Machine Learning Engineer Nanodegree

Capstone Project: Predicting Budget Cost Codes Based on Purchase Order Information

Daniel Kelly
Dan_kelly@telus.net
June 8th, 2019

I. Definition

Project Overview

I currently work for a Construction Management company and one critical function of a Construction Management company is creating and tracking the budget of a construction project during the entire lifecycle of the construction project.

During the construction of the project, the budget information may be used by developers/clients to track the performance of the architect, construction management team, and subcontractors. It can also be used to identify when a project may go over budget and require additional financing from their lender.

By breaking down the project budget into detailed sections like concrete, appliances, windows, etc. you can identify potential performance issues with individual subcontractors, products, or design elements and take steps to correct or mitigate them moving forward.

Additionally, when a project is complete you can then use the information in the finished budget to estimate the costs of new projects. Again, by breaking the budget into more granular sections, you can get more detailed information on what the costs of new projects are likely to be and how changes to the design or materials may affect that cost.

As a result of these use cases, it is important then to have accurate cost data and ensure that all costs are allocated to their respective budget sections. To do this, you use numerical Cost Codes that are unique and mutually exclusive. One numerical cost code is defined for each line in the budget. Every cost that arises during the construction of a

project, from concrete orders to safety equipment, is allocated to a cost code within the project budget.

For consistency, an organization will define a master list of every possible cost code that can be used in a budget, then a project will use only a subset of these codes that are relevant to the type project.

The projects that my company manages range in budget from between $1,000,000 and $200,000,000 with between 100 and 900 cost codes depending on their size and scope.

In this project I created a machine learning model to predict the budget category (Cost Code) that is associated with purchase orders (POs) created during a construction project. This project requires evaluating both text description and accompanying numerical data of a PO item to predict its category. This classification problem is similar to classifying bank transactions, Automatic Classification of Bank Transaction[1] paper reviews some techniques for addressing this.

This paper, however, did not include any numerical transaction data in the prediction, nor did it explore using an ensemble of multiple models as I will.

The dataset that I used for this project consisted of an export of all the purchase order items from my company's ERP system SQL database. The raw CSV file can be found here. I also used an export of the master list of cost codes and their description, found here. This raw dataset contains over 39,000 records.

Problem Statement

Currently, Project Coordinators on the construction site must review and assign a cost code to each item on a purchase order so it can be accounted for in the correct location of the construction project's budget. The Project Manager then reviews and confirms or modifies the correct cost code is selected for an item before committing it to the budget.

This is a time consuming and therefore expensive process for both the Project Coordinator and Project Manager that must be completed manually at least once per month for every purchase order item before the supplier of the purchase order can be paid and the costs of the project can be entered to the budget. A completed project will have between 1,000 and 10,000 purchase order items that will have needed to be manually coded.

This process cannot feasibly be accomplished by a one-to-one mapping of products to cost codes because products are continuously added or changed, different projects may

use different vendors, vendors are continuously being added, one product may be associated with different cost codes depending on how it is used, and many other factors.

I propose that one way to reduce the amount of time and the expense of choosing the correct cost code for an cost is to implement a predictive model that will use the data that is present on a purchase order (The vendor name, product description, cost, etc.) to predict what the associated cost code should be used for each item on a purchase order.

The solution that I propose includes the following:

- Preprocess the data
    - Discard outliers and null values
    - Normalize numerical data
    - One hot encode Categorical data
- Predict the cost code based on the text data
    - Split text from numerical/categorical features
    - Extract features from text data using CountVectorizer and TFIDF
    - Make a prediction based on the text features using either an SVM, Naive Bayes, or Logistic Regression classifier
- Predict the cost code based on numerical/categorical features + the prediction from the text data
    - Append the prediction from the text feature to the numerical/categorical feature dataset
    - Predict the cost code using either a Random Forest or KNeighbors classifier.

The output of this solution will be a prediction of the correct cost code that the purchase order item should be associated with. To arrive at the optimal solution, when training the algorithms I will use gridsearch to tune the algorithm's hyperparameters. I will also use SMOTE to reduce the effect of the imbalanced classes in the dataset by synthetically creating more data points and balancing the amount of data for each class.

There are several characteristics of this dataset that have shaped my proposed solution to this problem. The first characteristic is that there are text, numeric, and categorical features that make up each sample. While categorical features can be encoded (either through one hot encoding or label encoding) and then included with the numeric features for training and prediction, text data needs to be heavily pre-processed and handled differently. While it is possible to vectorize the text and include it as a single

feature with the other features, then use one algorithm to make a prediction, this method does not take advantage of the fact that different algorithms can have higher performance on different sets of data or data types. Secondly, the solution I propose uses stacking two different algorithms which generally provides higher performance than using a single algorithm.[2] For these reasons, I believe my solution of stacking two algorithms, each chosen and optimized for their respective types of features, is a good fit for this problem.

Metrics

This is a supervised classification problem where we are trying to predict what cost code a purchase order item belongs to. I believe that the best evaluation for this metric is an F1-Score. I based this decision on my findings in the data exploration workbook.

The conclusion of my findings is that the dataset is very unbalanced. The most used cost code appears 4,157 times, whereas the mean usage of a cost code is 97 times with a median of 5.5. This means that there are a small number of codes that are used a large number of times, but most codes are used relatively rarely.

This imbalance makes accuracy poor measure of performance because, as discussed in the benchmark model section, the model can achieve an accuracy of almost %15 by always predicting the most common cost code despite being an obviously poor model. This makes F1-score a better metric for the performance of this model than accuracy. The formula for f1-score is:

Figure 1. F1 Metric Equation. Source[3]

$$F1 = 2 \times \frac{Precision * Recall}{Precision + Recall}$$

By using the f1-score we get a balance between precision and recall that better reflects the performance of the model when compared to accuracy.

Fbeta-score is another metric that could be useful, however, this metric is used to weight either precision or recall higher than the other. This would be used if one metric was more important than the other. Eg. Recall is more important if the cost of a false negative is higher than the cost of a false positive. In the case of this model, since we are suggesting cost codes to an end-user, the cost of a false positive is the same as a false negative.

As a result, F1-score is the metric I have used to evaluate the model's performance but with recall and precision individually for reference.

II. Analysis

Data Exploration

The dataset that I will use for this project is a CSV list of purchase order items that I have exported from the SQL server database of my company's ERP system and obtained permission to use. This file contains over 39,000 examples of purchase order items, their accompanying information and their corresponding cost codes. These purchase order items have been previously entered into the company's ERP system and had their cost codes selected manually over the course of over 5 years and several construction projects. This labelled data will be split up and used as the training, validation, and test sets for the model.

