn

import numpy as np
import pandas as pd

def multilabel_sample(y, size=1000, min_count=1, seed=None): #min_count = 5 is temporarily changed. Same for belo
    """ Takes a matrix of binary labels `y` and returns
        the indices for a sample of size `size` if
        `size` > 1 or `size` * len(y) if size =< 1.
        The sample is guaranteed to have > `min_count` of
        each label.
    """
    try:
        if (np.unique(y).astype(int) != np.array([0, 1])).all():
            raise ValueError()
    except (TypeError, ValueError):
        raise ValueError('multilabel_sample only works with binary indicator matrices')

    if (y.sum(axis=0) < min_count).any():
        raise ValueError('Some classes do not have enough examples. Change min_count if necessary.')

    if size <= 1:
        size = np.floor(y.shape[0] * size)

    if y.shape[1] * min_count > size:
        msg = "Size less than number of columns * min_count, returning {} items instead of {}."
        warn(msg.format(y.shape[1] * min_count, size))
        size = y.shape[1] * min_count

    rng = np.random.RandomState(seed if seed is not None else np.random.randint(1))

    if isinstance(y, pd.DataFrame):
        choices = y.index
        y = y.values
    else:
        choices = np.arange(y.shape[0])

    sample_idxs = np.array([], dtype=choices.dtype)

    # first, guarantee > min_count of each label
    for j in range(y.shape[1]):
        label_choices = choices[y[:, j] == 1]
```

```
        label_idxs_sampled = rng.choice(label_choices, size=min_count, replace=False)
        sample_idxs = np.concatenate([label_idxs_sampled, sample_idxs])

    sample_idxs = np.unique(sample_idxs)

    # now that we have at least min_count of each, we can just random sample
    sample_count = int(size - sample_idxs.shape[0])

    # get sample_count indices from remaining choices
    remaining_choices = np.setdiff1d(choices, sample_idxs)
    remaining_sampled = rng.choice(remaining_choices,
                                   size=sample_count,
                                   replace=False)

    return np.concatenate([sample_idxs, remaining_sampled])


def multilabel_sample_dataframe(df, labels, size, min_count=1, seed=None):
    """ Takes a dataframe `df` and returns a sample of size `size` where all
        classes in the binary matrix `labels` are represented at
        least `min_count` times.
    """
    idxs = multilabel_sample(labels, size=size, min_count=min_count, seed=seed)
    return df.loc[idxs]


def multilabel_train_test_split(X, Y, size, min_count=1, seed=None):
    """ Takes a features matrix `X` and a label matrix `Y` and
        returns (X_train, X_test, Y_train, Y_test) where all
        classes in Y are represented at least `min_count` times.
    """
    index = Y.index if isinstance(Y, pd.DataFrame) else np.arange(Y.shape[0])

    test_set_idxs = multilabel_sample(Y, size=size, min_count=min_count, seed=seed)
    train_set_idxs = np.setdiff1d(index, test_set_idxs)

    test_set_mask = index.isin(test_set_idxs)
    train_set_mask = ~test_set_mask

    return (X[train_set_mask], X[test_set_mask], Y[train_set_mask], Y[test_set_mask])
```

In [9]:
```
NUMERIC_COLUMNS = ['FTE','Total']
```

pd.get_dummies() does not drop one column. Is it because we are not considering them as featuers?

In [10]:
```
numeric_data_only = df[NUMERIC_COLUMNS].fillna(-1000)

print(numeric_data_only.head())
# print(LABELS)

# Get labels and convert to dummy variables: label_dummies
label_dummies = pd.get_dummies(df[LABELS])
# print(label_dummies.head())
print(numeric_data_only.shape, label_dummies.shape)
#We get dummies only for some labels but not all of them

