Table of Contents

1. Intro

Working as classification analyst at company_name, I meet on an everyday basis information dedicated to part numbers, part descriptions, HTS codes, failed notes and, of course, audit trails. I use SQL queries to search the existing classifications, similar parts and so on. In other words, I work with a huge amount of text data every day. company_name database has collected gigabytes of data over the years. It would be imprudent not to use such treasury. Some months ago, I started to dive deeper into this topic and now I want to represent a small research dedicated to approaches that can be implemented in our everyday work. The main goal of this research is to show the prospects of the machine algorithms and how they can enhance our job.

2. What is the tariff classification?

Everyday work of classification analysts is pretty routine. Generally speaking to classify part means to assign the appropriate HTS code depending on its properties. Firstly, there is a need to find a part to do. It could be found in the tracker if it is an urgent part, or in the xls-file that gathers work required part numbers from TradeVia database. When part is found the analyst does a research to determine a correct HS code. There are several methods to obtain required information like drawings from client's PLM systems, answers from engineers, US rulings, BTIs and other tools. Sometimes it is very useful to start analysis from searching for the part with similar part number or part description.

Next, when classification is done, part goes to technical advisors. They need to decide if the classification code and audit trail is correct or not. If the classification is incorrect, the analyst receives a failed review and part should be reworked. As analysts describe their logic of choosing a code in the audit trail, technical advisors also write a short failed review that explains why part hadn't passed the review. Technical advisors see HS code, part number, part description and audit trail. They use their experience and deep knowledge of the client's business to decide. It is significant that they work mainly with text data as well as classification analysts.

Mistakes always happen in our work and it is normal. Audit trail can be interpreted not clearly, part can have unusual part number, broker can add wrong part description and there are many other reasons. Of course, it should not be forgotten that analysts as well as technical advisors have a monthly target of part number to be completed. Analysts also have quality target. So there is not much time that can be spent on analysis of each part.

3. How machine learning works?

There is a great definition of machine learning (ML) was defined by Tom Mitchell, an industry pioneer:

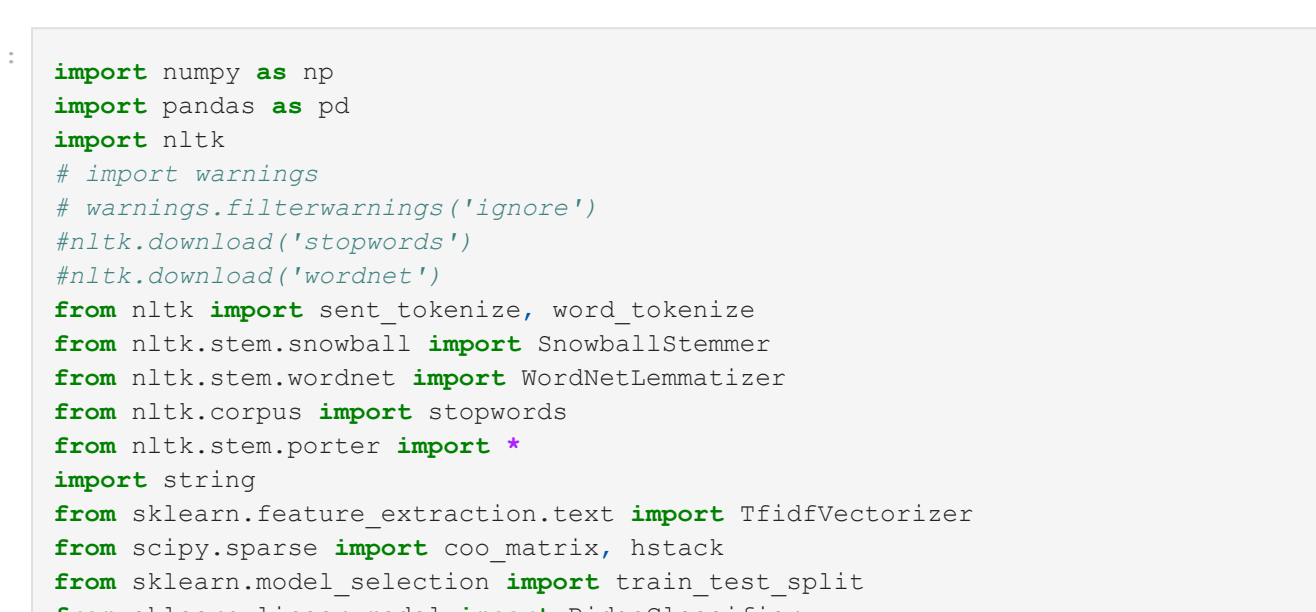
> *Machine learning (ML) is the study of computer algorithms that allow computer programs to automatically improve through experience*