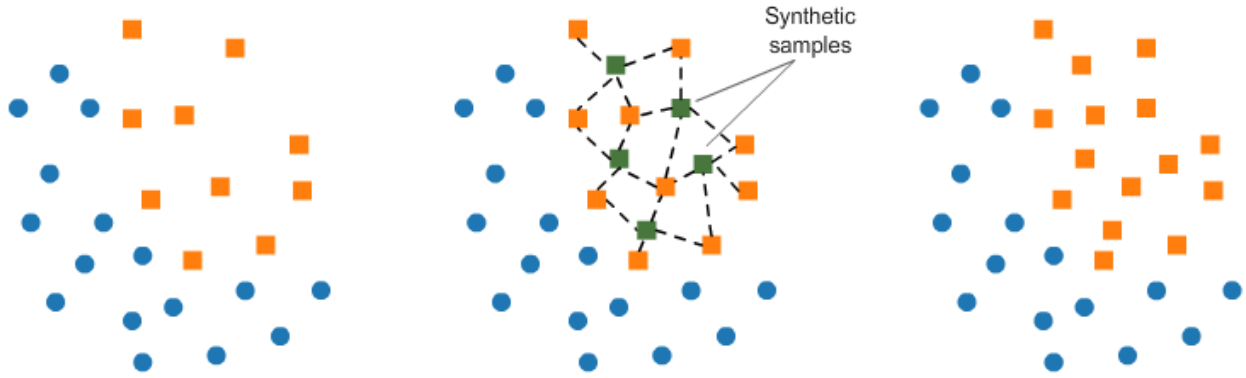
As previously mentioned the raw export contains over 39,000 records, however through some data exploration, documented in this workbook, I found that there were some records that would not be helpful for achieving an accurate model. For example, there were some items with negative cost amounts which would correspond with credits that we had received from a supplier for items. These records are not relevant to the purchase of new items.

As noted in the data exploration workbook, this is an unbalanced dataset with a few cost codes being used much more than others. This creates a problem where just randomly splitting the dataset into training, test, and validation sets could introduce a significant amount of bias. I will mitigate this by using the stratifying feature of sklearn's train_test_split and specifying the Cost Code column. This will maintain the ratio of codes relative to each other in each of the datasets.

Because splitting the data will reduce the number of samples I have for training, I can use SMOTE (Synthetic Minority Oversampling TEchnique) to generate more data points based on the existing information, giving my model more data to train with.

Figure 2. Synthetic Minority Oversampling TEchnique (SMOTE).

Source[4]



There are 9 features in this dataset, plus the variable that I want to predict. I will be using 7 of the features:

Figure 3. PO Dataset Head

Output

	Company #	Purchase Order	Item	Vendor	Description	Unit of Measure	Units	Unit Cost	Cost	Cost Code
0	8	1200-001	1	Paragon Electrical Installations Ltd.	Additional smoke detector/re-verification	LS	0	0.0	1444.0	26-20-20
1	8	1200-002	1	Accurate Aluminum Ltd	S&I railing as per quote Aug. 13 2015	LS	0	0.0	500.0	05-52-20
2	8	1200-003	1	Dura Productions	S&I metal ramp	LS	0	0.0	795.0	05-52-20
3	8	1200-004	1	Friesen Floors & Window Fashions Ltd	S&I hardwood flooring for enclosed balcony area	LS	0	0.0	2314.0	09-64-33
4	8	1209-1-01	1	Alba Painting Ltd.	Painting of two offices	LS	0	0.0	900.0	09-91-40

- **Company #** This is used internally to identify which internal company the project is associated with. It is not relevant to the prediction and will not be used.
- **Purchase Order** This is the purchase order number, this is not relevant to the prediction.
- **Item** This feature defines which item this record was on the purchase order, as purchase orders may contain multiple products. It is not useful for this prediction.
- **Vendor** This is the name of the vendor who sold the item on the purchase order. I will use one-hot encoding for this nominal data.
- **Description** This is a text field that contains the description of the item purchased. This should be the most important feature in the data.
- **Unit of Measure** This feature tells us how the units of each item are measured. Most commonly LS - lump sum or EA - each. I will one-hot encode this feature.

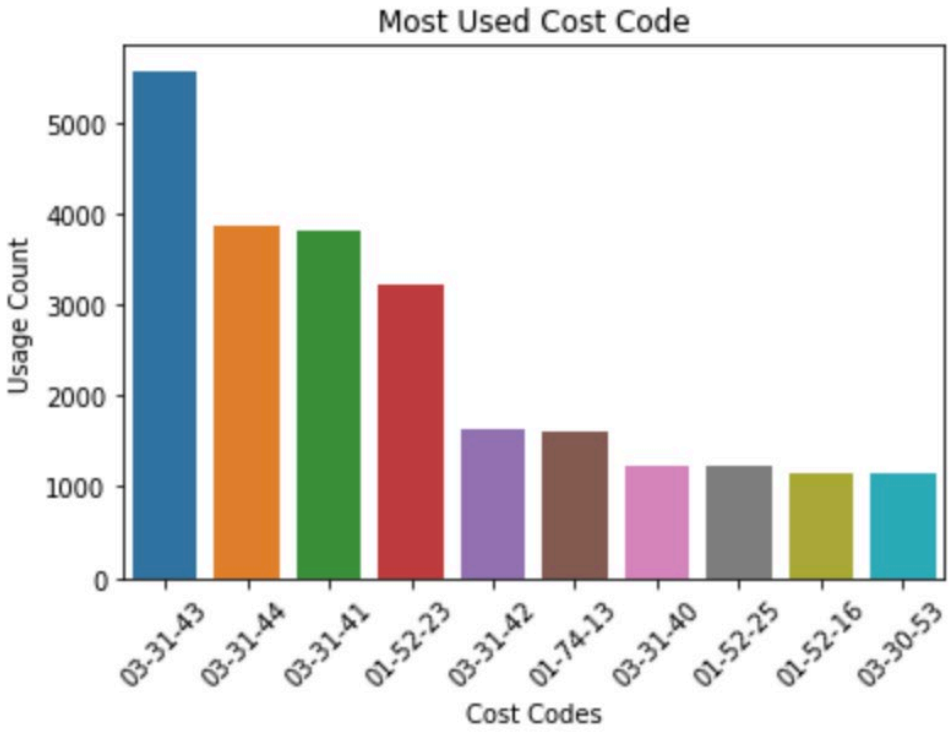
- **Units** How many of these items were ordered. I will use this feature after normalizing it.
- **Unit Cost** How much each unit ordered costs. I will use this feature after normalizing it.
- **Cost** This is the total cost of the line item (Units * Unit Cost). I will use this feature after normalizing it.
- **Cost Code** This is the variable that my model will predict and will be used for training. There are 905 unique cost codes in the master list, however, only 354 are used in the purchase orders in the dataset.

This dataset is very unbalanced, the average number of times a cost code is used is 111, but the median is 6.5. This means that there are a few cost codes that are used many more times than the others. For example, 03-31-43 concrete material - above grade verticals is used 5570 times.