# Note the shape of the first and second parameter in the function below. It is not like X,y before.
# The second parameter is 'fatter' than the first one.
X_train, X_test, y_train, y_test = multilabel_train_test_split(numeric_data_only,
                                                               label_dummies,
                                                               size=0.2,
                                                               seed=123)
#Be careful, it is not the standard split function. Normally we can use stratify option to handle
# the similar issue. However, the function here might be more specific. Check it out later.


print("X_train info:")
print(X_train.info())
print("\nX_test info:")
print(X_test.info())
print("\ny_train info:")
print(y_train.info())
print("\ny_test info:")
print(y_test.info())
```

```
           FTE         Total
437872 -1000.0   3823.392960
359734 -1000.0     99.954652
37681  -1000.0    340.650400
39474  -1000.0      0.810000
60718      1.0  81687.070000
(20014, 2) (20014, 104)
X_train info:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16012 entries, 437872 to 79429
Data columns (total 2 columns):
FTE      16012 non-null float64
```

```
Total    16012 non-null float64
dtypes: float64(2)
memory usage: 375.3 KB
None

X_test info:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4002 entries, 103771 to 270495
Data columns (total 2 columns):
FTE      4002 non-null float64
Total    4002 non-null float64
dtypes: float64(2)
memory usage: 93.8 KB
None

y_train info:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16012 entries, 437872 to 79429
Columns: 104 entries, Function_Aides Compensation to Operating_Status_PreK-12 Operating
dtypes: uint8(104)
memory usage: 1.7 MB
None

y_test info:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4002 entries, 103771 to 270495
Columns: 104 entries, Function_Aides Compensation to Operating_Status_PreK-12 Operating
dtypes: uint8(104)
memory usage: 437.7 KB
None
```

Training a model

- Use logistic regression and one versus rest classifiers to fit a multi-class logistic regression model to the NUMERIC_COLUMNS of your feature data.

In [11]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier

clf = OneVsRestClassifier(LogisticRegression(solver = 'lbfgs'))

clf.fit(X_train,y_train)

print("Accuracy: {}".format(clf.score(X_test,y_test)))
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\multiclass.py:76: UserWarning: Label not 9 is present in all
training examples.
  str(classes[c]))

Accuracy: 0.0
```

Use your model to predict values on holdout data

There is no holdout data here. However, it is possible to extract one from the complete dataset before training.

In [ ]:
```
clf = OneVsRestClassifier(LogisticRegression(solver='lbfgs'))
clf.fit(X_train, y_train)

holdout = pd.read_csv('HoldoutData.csv', index_col = 0)
predictions = clf.predict_proba(holdout[NUMERIC_COLUMNS].fillna(-1000))
```

Writing out your results to a csv for submission

When interpreting your log loss score, keep in mind that the score will change based on the number of samples tested (sometimes it might be normalized?). To get a sense of how this very basic model performs, compare your score to the DrivenData benchmark model performance: 2.0455, which merely submitted uniform probabilities for each class.

In [ ]:
```
predictions = clf.predict_proba(holdout[NUMERIC_COLUMNS].fillna(-1000))
prediction_df = pd.DataFrame(columns=pd.get_dummies(df[LABELS]).columns,
                             index=holdout.index,
                             data=predictions)
prediction_df.to_csv('predictions.csv')
score = score_submission(pred_path = 'predictions.csv')
print('Your model, trained with numeric data only, yields logloss score: {}'.format(score))
```

Tokenizing text

- Tokenization is the process of chopping up a character sequence into pieces called tokens.
- How do we determine what constitutes a token? Often, tokens are separated by whitespace. But we can specify other delimiters as well. For example, if we decided to tokenize on punctuation, then any punctuation mark would be treated like a whitespace. How we tokenize text in our DataFrame can affect the statistics we use in our model.
- A particular cell in our budget DataFrame may have the string content Title I - Disadvantaged Children/Targeted Assistance. The number of n-grams generated by this text data is sensitive to whether or not we tokenize on punctuation, as you'll show in the following exercise.

