Building Robust and Accurate Transaction Classifiers with Deep Transfer Learning *

Kyle Otstot[1]

[1]School of Computing and Augmented Intelligence, Arizona State University

Abstract

In this work, we develop a solution for the task of transaction categorization, specifically with the dataset provided by the 2022 Wells Fargo Campus Analytics Challenge. Overall, we achieve a few noteworthy contributions, including the engineering of two attributes responsible for improvement in ML performance, development of a word clustering algorithm that helps the practitioner better understand the relationships between words across categories, and design of a high-performing classifier using deep transfer learning and state-of-the-art optimization techniques. The link to the GitHub repository is: `https://github.com/kotstot6/WellsFargoChallenge`

1 Introduction

In today's world of online shopping, it has become especially convenient for modern consumers to not only browse records of their past transactions, but also gain insight into the breakdown of their purchasing habits. In this direction, banking companies like Wells Fargo have fostered advancements in the broadening of their application's functionality, including the development of a *transaction categorization* pipeline to process transaction data in a way that concisely and intuitively represents the purchasing breakdown of account holders. If this banking feature proves to be scalable, then users are immediately granted a powerful built-in budgeting tool. However, the scaling strategy at hand is nontrivial and rather challenging for practitioners to solve.

For the longest time, companies were restricted to the large-scale human labeling of transaction data, which may lead to accurate and reliable categorization, but fails to be done in a quick and efficient manner. Recently, companies have leveraged their massive accumulations of labeled transaction data by training machine learning (ML) models to replicate the task demonstrated by the pairs of data. In doing so, these companies have successfully deployed transaction classifiers that instantaneously do the job that humans once had to perform on their own. On this topic, Wells Fargo, through their 2022 Campus Analytics Challenge, seeks to make progress in the direction of automated, human-like transaction categorization. In their challenge, Wells Fargo provides a dataset of 40,000 labeled transaction examples, prompting students to develop predictive models that effectively classify 10,000 unlabeled examples into 10 purchasing categories– communication services, education, entertainment, finance, health and community services, property and business services, retail trade, services to transport, trade, professional, and personal services, and travel.

In this paper, I discuss the implementation, evaluation, and practical use of my own solution to the challenge. First, we begin exploring the training dataset and make important decisions on attribute selection and transformation. Next, we discuss and outline two engineered features that enhance the predictability of the existing features, specifically demonstrating this "enhancement" through comprehensive experiments on 6 baseline ML algorithms. Then, we pivot to the field of deep transfer learning, where we investigate and fine-tune two of Google's pre-trained transformers, BERT [2] and XLNet [9], achieving a notable improvement in performance compared to the best aforementioned ML algorithms. Lastly, I give a thorough discussion on the performance metric

*Submitted to the 2022 Wells Fargo Campus Analytics Challenge

limitations, and provide insight into how these metrics should be interpreted. Overall, I conclude that my best-performing BERT classifier appears to give more accurate categorizations than even the (synthetic) train set provided by Wells Fargo, which highlights the pressing need for models to be robust in the face of label noise at training time. The following subsection provides the analytic process flow diagram in Figure 1, which illustrates the rest of the paper to come.