ML algorithms build a model based on sample data in order to make predictions or decisions without being explicitly programmed to do so. We meet these algorithms everyday, they have become an essential part of our life so we don't even notice how often we use them. Virtual Personal Assistants like Siri, personalized recommendations on Netflix, email spam and malware filtering in Gmail, all these technologies are based on ML algorithms. Basic types of algorithms are represented below

Machine learning algorithms use computational methods to "learn" information directly from data without relying on a predetermined equation as a model. They find natural patterns in data that generate insight and help you make better decisions and predictions.

4. How texts can be classified?

One of the big areas of machine learning is classification of texts. It is the process of assigning tags or categories to text according to its content. It's one of the fundamental tasks in natural language processing with broad applications such as sentiment analysis, topic labeling, spam detection, and intent detection. Usual baseline of text classification is shown on the figure.



Firstly, we extract our text data (corpus) from any kind of the source (databases, CRMs, ERPs, social networks and so on). Then text should be preprocessed and usually there is a main part of the work. Corpus need to be cleaned and transformed into a numerical representation in the form of a vector. Next important task is the creation of new features that will help to improve the quality of predictions. When data is ready, then it could be applied various classification models. There are a lot of different mathematical approaches that can solve the problem of classification and there is no universal solution. Various problems require various models and various approaches.

5. How it could be implemented in our business processes?

There are some possible implementations of ML algorithms in company_name business processes. This is a very complex task that requires a lot of research and work. Hence I just want to show the simplest of many possible ways of such implementation.

The easiest and most common type of classification is binary classification. This is the task of classifying the elements of a set into two groups on the basis of a classification rule. During the review technical advisors read the audit trail, part number and part description, check HS code and make a decision if the part was classified correctly (0) or incorrectly (1). ML model does absolutely the same. The main idea is to create such a model that predicts, if the part will receive "work complete" (0) or "failed review" (1) status. It could be integrated in Hive and check the part before submitting. It can be some kind of additional filter or "firewall" against incorrect classifications and will help to simplify work of technical advisors, as well as increase quality of analyst's work.

6. How it works in our case?

Next, it will be shown the building of the simplest base model of binary classification and explained how it works. Before I go further, I want to draw your attention that all computation was done on my working laptop (Intel Core i5-8365U, 16 GB RAM).This imposes certain computational limitations and that's why the model was simplified. Once again, the task is to show the prospect of implementing various ML models in company_name business process not to present state-of-the-art working model.

This model will try to predict if the part will pass the review or not based on its part description, part number, HTS code and audit trail.