How many tokens (1-grams) are in the string "Title I - Disadvantaged Children/Targeted Assistance" if we tokenize on punctuation?

Answer is 6.

Tokenizing on punctuation means that Children/Targeted becomes two tokens and - is dropped altogether. See more on N-gram in https://en.wikipedia.org/wiki/N-gram (https://en.wikipedia.org/wiki/N-gram)

Testing your NLP credentials with n-grams

- A python list, one_grams, contains all 1-grams of the string petro-vend fuel and fluids, tokenized on punctuation. one_grams = ['petro', 'vend', 'fuel', 'and', 'fluids']
- Determine the sum of 1-grams, 2-grams and 3-grams generated by the string petro-vend fuel and fluids, tokenized on punctuation.

Answer: The number of 1-grams + 2-grams + 3-grams is 5 + 4 + 3 = 12.

Creating a bag-of-words in scikit-learn

- Study the effects of tokenizing in different ways by comparing the bag-of-words representations resulting from different token patterns.

- Focus on one feature only, the Position_Extra column, which describes any additional information not captured by the Position_Type label.
- Turn the raw text in this column into a bag-of-words representation by creating tokes that contain only alphanumeric characters.
- The first 15 tokens of vec_basic, which splits df.Position_Extra into tokens when it encounters only whitespace characters, have been printed out.

In [13]:
```
from sklearn.feature_extraction.text import CountVectorizer

# Create the token pattern: TOKENS_ALPHANUMERIC
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

df.Position_Extra.fillna('',inplace = True)
#Why the above fillna is necessary?

# Instantiate the CountVectorizer: vec_alphanumeric
vec_alphanumeric = CountVectorizer(token_pattern = TOKENS_ALPHANUMERIC)

vec_alphanumeric.fit(df.Position_Extra)
# After fitting, all the capital letters are transformed into lower-case letter.

# Print the number of tokens and first 15 tokens
msg = "There are {} tokens in Position_Extra if we split on non-alpha numeric"
print(msg.format(len(vec_alphanumeric.get_feature_names())))
print(vec_alphanumeric.get_feature_names()[:15])
```

```
There are 255 tokens in Position_Extra if we split on non-alpha numeric
['1st', '2nd', '3rd', '4th', '5th', '9th', 'a', 'ab', 'accountability', 'adaptive', 'addit', 'additional', 'ad
m', 'admin', 'administrative']
```

Combining text columns for tokenization

- In order to get a bag-of-words representation for all of the text data in the DataFrame, we first convert the text data in each row of the DataFrame into a single string.
- In the previous exercise, this wasn't necessary because we only looked at one column of data, so each row was already just a single string. CountVectorizer expects each row to just be a single string, so in order to use all of the text columns, you'll need a method to turn a list of strings into a single string.
- Use the function definition combine_text_columns() to convert all training text data in your DataFrame to a single string PER ROW that can be passed to the vectorizer object and made into a bag-of-words using the .fit_transform() method.

- The function uses NUMERIC_COLUMNS and LABELS to determine which columns to drop.

In [32]:
```
def combine_text_columns(data_frame, to_drop=NUMERIC_COLUMNS + LABELS):
    #LABELS and NUMERIC_COLUMNS are lists. Labels are to be predicted and also transformed into category type.
    #Numeric_columns are not necessary to transformed into tokens.
    """ converts all text in each row of data_frame to single vector """

    # Drop non-text columns that are in the df
#     print(to_drop)
#     print(data_frame.columns.tolist())
    to_drop = set(to_drop) & set(data_frame.columns.tolist()) #Check the output later. This sentence seems unnece
    # Note the nice way of transform pandas Series to list.
#     print(to_drop)

    text_data = data_frame.drop(to_drop, axis=1)
    #Dropping multiple columns simultaneously.

    # Replace nans with blanks
    text_data.fillna("", inplace=True)

    # Join all text items in a row that have a space in between
    return text_data.apply(lambda x: " ".join(x), axis=1)
```

What's in a token?