1.1 Analytic Process Flow Diagram

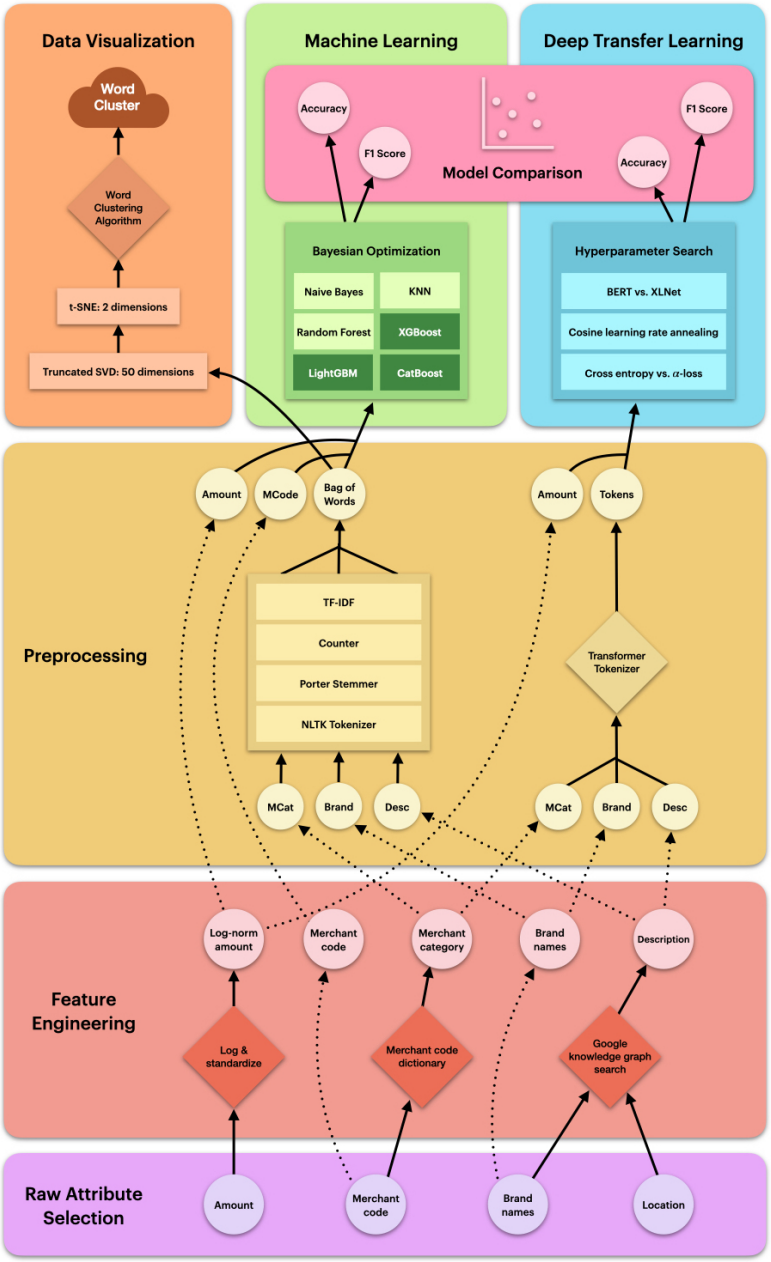

Figure 1: Illustration of the analytic process flow diagram, beginning with attribute selection and ending with model comparison and data visualization.

2 Exploratory Data Analysis

Towards the direction of building an automated transaction classifier, we begin with the traditional and fundamental step of exploratory data analysis to better understand the structure of our dataset. As mentioned earlier, the train set consists of 40,000 rows (examples) of transaction data, all of which are labeled a category (class). On the other hand, the test set consists of 10,000 unlabeled examples. First, we make note of the class distribution provided in Figure 2 below.

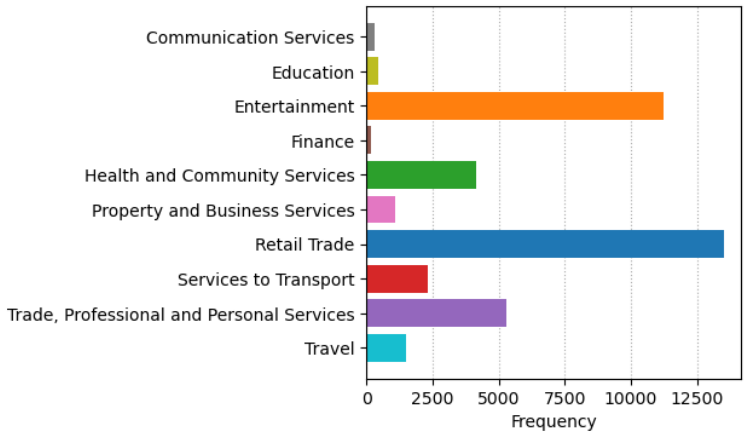

Figure 2: Distribution of transaction categories in the train set.