Firstly, let's import necessary libraries. I use Python 3.8.3, Pandas 1.1.4 and NLTK 3.5 libraries for pre-processing of the data and scikit-learn 0.23 for building model

In [1]:
```
import numpy as np
import pandas as pd
import nltk
# import warnings
# warnings.filterwarnings('ignore')
#nltk.download('stopwords')
#nltk.download('wordnet')
from nltk import sent_tokenize, word_tokenize
from nltk.stem.snowball import SnowballStemmer
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import stopwords
from nltk.stem.porter import *
import string
from sklearn.feature_extraction.text import TfidfVectorizer
from scipy.sparse import coo_matrix, hstack
from sklearn.model_selection import train_test_split
from sklearn.linear_model import RidgeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score, roc_auc_score, accuracy_score
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from sklearn.model_selection import StratifiedKFold
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
```

Scikit-learn uses random permutations to generate the splits. We need to fix a seed, it could be any natural number. This ensures that the random numbers are generated in the same order and we will receive the same results

In [2]:
```
SEED = 42
```

There are thousands of parts that had already been reviewed in TradeVia database. That makes our task much easier because our data is **labeled**. We know what kind of audit trails had passed the review and what had not. I used only part number from <client's name> business because I'm most familiar with that client. I used not all <client's name> businesses but only the following: <client's businesses>. In future it could be expanded for all clients and all business. Also due to the computational limitations I used limited quantity of parts that passed review. I took them since 2019-01-01 to 2019-06-30 for complete parts and whole 2019 for failed parts because there are fewer of them but our model must be trained on balanced dataset. Also I used parts reviewed by only several TA's because otherwise dataset would be too big. In additional I used parts that have strength ('8', '20', '55', '90', '91', '92', '95', '100'). To create a model that will find regularities in corpuses of audit trails and determine which of them leads to successful or unsuccessful classification, I will combine two parts of passed audit trails and failed audit trails. The appropriate SQL queries can be found here. Firstly let's import data to Python from .csv file and throw out unnecessary columns.

First part of the model will have parts that had passed the review.

In [3]:
```
passed = pd.read_csv('model_passed_review.csv', header = None)
passed = passed.drop([1, 5, 6, 8, 9], axis=1)
passed = passed.rename(columns=
                       {0 : "Business",
                        2 : "Part_number",
                        3 : "Part_description",
                        4 : "HS_code",
                        7 : "Audit_trail"})
passed = passed.drop_duplicates(subset=['Audit_trail']).reset_index(drop=True)
passed_data = passed.fillna('No audit trail')
passed_data['Failed'] = 0
```

Then let's do the same with the parts that hadn't passed the review. Also, I concatenate two datasets into one and split the target variable as **y**

In [4]:
```
failed = pd.read_csv('model_failed_review.csv', header = None)
failed = failed.drop([5, 6, 7, 8], axis=1)
failed = failed.rename(columns=
                       {0 : "Business",
                        1 : "Part_number",
                        2 : "Part_description",
                        3 : "HS_code",
                        4 : "Audit_trail"})
failed = failed.drop_duplicates(subset=['Audit_trail']).reset_index(drop=True)
```

In [5]:
```
failed_data = failed.fillna('No audit trail').reset_index(drop=True)
failed_data['Failed'] = 1
```

In [6]:
```
train_data = pd.concat([passed_data, failed_data])
train_data.groupby(['Failed']).count().Part_number
```

Out[6]:
```
Failed
0    36674
1    26106
Name: Part_number, dtype: int64
```

In [7]:
```
y = train_data['Failed']
train_data = train_data.drop(['Failed'], axis=1)
```

Our data is ready for further work. However the raw data, a sequence of symbols cannot be fed directly to the algorithms themselves as most of them expect numerical feature vectors with a fixed size rather than the raw text documents with variable length. That's why we need to preprocess it and prepare it for building the model. Here I will use the technique called "Bag-of-words". It is one of the most commonly used methods in text classifications and also it is easy to implement. In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity. The frequency of occurrence of each word is used as a feature for training a classifier. The following steps are:

1. reduce all words to lowercase
2. use regular expressions to represent all words and symbols as "tokens"
3. join tokens into one string separated by spaces
4. reduce words to their stem (e.g. "play" from "playing") using stemming algorithms. It also possible to use lemmatization but in our case stemming works slightly better