Exploratory Visualization

Figure 4 shows the ten most used cost codes.

Figure 4. Most Used Codes in PO Dataset



Furthermore, there are a small number of very high-value POs or POs with a large number of Units that skew the data. Figure 5 shows, for example, that the Units feature has a maximum value of over 100,000 while the 75th percentile is under 11.

Figure 5. Description of Numerical and Categorical Features

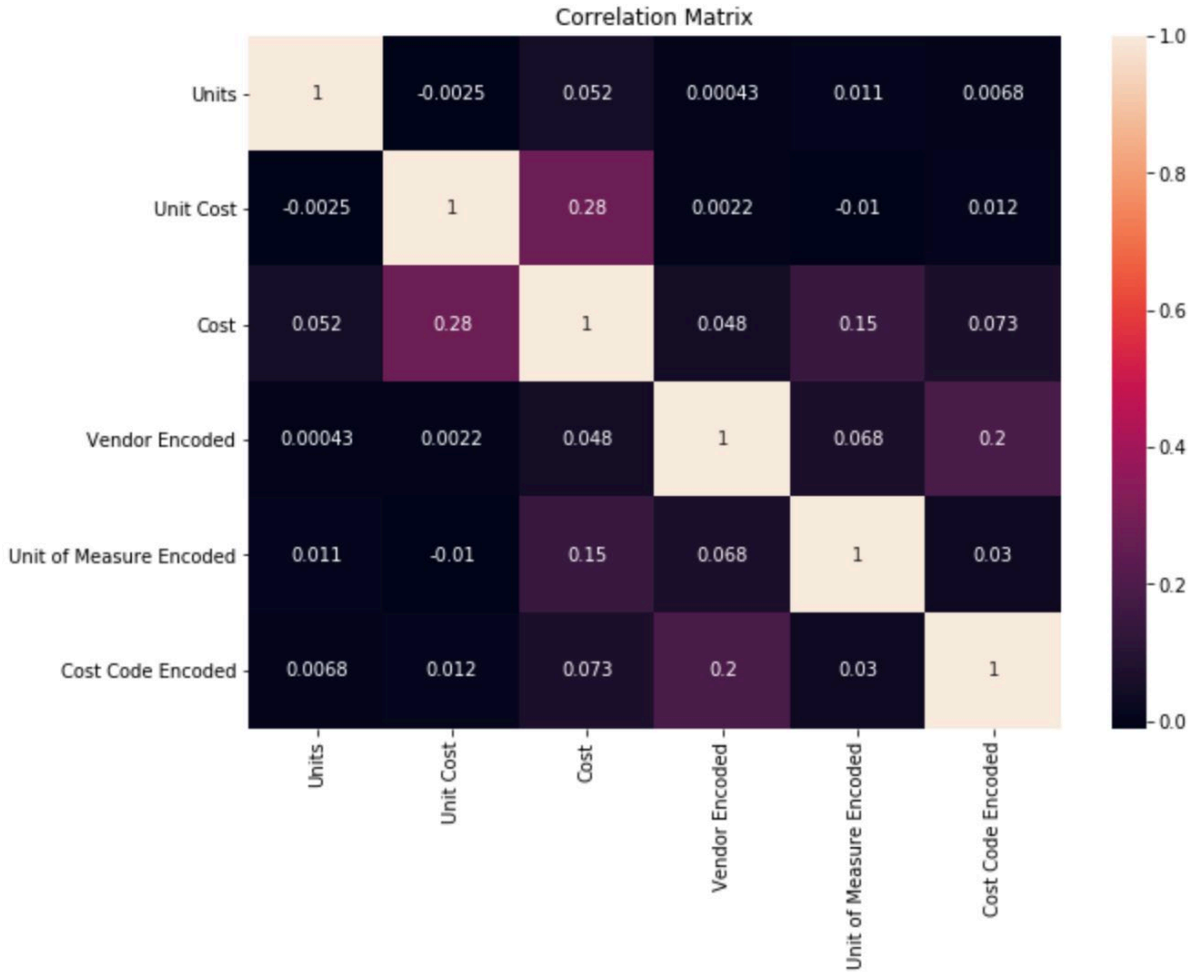
	Units	Unit Cost	Cost
count	39141.000000	39141.000000	39141.000000
mean	27.998488	149.901813	1383.047704
std	624.646593	1900.609644	6939.240080
min	0.000000	0.000000	0.000000
25%	1.000000	2.150000	37.800000
50%	2.700000	9.800000	150.000000
75%	11.000000	64.000000	700.000000
max	102726.000000	300000.000000	639612.130000

These POs are outliers and will be removed.

Figure 6 shows the correlation between the numerical and categorical features in the dataset.

Figure 6. Correlation Matrix of Numerical and Categorical

Features



The correlation matrix fig 6. shows us that the Vendor is the most closely correlated variable with the cost code, followed by the overall cost of the item. Intuitively, the vendor having a high correlation with the cost code makes sense. For the most part, vendors each sell a certain type of product related to its function. For example, a vendor called "Advanced Safety Supplies" sells mostly safety-related equipment that would be budgeted to a "safety supplies" cost code.

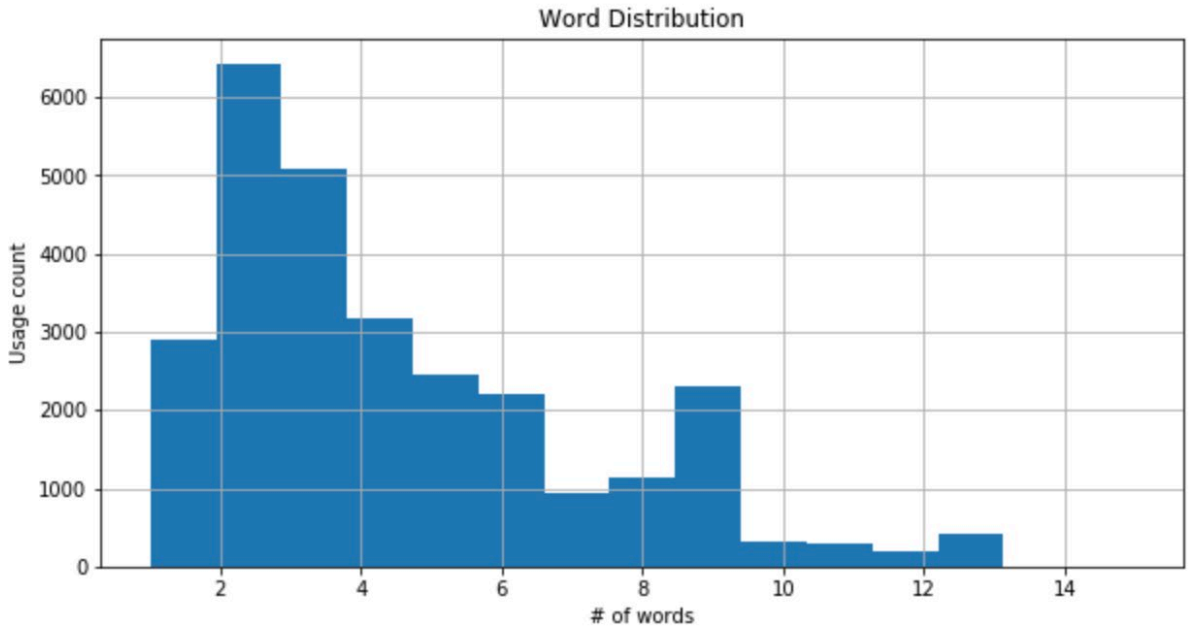
The cost and unit cost being closely correlated also makes sense, because the cost of a line item is just a multiple of the unit cost.