Here, we clearly see that two categories dominate the train set: *entertainment* and *retail trade*. The former category makes up over 25% of all train examples, while the latter makes up over 30%. Furthermore, some categories such as *communication services* and *finance* are very sparsely represented in the train set, which may make it harder for ML models to learn inner-class representations during training. Having acknowledged a strong degree of class imbalance, it is important for us to ensure that our future models are not greedily returning the majority classes for every guess, regardless of the input semantics. Therefore, along with validation accuracy (the standard metric for classification) we will also consider the *macro F1 score* when evaluating the performance of a classifier. The macro F1 score combines precision and recall, two metrics that are minimized when the false predictions are distributed evenly across the label space.

Moving to the attributes, we count a total of 14. Some consist of strings that identify and describe the transactions, some are flags that indicate the means of payment (e.g., debit card, credit card), and others add more details about the transaction (e.g., amount paid, merchant code). We observe that the *trans_desc* attribute is parsed and processed to form more structured attributes, like *default_brand* and *default_location*. Furthermore, the *default_brand* is sanitized to produce the *qrated_brand* and *coalesced_brand* attributes, which both exist to describe the entity collecting the payment. Therefore, we can disregard all aforementioned attributes except for *default_brand*, *default_location*, and *coalesced_brand*, which will all prove to be very useful in feature engineering. Additionally, we throw away 4 more attributes that exhibit very little diversity in values, which is clearly shown in Figure 10 of the Appendix. This leaves us with two attributes, *amt* and *merchant_cat_code*, to discuss in the upcoming subsections.

2.1 Attribute 1: Amount (*amt*)

The first attribute in consideration is a numerical value representing the amount of money ($) spent in the transaction. In the train set, this value can be as little as 0.01, and as high as ~20,000; in fact, the distribution is so heavily skewed right that it's rather difficult to visualize. Although the dollar unit is easy for us humans to understand, having this degree of asymmetry in the distribution makes it harder for models (*especially* deep learning models) to interpret. Thus, we apply the log

transformation to the attribute, and observe that the distribution is now symmetric as desired. Furthermore, we'd like the distribution to be zero-centered (speeds up convergence in deep learning) so we normalize the log-transformed attribute by applying standardization, which yields the following composition:

$$\text{log-norm-amt} = \frac{\log(\text{amt}) - \text{mean}(\log(\text{amt}))}{\text{std}(\log(\text{amt}))}. \tag{1}$$

As a result, we transform the *amt* attribute into *log-norm-amt*, which is now zero-centered and symmetrically distributed. The transformations for both train and test sets are shown in Figure 3 below. From this, we see that the distributions are practically identical, which gives us more comfort in assuming that the train and test examples are identically distributed. In Figure 11 of the Appendix, we box-plot the log-norm-amount distributions conditioned on each transaction category in the train set. Some diversity is observed, but not too much; nevertheless, we keep the attribute in consideration moving forward.

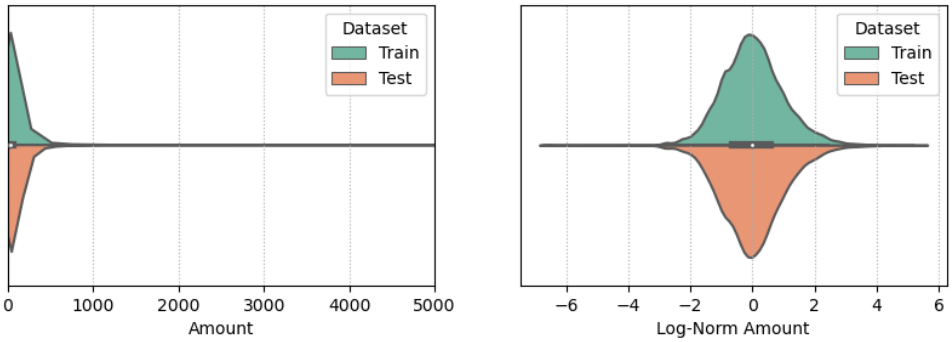

Figure 3: Transforming the amount (*amt*) attribute by applying $\log(\cdot)$ and standardization.