In [8]:
```
TOKENIZE_RE = re.compile(r'[a-zA-Z]+|-?\d*[-.,]?\d+|\S', re.I)
stop_words = set(stopwords.words("english"))
stemmer = SnowballStemmer("english")
#lemmatizer = WordNetLemmatizer()

def tokenize(txt):
    return TOKENIZE_RE.findall(txt)

def preparing_data(df):
    df = df.astype(str)
    df = df.applymap(lambda x: x.strip().lower())
    df = df.applymap(lambda x: tokenize(x))
    df = df['Business'] + df['Part_number'] + df['Part_description'] + df['Audit_trai
    df = df.apply(" ".join)
    df = df.apply(lambda x : stemmer.stem(x))
    #df = df.apply(lambda x: lemmatizer.lemmatize(x))

    return df
```

In [9]:
```
train_data = preparing_data(train_data)
```

In a large text corpus, some words will be very present hence carrying very little meaningful information about the actual contents of the document. If we were to feed the direct count data directly to a classifier those very frequent terms would shadow the frequencies of rarer yet more interesting terms.

In order to re-weight the count features into floating point values suitable for usage by a classifier it is very common to use the tf–idf transform. We will throw away so-called "stopwords", like "and", "the", "per". Also, here we got two additional thresholds: **max_df** and **min_df**. `max_df = 0.90` means "ignore terms that appear in more than 90% of the texts", `min_df = 4` means "ignore terms that appear in less than 4 texts". It will vectorized our train data into a sparse two-dimensional matrix suitable for feeding into a classifier. `ngram_range=(1, n)` means that it will be used a string of n words in a row. It our case `n = 3`

In [11]:
```
vectorizer = TfidfVectorizer(analyzer='word',
                             stop_words=stop_words,
                             max_df = 0.9,
                             min_df = 4,
                             ngram_range=(1, 3))
```

In [12]:
```
train_data_sparse = vectorizer.fit_transform(train_data)
```

In [13]:
```
len(vectorizer.vocabulary_)
```

Out[13]:
```
186405
```

Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called **overfitting**. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a `holdout` part. Usually it is 20..30% of the whole train dataset.

In [14]:
```
X_train, X_holdout, y_train, y_holdout = train_test_split(train_data_sparse, y,
                                                          test_size=0.3,
                                                          random_state=SEED)
```

Let's try to predict something)

In basic pipeline of text binary classification is usually used firstly a simple logistic regression. It is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function. `clf` will be the first model.

Before going further we need to discuss quality metrics, that would be used

In a binary classification task, the terms "positive" and "negative" refer to the classifier's prediction, and the terms "true" and "false" refer to whether that prediction corresponds to the external judgment (sometimes known as the "observation"). Given these definitions, we can formulate the following scheme:

		Predicted	
		Negative	Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

In our case actual labels were provided by technical advisors, predicted labels were provided by the model. In statistical analysis of binary classification, the F-score or F-measure is a measure of a test's accuracy. It is calculated from the precision and recall of the test, where the precision is the number of correctly identified positive results divided by the number of all positive results, including those not identified correctly, and the recall is the number of correctly identified positive results divided by the number of all samples that should have been identified as positive. The formulas and more detailed explanation could be found here

In our business process there are two factors that impact on choosing of a metric. A) quantity of parts that passed review usually is much higher than failed ones, classification data is imbalanced and B) both results (passed and failed) are important. We want model neither to "fail" correct classification nor to pass the incorrect ones. In that case is better to use F1-metrics. ***The highest possible value of an F-score is 1***, indicating perfect precision and recall, and the lowest possible value is 0, if either the precision or the recall is zero

In [15]:
```
clf = LogisticRegression(random_state=SEED).fit(X_train, y_train)
```

In [16]:
```
print('F1 score (train) %.3f' % f1_score(y_train, clf.predict(X_train)))
print('F1 score (holdout) %.3f' % f1_score(y_holdout, clf.predict(X_holdout)))
```

```
F1 score (train) 0.810
F1 score (holdout) 0.685
```

The F1 score can be interpreted as a weighted average of the precision and recall values. It tells us how precise our classifier is (how many instances it classifies correctly), as well as how robust it is (it does not miss a significant number of instances). The simple logistic regression model predicts labels of part numbers ***precisely*** more than in 65% of all cases. Results on train dataset is better, because it was counted on those data that was used for learning. For our purposes it is much more important results on a "new", holdout data which model hadn't seen before.