What I found surprising was that the unit cost of an item was not very closely correlated to the cost code. I would have expected the unit cost to be very closely related to the particular item being purchased, which would then correspond to a particular cost code. It could be that product prices have changed over the years, or with different vendors.

It's also possible that many products have similar prices, or simply that end users did not bother to put in the unit cost and just entered the total cost of the line item.

Looking at the text data in the Description feature (fig. 7), we can see that the majority of PO descriptions have between 2 and 6 words in them.

Figure 7. Word Usage Distribution from Description Feature



And the most frequently used words (fig. 8) are:

Figure 8. Most Commonly User words in the Description Feature

	Word	Frequecy
0	to	4805
1	concrete	4174
2	per	3529
3	all	3210
4	fuel	2670
5	applies	2564
6	charge	2445
7	m3	2281
8	environmental	2276
9	surcharge	2047
10	load	1729
11	and	1514
12	1	1467
13	14	1112
14	levy	996
15	mar	826
16	invoice	818
17	14mm	816
18	slump	776
19	carbon	716

Note that I did not remove stop-words from this dataset in exploration or in the training. This is because there is some research to suggest that removing stopwords can have a negative effect on classification performance.[5]

Algorithms and Techniques

Algorithms selected for text classification

- SGD Classifier
  - The Stochastic Gradient Descent classifier that performs similar to Logistic Regression, but is particularly suited to handle data sets with high dimensionality and is commonly used in Natural Language Processing (NLP) and text classification, due to its efficiency. This makes it suitable for the text classification portion of my solution. However, the number of hyperparameters and the complexity of tuning the algorithm can present an issue.
- Logistic Regression
  - This is a good classifier to use for high dimensional and sparse datasets, which matches the PO dataset that becomes very sparse when we encode the vendors feature using one hot encoding. It's also simple to understand and less complicated than an SVM
  - However, logistic regression assumes that the data is linearly separable and in the case of this dataset that is not something that I know.
- Naive Bayes
  - This algorithm has been used in text classification for decades and serves as a good baseline for this project. It scales easily and can be implemented quickly. Naive Bayes, while offering competitive performance, is often outperformed by more complex models if they are properly tuned.
  - Because this is not a binary classification problem, there are multiple classes, I will need to use the Multinomial Naive Bayes classifier.

Algorithms selected for numerical and categorical classification

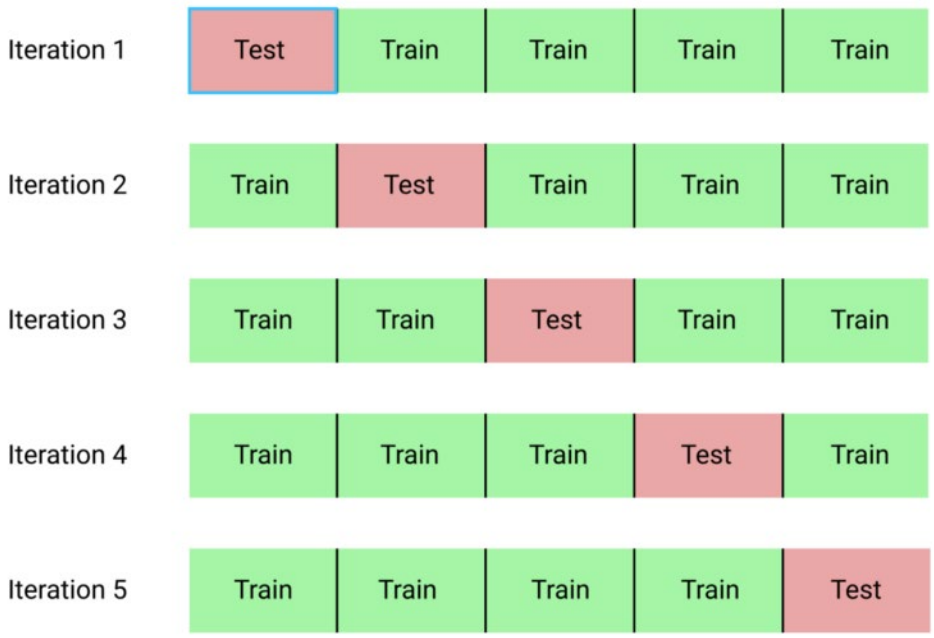
- Random Forest ensemble
  - Generally performs well on most datasets, less susceptible to overfitting because it combines multiple estimators for the final result.
  - Training and prediction times can be slow, but because the dataset is relatively small and the end use of the model is less sensitive to prediction times, these are not critical issues.
  - This algorithm may perform better on the dataset than K Nearest Neighbors because it is better able to handle data sets with higher dimensionality.
- K Nearest Neighbors (KNN)
  - Simple to understand and implement, this algorithm has few parameters to tune. KNN does not assume any underlying structure to the data and usually works best with smaller datasets with fewer dimensions, otherwise, prediction times can be long.

- I wanted to try this algorithm because it is simple and different from the algorithms chosen for text classification, which in theory, is beneficial when stacking models.
- Also, KNN does not assume any underlying structure to the data, and I am unsure if such a structure exists in the dataset for this project.

Gridsearch Cross-Validation hyperparameter tuning

- Gridsearch is a technique that allows you to specify multiple values for the hyperparameters of an algorithm, it then iterates through every combination of these parameters and outputs the parameters that produce the highest score.
- Scoring of the iterations is determined by a scoring metric that you select. In this case, because the metric I chose to evaluate the overall performance of the solution is F1-Score, I also used this to score the iterations in gridsearch.
- To minimize overfitting and underfitting, k-fold cross-validation is used to split the training data into groups, hold one group out and test on the rest. Because some of the classes in the PO dataset have only 10 samples (after discarding classes with fewer than 10) I configured the gridsearch to use 5 folds, so each fold would have 2 examples of each class.
- Figure 9 shows how the data is split into groups and training and testing is done in iterations using 5 fold cross- validation.