- Combine_text_columns to convert all training text data in your DataFrame to a single vector that can be passed to the vectorizer object and made into a bag-of-words using the .fit_transform() method.
- Compare the effect of tokenizing using any non-whitespace characters as a token and using only alphanumeric characters as a token.

In [15]:
```
from sklearn.feature_extraction.text import CountVectorizer

# Create the basic token pattern
TOKENS_BASIC = '\\S+(?=\\s+)'

# Create the alphanumeric token pattern
TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

vec_basic = CountVectorizer(token_pattern = TOKENS_BASIC)

vec_alphanumeric = CountVectorizer(token_pattern = TOKENS_ALPHANUMERIC)

text_vector = combine_text_columns(df)

vec_basic.fit_transform(text_vector)

print("There are {} tokens in the dataset".format(len(vec_basic.get_feature_names())))

vec_alphanumeric.fit_transform(text_vector)

print("There are {} alpha-numeric tokens in the dataset".format(len(vec_alphanumeric.get_feature_names())))
```

```
['FTE', 'Total', 'Function', 'Use', 'Sharing', 'Reporting', 'Student_Type', 'Position_Type', 'Object_Type', 'Pr
e_K', 'Operating_Status']
['Function', 'Use', 'Sharing', 'Reporting', 'Student_Type', 'Position_Type', 'Object_Type', 'Pre_K', 'Operating
_Status', 'Object_Description', 'Text_2', 'SubFund_Description', 'Job_Title_Description', 'Text_3', 'Text_4',
'Sub_Object_Description', 'Location_Description', 'FTE', 'Function_Description', 'Facility_or_Department', 'Pos
ition_Extra', 'Total', 'Program_Description', 'Fund_Description', 'Text_1']
{'Use', 'Operating_Status', 'Sharing', 'Object_Type', 'Function', 'Reporting', 'FTE', 'Pre_K', 'Total', 'Positi
on_Type', 'Student_Type'}
There are 2727 tokens in the dataset
There are 2018 alpha-numeric tokens in the dataset
```

Improving your model

- Explore how the flexibility of the pipeline workflow makes testing different approaches efficient.

Instantiate pipeline

- **No data for the following code**. Here is just a description of the data. The data is stored in the DataFrame, sample_df, which has three kinds of feature data: numeric, text, and numeric with missing values. It also has a **label column with two classes, a and b.**
- Instantiate a pipeline that trains using the numeric column of the sample data.

In [ ]:
```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier

X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric']],
                                                    pd.get_dummies(sample_df['label']),
                                                    random_state=22)

pl = Pipeline([
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])


pl.fit(X_train, y_train)

accuracy = pl.score(X_test, y_test)
print("\nAccuracy on sample data - numeric, no nans: ", accuracy)
#Here we use only standard score. Later we will use logloss score.
```

Preprocessing numeric features

By default, the imputer transformer replaces NaNs with the mean value of the column. That's a good enough imputation strategy for the sample data.

In [ ]:
```
from sklearn.preprocessing import Imputer

X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric','with_missing']],
                                                    pd.get_dummies(sample_df['label']),
                                                    random_state=456)

pl = Pipeline([
        ('imp', Imputer()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
#Note the Imputer() set column NaN with column mean by default. That is: axis = 0, and strategy = 'mean'

pl.fit(X_train, y_train)

accuracy = pl.score(X_test, y_test)
print("\nAccuracy on sample data - all numeric, incl nans: ", accuracy)
```

Preprocessing text features

- Perform a similar preprocessing pipeline step, but will use the text column from the sample data.
- Use CountVectorizer() to generate a bag-of-words representation of the data. Using the default arguments, add a (step, transform) tuple to the steps list in the pipeline.
- Select only the text column for splitting training and test sets.