Usually logistic regression is a started point of the text classification pipeline. Very often but not always it works better than other basic methods. But let's try another ones. The next model (`clf2`) will use ridge classifier.

In [17]:
```
clf2 = RidgeClassifier(random_state=SEED).fit(X_train, y_train)
```

In [18]:
```
print('F1 score (train) %.3f' % f1_score(y_train, clf2.predict(X_train)))
print('F1 score (holdout) %.3f' % f1_score(y_holdout, clf2.predict(X_holdout)))
```

```
F1 score (train) 0.915
F1 score (holdout) 0.708
```

Looks better on holdout dataset and much better on train data set. But it could be just a result of overfitting.

The next model `clf3` will use method of another class of models. Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

In [19]:
```
clf3 = MultinomialNB().fit(X_train, y_train)
```

In [20]:
```
print('F1 score (train) %.3f' % f1_score(y_train, clf3.predict(X_train)))
print('F1 score (holdout) %.3f' % f1_score(y_holdout, clf3.predict(X_holdout)))
```

```
F1 score (train) 0.738
F1 score (holdout) 0.620
```

After it I tried SVM — Support Vector Machine. In our case I use linear approximation of SVM as it works very slow because of its high complexity. It shows good results on train and holdout data.

In [21]:
```
clf4 = svm.LinearSVC(random_state=SEED).fit(X_train, y_train)
print('F1 score (train) %.3f' % f1_score(y_train, clf4.predict(X_train)))
print('F1 score (holdout) %.3f' % f1_score(y_holdout, clf4.predict(X_holdout)))
```

```
F1 score (train) 0.947
F1 score (holdout) 0.708
```

7. How it performs on test data?

Now let's test our models on test dataset. For that purpose I've extracted part numbers that were classified in first half of 2020 (passed parts) and from January till November (failed ones) and combined them into one dataset. Like in train dataset I used parts of several <client's name> business reviewed by several TAs. As the quantity of parts that passed review is much bigger than failed one I randomly chose 15000 parts. I used it to make a balanced dataset. Otherwise, F1 - score won't show correct result. Actions applied to preprocess data were the same. Code and results are below.

In [22]:
```
test_passed = pd.read_csv('test_passed_review.csv', header = None)
```

In [23]:
```
test_passed = test_passed.drop([1, 5, 6, 8, 9], axis=1)
test_passed = test_passed.rename(columns=
                                 {0 : "Business",
                                  2 : "Part_number",
                                  3 : "Part_description",
                                  4 : "HS_code",
                                  7 : "Audit_trail"})
test_passed = test_passed.drop_duplicates(subset=['Audit_trail'])
test_passed['Failed'] = 0
test_passed_sample = test_passed.sample(n=15000,random_state=SEED)
```

In [24]:
```
test_failed = pd.read_csv('test_failed_review.csv', header = None)
```

In [25]:
```
test_failed = test_failed.drop([4, 5, 6], axis=1)
test_failed = test_failed.rename(columns={0 : "Business",
                                          1 : "Part_number",
                                          2 : "Part_description",
                                          3 : "Audit_trail",
                                          7 : "HS_code"})
test_failed = test_failed.drop_duplicates(subset=['Audit_trail'])
test_failed['Failed'] = 1
```

In [26]:
```
test_data = pd.concat([test_passed_sample, test_failed])
test_data.groupby(['Failed']).count().Part_number
```