Figure 9. Cross-Validation



Synthetic Minority Over-sampling TEchnique (SMOTE)

- SMOTE is an over-sampling technique used to address imbalanced classes in a dataset, where one class has many more samples in the dataset than others. It uses a data point and it's K nearest neighbors to generate new data. Figure 10 contains the pseudo-code that explains how SMOTE works.

Figure 10. SMOTE pseudo-code. Source

Algorithm *SMOTE*(T, N, k)
Input: Number of minority class samples T; Amount of SMOTE $N\%$; Number of nearest neighbors k
Output: $(N/100) * T$ synthetic minority class samples

(∗ *If N is less than 100%, randomize the minority class samples as only a random percent of them will be SMOTEd.* ∗)
if $N < 100$
  then Randomize the T minority class samples
    $T = (N/100) * T$
    $N = 100$
endif
$N = (int)(N/100)$ (∗ *The amount of SMOTE is assumed to be in integral multiples of 100.* ∗)
k = Number of nearest neighbors
$numattrs$ = Number of attributes
$Sample[\,][\,]$: array for original minority class samples
$newindex$: keeps a count of number of synthetic samples generated, initialized to 0
$Synthetic[\,][\,]$: array for synthetic samples
(∗ *Compute k nearest neighbors for each minority class sample only.* ∗)
for $i \leftarrow 1$ **to** T
    Compute k nearest neighbors for i, and save the indices in the $nnarray$
    Populate(N, i, $nnarray$)
endfor

Populate(N, i, $nnarray$) (∗ *Function to generate the synthetic samples.* ∗)
while $N \neq 0$
    Choose a random number between 1 and k, call it nn. This step chooses one of the k nearest neighbors of i.
    for $attr \leftarrow 1$ **to** $numattrs$
        Compute: $dif = Sample[nnarray[nn]][attr] - Sample[i][attr]$
        Compute: gap = random number between 0 and 1
        $Synthetic[newindex][attr] = Sample[i][attr] + gap * dif$
    endfor
    $newindex{+}{+}$
    $N = N - 1$
endwhile
return (∗ *End of Populate.* ∗)
End of Pseudo-Code.

- I chose to use SMOTE to augment this dataset because, as noted in the dataset exploration, the classes are very imbalanced. The most used cost code has more than 50 times more samples than the average cost code. Using SMOTE balances the classes so they all have similar numbers of samples and one class does not skew the predictions of the algorithm trained on the data.

- I installed and used the imbalanced-learn python package which contains the SMOTE module.
- In practice, using SMOTE is simple; import the module, create new X and y datasets by fitting to the original X and y data.

```
from imblearn.over_sampling import SMOTE
X_resampled, y_resampled = SMOTE().fit_resample(X, y)
```

Benchmark

Currently, the processes for selecting cost codes for a purchase order items is entirely manual. We do not have statistics for how accurate the initial cost coding is, or how often Project Managers change cost codes when they are reviewing them. Additionally, we also have no documented data on relationships between the information on a purchase order and the cost codes. Therefore, without any additional data on what the correlation is between a cost code and the information in the purchase order, I believe that the most relevant benchmark model would be to use a model that always predicts the most commonly used cost code. After doing data exploration and clean up, the most commonly used cost code was used 4,157 times out of a total of 28,446 records so a model that always predicted this cost code would have an accuracy of almost 15%.

III. Methodology

Data Preprocessing

The dataset that I used for this project required several data processing steps that I identified in the Data Explorationnotebook, and implemented in the project notebook.

The first processing step was to convert the Units column from an object to a float. I'm not sure why an ERP system would allow text values to be entered into the units field, which should only be the number of units of an item purchased.

```
#Convert the Units column to float
df['Units'] = pd.to_numeric(df['Units'], errors='coerce').fillna(0)
df['Units'] = df['Units'].astype('float64')
```

I then dropped any PO items with null values in any of the columns. There were only 21 of them, so it didn't make sense to include them.

```
df.dropna(inplace=True)
```

Next, using a master list of valid cost codes exported from the system, I dropped any PO items that had an invalid cost code. This could possibly occur due to incorrect data entry or if a cost code was made invalid and is no longer used in POs.

```
#Read in Master list of valid cost codes
df_ml = pd.read_csv('raw_data/Code_Master_list.csv')
#Drop rows where the cost code is not in the master list
df = df[df['Cost Code'].isin(df_ml['Cost Code'])].dropna()
```

Looking at the numerical fields, there were some negative values in the Units, Unit Cost, and Costs fields. These are likely related to credits back to the company and are not relevant to predicting PO cost codes, so they needed to be removed.

```
#Update dataset to exclude rows with Units, Unit Cost, or Costs that are negative.
df = df[(df[['Units','Unit Cost','Cost']] >= 0).all(axis=1)]
```

Next, looking at the description of the data in the Units, Unit Cost, and Costs fields, we can see that there are a few outliers with very large values that are skewing the data. So by dropping the records in the top 10% of these fields, we get a more representational dataset.

```
#Create a new dataframe that takes only the 90th quartile of data from the 3
numerical columns.
df_90 = df[df['Cost'] < df['Cost'].quantile(.90)]
df_90 = df_90[df_90['Units'] < df_90['Units'].quantile(.90)]
df_90 = df_90[df_90['Unit Cost'] < df_90['Unit Cost'].quantile(.90)]
```

Finally, it is a best practice to scale numerical values between 1 and 0, so I used sklearns MinMaxScaler() to scale the Units, Unit Cost, and Costs features.

```
#It's a good practice to scale numerical data
#Initialize a scaler, then apply it to the features
scaler = MinMaxScaler()
numerical = ['Units','Unit Cost','Cost']
df_90[numerical] = scaler.fit_transform(df_90[numerical])
```

We'll need cost codes with at least 10 examples in the database to have at least one example in both the training and testing datasets and we want enough samples so that the KNeighbors classifier has some data to work with. So drop any codes with a count fewer than 10 samples.

```
#When splitting for training and testing later, we'll need a minimum of 10 examples
of each cost code.
#Assign cost code to a variable
df_count = df_90['Cost Code'].value_counts()
#New dataframe only includes lines with cost codes with a count of 10 or greater
df_90 = df_90[~df_90['Cost Code'].isin(df_count[df_count <= 10].index)]
```

Next, the categorical features Vendor and Unit of Measure need to be dealt with. I'll use one-hot-encoding for these features.

```
#One Hot Encode categorical features
categorical = ['Vendor', 'Unit of Measure']
df_90 = pd.get_dummies(df_90, columns = categorical )
```

The target variable, "Cost Code" needs to be encoded as well. Label encoding makes sense here.

```
#Numerically encode cost codes.
le = LabelEncoder()
cost_code = df_90['Cost Code']
df_90['Cost Code Encoded'] = le.fit_transform(cost_code)
```

Drop features that are irrelevant to the prediction.

```
#drop features I won't be using
df_90 = df_90.drop(['Company #','Purchase Order', 'Item'], axis = 1)
```