In [ ]:
```
from sklearn.feature_extraction.text import CountVectorizer

X_train, X_test, y_train, y_test = train_test_split(sample_df['text'],
                                                    pd.get_dummies(sample_df['label']),
                                                    random_state=456)

pl = Pipeline([
        ('vec', CountVectorizer()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])

pl.fit(X_train, y_train)

accuracy = pl.score(X_test,y_test)
print("\nAccuracy on sample data - just text data: ", accuracy)
```

Multiple types of processing: FunctionTransformer

- **Any step in the pipeline must be an object that implements the fit and transform methods. The FunctionTransformer creates an object with these methods out of any Python function that you pass to it**.
- Numeric data that needs imputation, and text data that needs to be converted into a bag-of-words. Create functions that separate the text from the numeric variables and see how the .fit() and .transform() methods work.

In [ ]:
```
from sklearn.preprocessing import FunctionTransformer

get_text_data = FunctionTransformer(lambda x: x['text'], validate=False)
#My Comments: Because object in each step of a pipeline need have at least two methods defined: fit() and transfo
#Normal Python object does not necessarily have these two methods. So we we need transform the to-be used functio
#have such two methods. This is at least the effect of FunctionTransFormer()? Use dir() I found the above stateme

get_numeric_data = FunctionTransformer(lambda x: x[['numeric', 'with_missing']], validate=False)

# Fit and transform the text data: just_text_data
just_text_data = get_text_data.fit_transform(sample_df)

# Fit and transform the numeric data: just_numeric_data
just_numeric_data = get_numeric_data.fit_transform(sample_df)

print('Text Data')
print(just_text_data.head())
print('\nNumeric Data')
print(just_numeric_data.head())
```

Multiple types of processing: FeatureUnion

- These tools will allow you to streamline all preprocessing steps for your model, even when multiple datatypes are involved. Here, for example, you don't want to impute our text data, and you don't want to create a bag-of-words with our numeric data. Instead, you want to deal with these separately and then join the results together using FeatureUnion().
- We still have only two high-level steps in your pipeline: preprocessing and model instantiation. The difference is that the first preprocessing step actually consists of a pipeline for numeric data and a pipeline for text data. The results of those pipelines are joined using FeatureUnion().

In [ ]:

```
from sklearn.pipeline import FeatureUnion

X_train, X_test, y_train, y_test = train_test_split(sample_df[['numeric', 'with_missing', 'text']],
                                                    pd.get_dummies(sample_df['label']),
                                                    random_state=22)

# Create a FeatureUnion with nested pipeline: process_and_join_features
# Note we cannot impute() in one step, and then use CountVectorizer in the next step in a pipeline.
# This is because when doing impute(), it does not know how to impute the text part. similarly, when we do CountV
# it does not know how to do this on numerical data. So we need do them separately and then combine---unionize.
process_and_join_features = FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    # Here get_numerical_data object in Pipeline must have fit and transform method.
                    #So we need FunctionTransformer() as used before.
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer())
                ]))
             ]
        )
# The above returns a FeatureUnion Object
#My Comments: Because object in each step of a pipeline need have at least two methods defined: fit() and transfo
#If we combine the two objects with normal functions, then the combined object might not have the two methods mer
#However, using FeatureUnion(), then the returned object will have the two methods required by skilearn. Verify t
# Furthermore, #using FeatureUnion, we let two tasks undergoing separately and then join, but not one after anoth

pl = Pipeline([
        ('union', process_and_join_features),
        #process_and_join_features is a FeatureUnion object and it contains two sub-pipelines here.
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])

pl.fit(X_train, y_train)

accuracy = pl.score(X_test, y_test)
print("\nAccuracy on sample data - all data: ", accuracy)
```

Using FunctionTransformer on the main dataset

Use FunctionTransformer on the primary budget data, before instantiating a multiple-datatype pipeline in the next exercise.