Out[26]:
```
Failed
0    15000
1    14316
Name: Part_number, dtype: int64
```

In [27]:
```
test_data = test_data.reset_index(drop=True)
```

In [28]:
```
y_test = test_data['Failed']
test_data = test_data.drop(['Failed'], axis=1)
```

In [29]:
```
test_data = preparing_data(test_data)
test_data_sparse = vectorizer.transform(test_data)
X_test = test_data_sparse
```

In [30]:
```
print('Logistic regression F1 score (test) %.3f' % f1_score(y_test, clf.predict(X_tes
print('Ridge Clasifier F1 score (test) %.3f' % f1_score(y_test, clf2.predict(X_test)))
print('Naive Bayes F1 score (test) %.3f' % f1_score(y_test, clf3.predict(X_test)))
print('Linear SVM F1 score (test) %.3f' % f1_score(y_test, clf4.predict(X_test)))
```

```
Logistic regression F1 score (test) 0.655
Ridge Clasifier F1 score (test) 0.651
Naive Bayes F1 score (test) 0.571
Linear SVM F1 score (test) 0.654
```

8. What is about results?

Logistic regression `clf` showed the best result on test dataframe. This result may seem unsatisfactory. Indeed, approx.. 65% of quality in binary classification is a little bit better than coin flipping. But don't rush to jump to conclusions and bury this idea. Such result has several reasons:

1. Train dataset was too small. The bigger data set the more probability to meet new rare word in vocabulary.
2. Preprocessing was very superficial just to throw out obvious inaccuracies. Some crazy audit trails could be included as well as auto classified or uploaded part numbers. In addition, there are various reasons for failed review. For example it could be a situation when AT is correct, but HS code is wrong. Much more accurate deeper work with test dataset is needed.
3. Ideally is to have equal parts of training dataset made by all technical advisors. This will help model to perform better on new data and help, let's say, "an average understanding" of a correct classification.
4. The simplest methods were used as they don't require a lot of work to implement and are not demanding to hardware.
5. There were absolutely no hyperparameters tuning. Usually it is used GridSearch with cross-validation to find optimal parameters.

On the last point I would like to stay a little longer. Let's do the easiest gridsearch to find optimal C parameter for Logistic regression. New model is called `clf5`

In [31]:
```
folds = 5
skf = StratifiedKFold(n_splits=folds, shuffle=True, random_state=SEED)
params = {'solver' : ['newton-cg', 'lbfgs', 'liblinear'],
          'penalty' : ['l2'],
          'C' : [100, 10, 1.0, 0.1, 0.01]}
grid = GridSearchCV(clf, param_grid=params, scoring='f1', n_jobs=-1, cv=skf).fit(X_tra
```

In [32]:
```
clf5 = grid.best_estimator_
grid.best_params_
```

Out[32]:
```
{'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}
```

In [33]:
```
print('F1 score (train) %.3f' % f1_score(y_train, clf5.predict(X_train)))
print('F1 score (holdout) %.3f' % f1_score(y_holdout, clf5.predict(X_holdout)))
```

```
F1 score (train) 0.946
F1 score (holdout) 0.708
```

...and the result on test data

In [34]:
```
print('F1 score (test) %.3f' % f1_score(y_test, clf5.predict(X_test)))
```

```
F1 score (test) 0.656
```

Results are better on test and holdout datasets and the same on the test one. After tuning it has become more robust. It is good to see how models perform on only passed review and only failed review. Firstly let's take a look on whole passed test data. Also here we will use accuracy metric because F1-score requires both labels. It computes the accuracy, the fraction (default) of correct predictions.

In [35]:
```
test_data_passed = test_passed
y_test = test_data_passed['Failed']
test_data_passed = test_data_passed.drop(['Failed'], axis=1)
test_data_passed = preparing_data(test_data_passed)
test_data_sparse = vectorizer.transform(test_data_passed)
X_test = test_data_sparse
```

In [36]:
```
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf2.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf3.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf4.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf5.predict(X_test)))
```

```
Accuracy (test) 0.682
Accuracy (test) 0.636
Accuracy (test) 0.819
Accuracy (test) 0.621
Accuracy (test) 0.622
```

...and than only failed parts...