2.2 Attribute 2: Merchant Code (*merchant_cat_code*)

The second attribute is an ordinal value taking form of a code provided by the merchant to help clarify the type of purchase being made; specifically, the merchant code follows the assignment given by ISO-18245. Each code is 4 digits, and the code space is arranged in such a way that neighboring codes tend to reflect similar categories. For example, the range 4000-4799 is related to *transportation* and 7530-7799 is related to *repair services* [1]. To illustrate the organization of the code space, we segmented the space into 10 bins (plus 1 "missing value" bin) and plot the transaction category distribution conditioned on each bin in Figure 4 below.

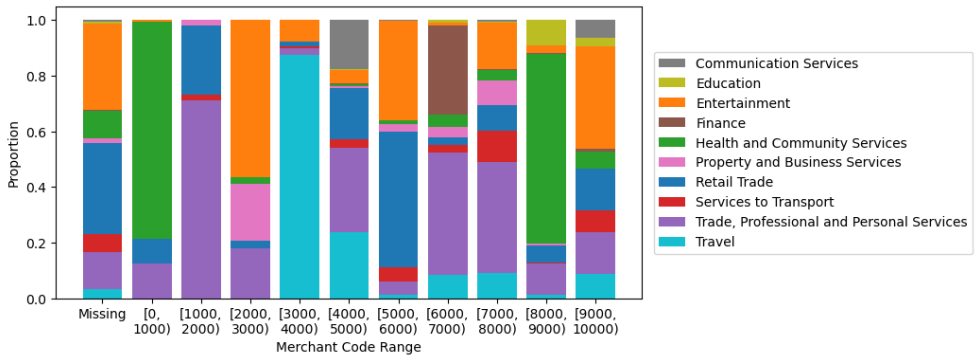

Figure 4: Transaction category distribution organized by merchant code ranges.

[1] More range descriptions can be found at `https://classification.codes/classifications/industry/mcc/`

Looking at this chart, we observe that the transaction category distribution varies across the merchant code ranges. For example, the code ranges $[0, 1000)$ and $[8000, 9000)$ contain a majority of transactions related to *health and community services*, while the range $[6000, 7000)$ contains a relatively high proportion of *finance*. More can be said on this visualization, but the general pattern will be dissected in more detail in the following section on feature engineering. Lastly, in Figure 12 of the Appendix, we confirmed that the train and test sets reflect similar distributions across the merchant code range space, which highlights the general similarity between the two sets.

Lastly, each figure found in this section (as well as the Appendix) was created in Python, primarily with the `matplotlib` and `seaborn` libraries. The code for each figure can be found in the Jupyter notebook file `report_figures.ipynb`.

3 Feature Engineering

Through exploration of the data, we identified several important attributes– namely, the log-norm amount, merchant category code, brand names, and location. As outlined in the previous section, the log-norm amount is a descendant of the original *amt* attribute, and we are content with its current role in the dataset. However, our analysis on the merchant category code revealed a potential pattern that we should look to exploit; this will be detailed further in the next subsection. Moreover, we kept the brand names and locations, but the semantics in these text strings are often not enough to confidently categorize the transaction; in the second subsection, we walk through a web-scraping technique that leverages the brand names and location to extract more information about the transaction. As a result, we will add two engineered attributes to our dataset, and both of which will improve the predictability of our train and test features.

3.1 Engineered Attribute 1: Merchant Category

The first engineered attribute is simply a quantitative description of the merchant code. Luckily, the conversion from merchant code to merchant category is easily implemented with access to existing definitions (see `data/merchant_dictionary.csv`). The reason we add a text description of the merchant category is simple: the assumption that neighboring merchant codes possess similar merchant categories is weak and implicit, while the similarity between merchant categories can be *explicitly* measured. For example, Table 1 includes 3 transactions found in the train set, showing their merchant codes, merchant categories, and transaction categories.