In [26]:
```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import FunctionTransformer
from sklearn.pipeline import FeatureUnion
from sklearn.preprocessing import Imputer

dummy_labels = pd.get_dummies(df[LABELS])

# Get the columns that are features in the original df. List comprehension and lambda function are very convenier
NON_LABELS = [c for c in df.columns if c not in LABELS]

X_train, X_test, y_train, y_test = multilabel_train_test_split(df[NON_LABELS],dummy_labels,size=0.2,seed=123)

# Preprocess the text data: get_text_data
get_text_data = FunctionTransformer(combine_text_columns, validate=False)
#Be sure to understand the above format, which is different from the function below.

# Preprocess the numeric data: get_numeric_data
get_numeric_data = FunctionTransformer(lambda x: x[NUMERIC_COLUMNS], validate=False)
```

Add a model to the pipeline

In [27]:
```
# The nested pipeline initialization is more clear than the step-by-step introduction.
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer())
                ]))
             ]
        )),
        ('clf', OneVsRestClassifier(LogisticRegression()))
        # ('clf', RandomForestClassifier())
        # ('clf', RandomForestClassifier(n_estimators=15))
    ])


pl.fit(X_train, y_train)

accuracy = pl.score(X_test, y_test)
print("\nAccuracy on budget dataset: ", accuracy)
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\utils\deprecation.py:58: DeprecationWarning: Class Imputer i
s deprecated; Imputer was deprecated in version 0.20 and will be removed in 0.22. Import impute.SimpleImputer f
rom sklearn instead.
  warnings.warn(msg, category=DeprecationWarning)

['FTE', 'Total', 'Function', 'Use', 'Sharing', 'Reporting', 'Student_Type', 'Position_Type', 'Object_Type', 'Pr
e_K', 'Operating_Status']
['Object_Description', 'Text_2', 'SubFund_Description', 'Job_Title_Description', 'Text_3', 'Text_4', 'Sub_Objec
t_Description', 'Location_Description', 'FTE', 'Function_Description', 'Facility_or_Department', 'Position_Extr
a', 'Total', 'Program_Description', 'Fund_Description', 'Text_1']
{'FTE', 'Total'}

C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\linear_model\logistic.py:433: FutureWarning: Default solver
will be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\multiclass.py:76: UserWarning: Label not 9 is present in all
training examples.
  str(classes[c]))
```

```
['FTE', 'Total', 'Function', 'Use', 'Sharing', 'Reporting', 'Student_Type', 'Position_Type', 'Object_Type', 'Pr
e_K', 'Operating_Status']
['Object_Description', 'Text_2', 'SubFund_Description', 'Job_Title_Description', 'Text_3', 'Text_4', 'Sub_Objec
t_Description', 'Location_Description', 'FTE', 'Function_Description', 'Facility_or_Department', 'Position_Extr
a', 'Total', 'Program_Description', 'Fund_Description', 'Text_1']
{'FTE', 'Total'}

Accuracy on budget dataset:  0.4350324837581209
```

Try a different class of model

- **One of the great strengths of pipelines** is how easy they make the process of testing different models.
- As in the above code, we can change one line to try different models.

Can you adjust the model or parameters to improve accuracy?

- Try changing the parameter n_estimators of RandomForestClassifier(), whose default value is 10, to 15.
- It's time to get serious and work with the log loss metric. Learn expert techniques in the next chapter to take the model to the next level.

Learning from the experts

How many tokens?

- Use alpha-numeric sequences as tokens. Alpha-numeric tokens contain only letters a-z and numbers 0-9 (no other characters). In other words, you'll tokenize on punctuation to generate n-gram statistics. In this case, how many tokens are in the following string from the main dataset?
  'PLANNING,RES,DEV,& EVAL '
  Answer 4.

Deciding what's a word

- Use CountVectorizer on the training data X_train to see the effect of tokenization on punctuation.
- Since CountVectorizer expects a vector, we need the function, combine_text_columns before fitting to the training data.