Separate the target variable from the features that will be used to predict it.

```
#Separate the target variable from the features
cost_codes = df['Cost Code Encoded']
features = df.drop(['Cost Code','Cost Code Encoded'], axis=1)
```

Now I split the data into the training and test sets using sklearn's train test split. 80% of the data will be in the training set, and 20% will be in the testing set. The stratify parameter will ensure that the same ratio of cost codes will be included in each set. `#Use sklearn train test split to split the data into training and testing sets.`

```
#Testing set is 20% of total dataset size.
#Stratify the data so we don't introduce bias in the sets.
X_train, X_test, y_train, y_test = train_test_split(features,
cost_codes,
test_size = 0.2,
stratify = cost_codes
)
```

Lastly, for use in the two separate models, I need to extract the Description feature from the training and test sets so they can be used separately from the other features.

```
#Split X_train and X_test text Descriptions for use in a separate model.
X_train_desc = X_train['Description'].copy()
X_train = X_train.drop('Description', axis=1)
X_test_desc = X_test['Description'].copy()
X_test = X_test.drop('Description', axis=1)
```

With those steps, the data pre-processing is complete. Further processing and feature extraction of the text data in the description feature will be done in the pipeline described in the implementation section.

Implementation

In general, the implementation of this project was relatively simple once the data processing was completed. There were basically three parts: Train and test a couple of classifiers to predict the Cost Code value based on the PO description text. Then choose the best model, combine its output with the original training set, then test another set of classifiers to predict the cost code based on the first model's prediction, and the additional numerical and categorical data. Finally, the last part was to try using SMOTE

to augment the number of samples in the data and eliminate bias introduced by the imbalanced dataset, which I will discuss in the refinement section.

As previously noted, I began by creating a pipeline for the first classifier I wanted to test; the SGD Classifier The input to this pipeline is the PO description text.

```
SGDC_pipeline = Pipeline([('vect', CountVectorizer()),
('tfidf', TfidfTransformer()),
('clf', SGDClassifier(random_state=42, tol = 1e-3)),
])
```

Then I then vectorize the text, splitting it into terms that are passed to the Tfidf transformer to get the Tfidf values for the terms. Lastly, this data is passed to the classifier for training or prediction.

I also used GridSearchCV to iterate through hyperparameters for all three components of the pipeline.

```
parameters = {
'clf__loss':['hinge','log'],
'clf__penalty':['l1','l2'],
'clf__alpha':[1e-3,1e-4],
'clf__max_iter':[15,20,25],
'vect__ngram_range':[(1,1),(1,2)],
'tfidf__use_idf':[True,False]
}
```

I then configured gridsearch to use the pipeline and the parameter list to find the best combination of hyperparameters based on the weighed F1-Score and executed it.

```
SGDC_CV = GridSearchCV(SGDC_pipeline, parameters, scoring = 'f1_weighted', n_jobs=4,
cv = 5, verbose = 5)
SGDC_CV.fit(X_train_desc, y_train)
```

The next step was to have the trained model predict the values of the testing dataset and print a classification report to get a summary of the accuracy, precision, recall, and f1-score

```
SGDC_y_pred = SGDC_CV.predict(X_test_desc)
print(classification_report(y_test, SGDC_y_pred))
```

I then repeated these steps for the Logistic Regression Classifier, and the Multinomial Naive Bayes Classifier, and chose the algorithm and hyperparameters that produced the highest F1-Score, Precision, and Recall for the next step.

Next, I added the predicted values from the best performing classifier to the training and test set of data that still included the numerical and categorical features.

```
X_train['Desc Pred'] = LR_CV.predict(X_train_desc)
X_test['Desc Pred'] = LR_CV.predict(X_test_desc)
```

The next step was to train and test a couple of algorithms on this new dataset. I chose a Random Forest classifier and K Neighbors classifier to test and again used a gridsearch to find the optimum hyperparameters. The code below was used for the random forest classifier and is very similar to what was used for the KNeighbors classifier.

```
RF_clf = RandomForestClassifier(random_state=42)
parameters = {'max_depth': [10,50,100],
'min_samples_split': [1,2,3],
'min_samples_leaf': [1,2,3],
'n_estimators': [100, 500, 700, 1000]
}
RF_CV = GridSearchCV(RF_clf, parameters, scoring = 'f1_weighted', n_jobs=4, cv = 5,
verbose = 5)
RF_CV.fit(X_train, y_train)
print(classification_report(y_test, RF_y_pred))
```

Refinement

There were several techniques that I used to attempt to refine the solution further. As mentioned previously, I attempted to use multiple combinations of models to achieve the highest performance. And I also used grid searches on each algorithm to find the hyper-parameters that produced the best results with this dataset.

I began by using gridsearch to tune the parameters, using 5 folds for cross-validation, and arrived at the following optimal parameters for the following transforms and classifier.

CountVectorizer

- ngram_range = (1,2)

TfidfVectorizer

- use_idf = True

LogisticRegression

- C = 20
- solver = saga
- max_iter = 100
- tol = 1e-3

Tuning the logistic regression and pipeline hyperparameters increased the weighted F1-score from 0.40 to 0.46 accuracy from 44.6% to 49%

The next refinement step was to tune the hyperparameters of the random forest classifier. The optimal parameters that I found for this dataset are listed below.

RandomForestClassifier

- max_depth = 100
- min_samples_split = 2
- min_samples_leaf = 2
- n_estimators = 700

Tuning the random forest classifier's hyperparameters resulted in a minimal gain, accuracy increased from 51.1% to 52.6% the weighted F1-score remained the same at 0.50 precision and recall increased to 0.57 to 0.53 from 0.51 and 0.51 respectively

The most complex refinement technique that I employed was to attempt using the SMOTE over-sampling technique to minimize the effect of having imbalanced classes.[7]

As noted in the data exploration phase of this project, there are a relative few number of cost codes that are used significantly more than the others. The most used code appeared 5570 times in the dataset, the average code 111 times, and the median was 6.5.

This means there is a very imbalanced dataset, and this can skew the results of a machine learning algorithm and decrease the accuracy of the model.

One method to combat this is to use SMOTE (Synthetic Minority Over-Sampling TEchnique). SMOTE uses a K Nearest neighbors method to synthetically create more examples of the minority classes in the data based on the existing data in the dataset.

After installing the imblearn package and importing SMOTE using the following code:
`from imblearn.over_sampling import SMOTE`
I then created a new enhanced training set that included more samples.

```
sm = SMOTE(random_state=42)
X_train_res, y_train_res = sm.fit_sample(X_train,y_train.ravel())
```
This increased the training data from 22,349 samples to 380,995 and there is now 3318 examples of each code.