In [37]:
```
test_data_failed = test_failed
y_test = test_data_failed['Failed']
test_data_failed = test_data_failed.drop(['Failed'], axis=1)
test_data_failed = preparing_data(test_data_failed)
test_data_sparse = vectorizer.transform(test_data_failed)
X_test = test_data_sparse
```

In [38]:
```
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf2.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf3.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf4.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf5.predict(X_test)))
```

```
Accuracy (test) 0.650
Accuracy (test) 0.664
Accuracy (test) 0.476
Accuracy (test) 0.677
Accuracy (test) 0.680
```

Models work worse on failed review parts. But even ~60% quality can help to increase our work efficiency. It can be interpreted as more than half of failed review won't be received.

I also want to test models on extremely raw data. The last query takes part numbers that still have failed review status in <clinet's name>

In [39]:
```
total_failed = pd.read_csv('test_total_failed.csv', header = None)
```

In [40]:
```
total_failed = total_failed.drop([4, 5, 6], axis=1)
total_failed = total_failed.rename(columns={0 : "Business",
                                            1 : "Part_number",
                                            2 : "Part_description",
                                            3 : "Audit_trail",
                                            7 : "HS_code"})
total_failed = total_failed.drop_duplicates(subset=['Audit_trail'])
total_failed['Failed'] = 1
test_data_passed = total_failed
y_test = test_data_passed['Failed']
test_data_passed = test_data_passed.drop(['Failed'], axis=1)
test_data_passed = preparing_data(test_data_passed)
test_data_sparse = vectorizer.transform(test_data_passed)
X_test = test_data_sparse
```

In [41]:
```
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf2.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf3.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf4.predict(X_test)))
print('Accuracy (test) %.3f' % accuracy_score(y_test, clf5.predict(X_test)))
```

```
Accuracy (test) 0.623
Accuracy (test) 0.692
Accuracy (test) 0.423
Accuracy (test) 0.717
Accuracy (test) 0.710
```