In [42]:
```
from sklearn.feature_extraction.text import CountVectorizer

text_vector = combine_text_columns(X_train)
# print(text_vector[0:5])

TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

text_features = CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC)

text_features.fit(text_vector)

print(text_features.get_feature_names()[:10])
```

```
['00a', '12', '1st', '2nd', '3rd', '4th', '5', '5th', '70', '70h']
```

N-gram range in scikit-learn

- Insert a CountVectorizer instance into your pipeline for the main dataset, and compute multiple n-gram features to be used in the model.
- In order to look for ngram relationships at multiple scales, use the ngram_range parameter.
- Add new steps in the pipeline, the dim_red step following the vectorizer step , and the scale step preceeding the clf (classification) step. dim_red step performs a dimensionality reduction technique, and scale step performs a scaling.
- The dim_red step uses a scikit-learn function called SelectKBest(), applying something called the **chi-squared test** to select the K "best" features. The scale step uses a scikit-learn function called MaxAbsScaler() in order to squash the relevant features into the interval -1 to 1.

In [ ]:
```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import Imputer
from sklearn.feature_selection import chi2, SelectKBest

# Select 300 best features
chi_k = 300

from sklearn.preprocessing import FunctionTransformer, MaxAbsScaler
from sklearn.pipeline import FeatureUnion

get_text_data = FunctionTransformer(combine_text_columns, validate=False)
get_numeric_data = FunctionTransformer(lambda x: x[NUMERIC_COLUMNS], validate=False)

TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC,
                                                   ngram_range=(1, 2))),

                    ('dim_red', SelectKBest(chi2, chi_k)) #newly added
                ]))
             ]
        )),
        ('scale', MaxAbsScaler()), #newly added. We have used a similar scaling function before
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```

Implement interaction modeling in scikit-learn

- Add interaction features to your model. The PolynomialFeatures object in scikit-learn does just that, but here you're going to use a custom interaction object, SparseInteractions. Interaction terms are a statistical tool that lets your model express what happens if two features **appear together in the same row**.
- SparseInteractions does the same thing as PolynomialFeatures, but it uses sparse matrices to do so.
- PolynomialFeatures and SparseInteractions both take the argument degree, which tells them what polynomia degree of interactions to compute.
- Consider interaction terms of degree=2 in your pipeline.
- Pipelines with interaction terms take a while to train (since making n features into n-squared features!). So actually the calculation here is very expensive for a PC.

In [ ]:

```python
from itertools import combinations

import numpy as np
from scipy import sparse
from sklearn.base import BaseEstimator, TransformerMixin


class SparseInteractions(BaseEstimator, TransformerMixin):
    def __init__(self, degree=2, feature_name_separator="_"):
        self.degree = degree
        self.feature_name_separator = feature_name_separator

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        if not sparse.isspmatrix_csc(X):
            X = sparse.csc_matrix(X)

        if hasattr(X, "columns"):
            self.orig_col_names = X.columns
        else:
            self.orig_col_names = np.array([str(i) for i in range(X.shape[1])])

        spi = self._create_sparse_interactions(X)
        return spi

    def get_feature_names(self):
        return self.feature_names

    def _create_sparse_interactions(self, X):
        out_mat = []
        self.feature_names = self.orig_col_names.tolist()

        for sub_degree in range(2, self.degree + 1):
            for col_ixs in combinations(range(X.shape[1]), sub_degree):
                # add name for new column
                name = self.feature_name_separator.join(self.orig_col_names[list(col_ixs)])
                self.feature_names.append(name)

                # get column multiplications value
                out = X[:, col_ixs[0]]
```

```
                for j in col_ixs[1:]:
                    out = out.multiply(X[:, j])

                out_mat.append(out)

        return sparse.hstack([X] + out_mat)
```

In [ ]:
```
# Instantiate pipeline: pl
pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', CountVectorizer(token_pattern=TOKENS_ALPHANUMERIC,
                                                   ngram_range=(1, 2))),
                    ('dim_red', SelectKBest(chi2, chi_k))
                ]))
             ]
        )),
        ('int', SparseInteractions(degree = 2)),
        ('scale', MaxAbsScaler()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```

Log loss score: 1.2256. Nice improvement from 1.2681!