I then re-trained the Random Forest and K Neighbors classifiers using the new dataset. Example code:

```
KN_clf_res = KNeighborsClassifier(n_neighbors=10, weights = 'distance', n_jobs = 4)
KN_clf_res.fit(X_train_res, y_train_res)
KN_y_pred_res = KN_clf_res.predict(X_test)
```

```
print(classification_report(y_test, KN_y_pred_res))
```
The results of applying SMOTE were disappointing, all metrics for both models decreased using the SMOTE augmented dataset (a comparison of all the results is visually represented in the results section). Comparing the F1 Score of the model on the training set, versus on the testing set confirmed my suspicion that the model was overfitting. The weighted F1 Score for the K Neighbors algorithm using the SMOTE training set was 0.976, and on the testing set it was only 0.48. I suspect the overfitting is a result of there not being enough samples of some of the classes to create an accurate representation of that class.

IV. Results

Model Evaluation and Validation

The final solution that I found performed the best was the following pipeline and parameters for predicing the cost code of a PO based on the text of the description.

```
pipeline = Pipeline([('vect', CountVectorizer()),
('tfidf', TfidfTransformer()),
('clf', LogisticRegression(random_state=42,multi_class='multinomial')),
])
parameters = {
'clf__C':[20],
'clf__solver':['saga'],
'clf__max_iter':[100],
'clf__tol': [1e-3],
'vect__ngram_range':[(1,2)],
'tfidf__use_idf':[True]
}
```

Then taking that prediction and adding it to the remaining categorical training and test sets, the Random Forest classifier performed best with the following parameters.

```
RF_clf = RandomForestClassifier(random_state=42)
parameters = {'max_depth': [100],
'min_samples_split': [2],
'min_samples_leaf': [2],
'n_estimators': [700]
}
```

Overall, combining the Logistic Regression and Random Forest models increased the F1-score from 0.46 and 0.44 respectively to a combined 0.50.

RF - Random Forest Classifier
KN - K Neighbors Classifier
RF_res - Random Forest Classifier with Smote enhanced Dataset
KN_res - K Neighbors Classifier with Smote enhanced Dataset

Figure 11. Classifier Score Comparison

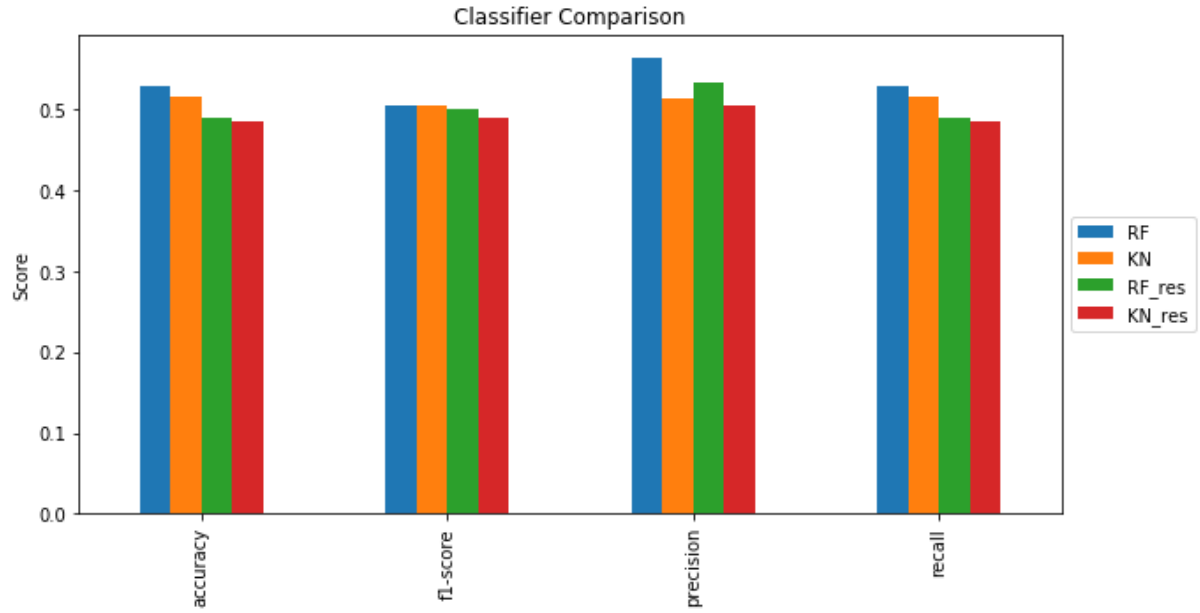


Figure 11 compares the accuracy, F1-score, precision, and recall of the final algorithms that I used in my solution when run against the test set of data. The final solution of using the Logistic Regression algorithms output combined with a Random Forest classifier produced the best results when looking at the accuracy, precision, and recall metrics, and matched using the KNeighbors classifier for F1-Score.

One measure I took to check the robustness of the model was to change the pre-processing code to exclude cost codes with fewer than 2 samples, rather than fewer than 10. This changed the training environment and increased the number of samples available to the model, and increased the number of classes from 114 to 223 but only increased the number of samples from 22,349 to 22,739. This meant the model had to be able to predict an additional 109 classes with only an increase of 390 data samples. The result was actually a negligible increase in accuracy and the same weighted F1-Score. This demonstrates that the model can generalize and does not overfit the existing data.

Justification

Looking at the visualization in the Results section, we can see that the accuracy of all the models performed significantly better than the benchmark model's accuracy of 15%, with the most accurate model having a 52% accuracy rate.

When taking into consideration the nature of the problem, which is to suggest a cost code to an end user who will simply accept or reject the suggestion. I think I can safely argue that my solution to the problem provides enough value to be considered to have solved the problem. There is little cost associated with an incorrect suggestion, and even an incorrect suggestion may provide value to an end user by being close to the recommended code.

In addition, due to the nature of the data available in a PO, there are some instances where there just is not enough information in the PO to more accurately predict the cost code or the correct code to use may be subjective, so expecting a very high accuracy rate is not realistic.

V. Conclusion

Free-Form Visualization

Figure 12 shows some examples of predictions from my model and the actual cost codes.

Figure 12. Prediction Ouput Comparison Table

	Text	Predicted Code	Predicted Code Desc.	Actual Code	Actual Code Desc.
4427	Polarcon Accelerating - Bronze	03-31-41	concrete material - below-grade verticals	03-31-41	concrete material - below-grade verticals
5323	Offix Marker whiteb. chi.ass.	01-52-22	field office supplies	01-52-22	field office supplies
93	306860	01-52-23	field supplies	01-52-23	field supplies
5179	Fuel Surcharge-Per Load	03-31-42	concrete materials - below-grade horizontals	03-31-40	concrete material - footings
933	FREIGHT IN	01-52-23	field supplies	01-54-24	temporary stairway

I think this example demonstrates why getting a high accuracy of the prediction on this dataset is difficult. There are some items and descriptions that could apply to multiple cost codes. For example, the Fuel Surcharge on a concrete delivery would have the same description but could apply to any of several concrete related cost codes. As I mention later in the improvements section, if each line item is taken out of the context of the PO and evaluated by itself, there are instances where there is not enough information to predict which cost code an item belongs to. And again, the "Polarcon Accelerating - Bronze" is a product that is added to concrete to speed its curing time. This product could be used in multiple concrete related cost codes. I think these items demonstrate

that an above 50% accuracy rate for the model is in-fact impressive, and if it does not give the end-user the exact cost code to use, it suggests one that is close.