9. What can be improved?

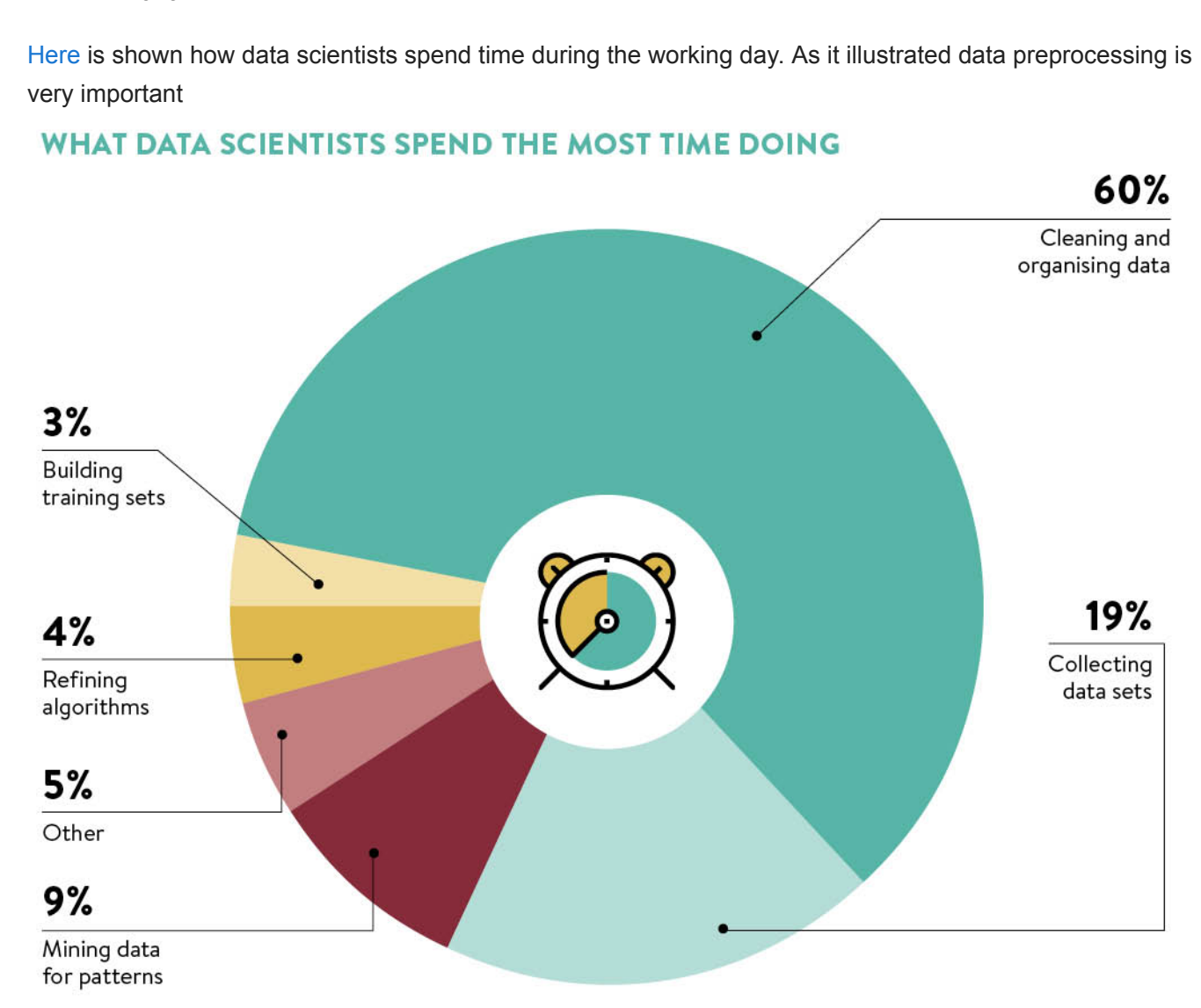
There are 2 main areas to move in to improve quality:

1. ***Choosing of better data for train dataset.*** "Garbage In, Garbage Out" principle is extremely important in ML. The better input data will be used the better results would be received.

   The following steps can be useful:

   - It is good to use only proved and re-checked audit trails. For example, it is possible to find or even to create an ideal audit trail for various often used parts like thermocouples, hoses, screws and etc. The model based on solid trained data will perform better
   - In TradeVia we also have additional data that can be converted to new features, e.g. country name, the presence or absence of ruling or legal notes, remarks and many more other
   - We can use much more data. In this research I used only parts from <client's name> classified in 2019

Here is shown how data scientists spend time during the working day. As it illustrated data preprocessing is very important



2. ***Using of more complex models.*** The models that were presented are very simple. As I mentioned it before, they are used as baseline model. But usually much more advanced models and techniques are used. Neural Networks is a cutting edge in text classification. More information can be found e.g. here. Usually they work better than classical ML algorithms if there are a lot of data can be mined or features in dataset can't be easily created. Also presented model can be improved e.g. hyperparameters tuning can increase quality metric. But due to its computational complexity all this methods require either workstation with GPU or cloud-computing services like AWS or Azure.

10. Conclusion

I tried to represent the basic principles of how machine learning algorithms can be used in company_name business processes. Here it was described how model can predict if the part would received failed review or not. The simplest one but it is only a beginning. In my opinion this field is very perspective because it could reduce costs and increase classification quality as well as clients satisfaction. Of course, it requires a lot of work and investments for developing, testing and implementing to create a working model ready for use but it is a really good starting point to dive into the world of machine learning.