Why is hashing a useful trick?

- By explicitly stating how many possible outputs the hashing function may have, we limit the size of the objects that need to be processed. With these limits known, computation can be made more efficient and we can get results faster, even on large datasets.
- Need figure out the how hashing trick is used in HashingVectorizer.

Implementing the hashing trick in scikit-learn

In this exercise you will check out the scikit-learn implementation of HashingVectorizer before adding it to your pipeline later.

As you saw in the video, HashingVectorizer acts just like CountVectorizer in that it can accept token_pattern and ngram_range parameters. The important difference is that it creates hash values from the text, so that we get all the computational advantages of hashing!

In [ ]:
```
from sklearn.feature_extraction.text import HashingVectorizer

text_data = combine_text_columns(X_train)

TOKENS_ALPHANUMERIC = '[A-Za-z0-9]+(?=\\s+)'

hashing_vec = HashingVectorizer(token_pattern=TOKENS_ALPHANUMERIC)

hashed_text = hashing_vec.fit_transform(text_data)

hashed_df = pd.DataFrame(hashed_text.data)
print(hashed_df.head())
```

```
   17768  0.400000
   17769 -0.200000
   17770  0.600000
   17771 -0.200000
     .....
   17768  0.400000
   17769 -0.200000
   17770  0.600000
   17771 -0.200000

   [17772 rows x 1 columns]
```

As you can see, some text is hashed to the same value, but as Peter mentioned in the video, this doesn't neccessarily hurt performance.

Build the winning model

- Add the HashingVectorizer step to the pipeline to replace the CountVectorizer step.
- The parameters non_negative=True, norm=None, and binary=False make the HashingVectorizer perform similarly to the default settings on the CountVectorizer, so we can just replace one with the other.

In [ ]:
```
from sklearn.feature_extraction.text import HashingVectorizer

pl = Pipeline([
        ('union', FeatureUnion(
            transformer_list = [
                ('numeric_features', Pipeline([
                    ('selector', get_numeric_data),
                    ('imputer', Imputer())
                ])),
                ('text_features', Pipeline([
                    ('selector', get_text_data),
                    ('vectorizer', HashingVectorizer(token_pattern=TOKENS_ALPHANUMERIC,
                                                     non_negative=True, norm=None, binary=False,
                                                     ngram_range=(1,2))),
                    ('dim_red', SelectKBest(chi2, chi_k))
                ]))
             ]
        )),
        ('int', SparseInteractions(degree=2)),
        ('scale', MaxAbsScaler()),
        ('clf', OneVsRestClassifier(LogisticRegression()))
    ])
```

Well done! Log loss: 1.2258. Looks like the performance is about the same, but this is expected since the HashingVectorizer should work the same as the CountVectorizer.

What tactics got the winner the best score?

The Jupyter notebook in the following link contains all the code. https://github.com/datacamp/course-resources-ml-with-experts-budgets/blob/master/notebooks/1.0-full-model.ipynb (https://github.com/datacamp/course-resources-ml-with-experts-budgets/blob/master/notebooks/1.0-full-model.ipynb)

The main advantage of the winner is not try different models, not use deep neural network..., but lies in the skillful NLP, efficient computation, and simple but powerful stats tricks to master the budget data. Often times simpler is better, and understanding the problem in depth leads to simpler solutions!

Comments: The project above may still have room to improve:

- Nonlinearity is considered only up to 2nd order. It might be use kernel trick in either logistic regression, or SVM.
- Bias-variance trade-off, grid-search, though expensive, might be a way to do.