Reflection

In summary, the solution that I arrived at for this problem involved the following:

- The first part of the solution was pre-processing the training data. Several of the features had extreme outliers in the data and had to be trimmed down, I also scaled the numerical features and converted the categorical features using one-hot-encoding. The data also contained codes that did not appear enough times to make a good prediction, so I dropped these instances. There were also some irrelevant columns that I dropped. Lastly, I label encoded the target variable.

- I then split the training at test data, making sure to stratify the split so that there was a representational portion of all the cost codes in both the training and test set. Since the dataset is very unbalanced, this helps to reduce any bias that could have been introduced if the data as split randomly.

- Next, I extracted the description text and separated it from the numerical and categorical data and use a pipeline that extracts the features from the text data. The last step in the pipeline is training a logistic regression classifier to predict the cost code value based on the text features.

- I then add the predicted cost code from the first model as a feature into the remaining categorical and numerical dataset. I then trained a random forest classifier to predict the final cost code based on this dataset.

As I suspected, I found the most difficult part of this project was figuring out how to deal with text-based data and numerical and categorical data. Most of the resources I found for dealing with text-based data were sentiment analysis based and used only text information. For example, in this paper [8] which addresses a similar problem - classifying products based on their description and other information - they simply treated features that could be categorical, like brand, as text and included it as a word.

Again, the most unexpected result was that the classifiers trained on the dataset that had been augmented using SMOTE performed worse than the classifiers trained on the base dataset. I believe this was due to SMOTE causing overfitting. I noticed that the f1 score of the training set for the classifiers trained on the SMOTE dataset was much higher than the f1 score of the testing dataset indicating overfitting.

When considering this project I was originally hoping to achieve an accuracy score close to 75% and was slightly disappointed to only achieve ~50% accuracy. However, when looking more closely at the data I think this is a good result. Some of the cost codes possibly overlap each other in their use or are confusing and may be frequently miscoded by end-users. One example is the cost codes "01-52-22 field office supplies" and "01-52-23 field supplies" these codes are very similar and possibly misused. So taking this into context I think 50% is a good result, and if you consider the cost of suggesting an incorrect cost code is so low, I think that even at 50% accuracy, the prediction still has value. Lastly, my model significantly beats the benchmark model's accuracy of 15%.

Improvement

There are a few areas where I think I could improve this project:

The first is the way that I handled stacking the two different models. I believe that it is possible to create a pipeline and use feature unions to process the text and numeric and categorical data separately then pass them to a final classifier.[9] By better-using pipeline and stacking functionality, I could improve the accuracy of the model. It would also make testing multiple classifiers easier, and tuning hyperparameters. However, I did not have the time to fully research and understand this technique enough to be confident in implementing it.

The second area that I think could be improved on is the second classifier that I used. My research showed that XGBoost is generally one of the highest performing classifiers. I did manage to get XGBoost working, however, I found that the time it took to run was excessive and I could not properly tune its parameters. I believe the problem with using XGBoost is because of the number of features in the dataset that are created when one-hot-encoding the vendors feature. When I tried label encoding the vendors feature XGBoost ran at an acceptable speed, however all of the classifiers prediction performance dropped unacceptably low. With more time, I would have liked to get XGBoost performing better so I could fully evaluate its performance and possibly increase the accuracy of my model. Another option would have been to try using the LightGBM classifier which is similar to XGBoost and is also supposed to produce great results.[10]

It may also be possible to improve the accuracy of the predictions by taking into consideration other items on a PO. One PO may have several items on it that are related. For example, a concrete purchase order may have 3 items: 1 - the concrete material itself, 2 - a fuel surcharge for the delivery of the concrete, 3 - a disposal fee for leftover concrete. The first item, the concrete material, may be a specific mix that has

information in its description that indicates it is for a concrete footing and, therefore, should be associated with cost code "03-31-40 concrete material footings". The other two items, the fuel surcharge and disposal fee are generic and could apply to any one of many concrete cost codes, however when taken into context with the other item on the PO should also go to the "03-31-40 concrete material footings" cost code.

Overall I'm sure there is room for improvement of my final result, however, this project demonstrated the proof of concept that a machine learning model can use the information in a purchase order to make useful predictions on what the cost code of a purchase order item should be.

References

1 - Olav Eirik Ek Folkestad, E. E. (2017, June). Automatic Classification of Bank Transactions. Retrieved from Norwegian University of Science and Technology Department of Computer Science: https://brage.bibsys.no/xmlui/bitstream/handle/11250/2456871/17699_FULLTEXT.pdf?sequence=1&isAllowed=y

2 - Gorman, B. (2016, December 27) A Kaggler's Guide to Model Stacking in Practice. Retrieved from blog.kaggle.com http://blog.kaggle.com/2016/12/27/a-kagglers-guide-to-model-stacking-in-practice/

3 - Shung, K. P. (2018, March 15). Accuracy, Precision, Recall or F1? Retrieved from Towards Data Science: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

4 - Alencar, R. (2017). Resampling strategies for imbalanced datasets. Retrieved from Kaggle.com: https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets/notebook?scriptVersionId=1745745

5 - Hassan Saif, M. F. (n.d.). On Stopwords, Filtering and Data Sparsity for Sentiment Analysis of Twitter. Retrieved from lrec-conf.org: http://www.lrec-conf.org/proceedings/lrec2014/pdf/292_Paper.pdf

6 - Vectorization, Multinomial Naive Bayes Classifier and Evaluation. (n.d.). Retrieved from ritchieng.com: https://www.ritchieng.com/machine-learning-multinomial-naive-bayes-vectorization/

7 - G. Lemaitre, F. N. (n.d.). Over-Sampling. Retrieved from imbalanced-learn: https://imbalanced-learn.readthedocs.io/en/stable/over_sampling.html

8 - Sushant Shankar, I. L. (2011). Applying Machine Learning to Product Categorization. Department of Computer Science, Stanford University. Retrieved from http://cs229.stanford.edu/proj2011/LinShankar-Applying%20Machine%20Learning%20to%20Product%20Categorization.pdf

9 - Trunov, A. (2017). Work like a Pro with Pipelines and Feature Unions. Retrieved from kaggle.com: https://www.kaggle.com/metadist/work-like-a-pro-with-pipelines-and-feature-unions

10 - Khandelwal, P. (2017, June 12). Which algorithm takes the crown: Light GBM vs XGBOOST? Retrieved from analyticsvidhya.com: https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/