Train index	Merchant code	Merchant category	Transaction category
12947	5541	gas/service stations with/without ancillary services	Services to Transport
12886	5533	automotive parts accessories stores	Retail Trade
15907	7538	automotive service shops (non-dealer)	Services to Transport

Table 1: Three examples of transaction data highlighting a notable difference between merchant code similarity and merchant category similarity.

Here, the first two examples have much closer merchant codes (5541 vs. 5533), but the transaction categories are different. The first example's merchant code may be closer to the second example's code, but its merchant category is more similar to that of the third example, as they both contain the word "service". Both examples reflect a "Services to Transport" transaction category, which may indicate that the word "service" in an automotive context may be the deciding factor between the classification of "Services to Transport" versus "Retail Trade". More examples like this exist, but the point is simply that the addition of a merchant category text description gives us more information to use in the development of our classifier.

3.2 Engineered Attribute 2: Description

The second engineered attribute takes form of a simple text description of the brand names and location, which requires a much more involved collection process. The motivation is straightforward: some brand names like "teriyaki grill" clearly convey the type of expenditure (entertainment), while others like "yoli" and "mlm" do not. Indeed, the addition of a merchant category should go a long way in correctly classifying these transactions, but almost 40% of the data fail to possess a merchant code. The question is: if the code is missing, and the brand name is vague or not category-specific, then how should we go about categorizing the transaction? Often times, this shows to be a major limitation of the raw dataset, especially when the brand name is unforeseen. To address this issue, we turn to web-scraping for more information by using the given brand names and location.

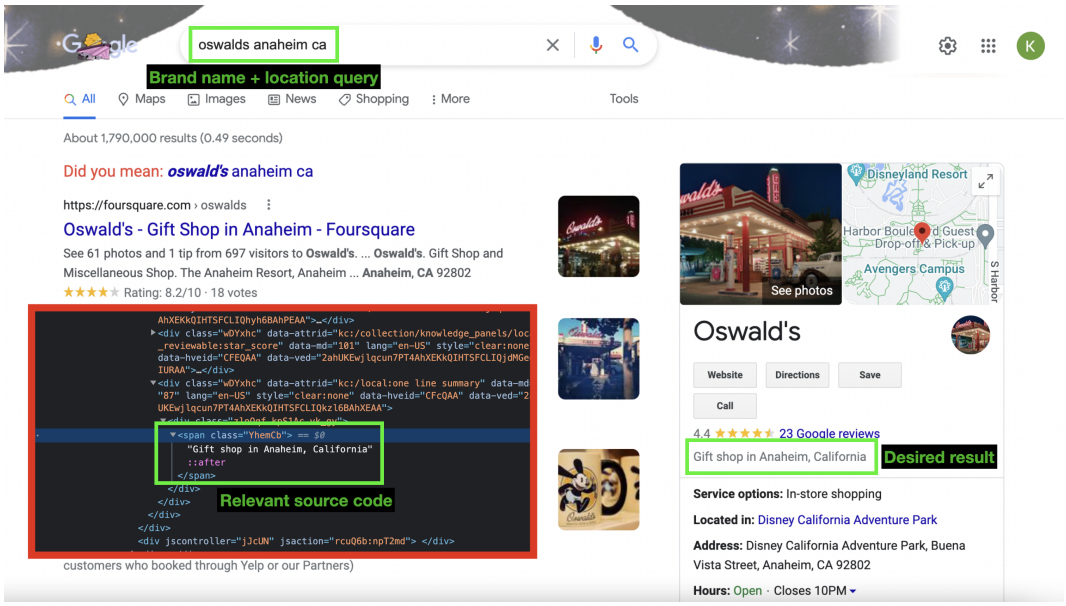

Figure 5: Transaction category distribution organized by merchant code ranges.

Querying Google search is a fair idea, but the amount of unstructured data is rather overwhelming when all we typically need is a short description of the brand. Luckily, in many cases, Google provides such a description in its *knowledge graph*– a panel that is reserved for the most important information related to the query. For example, in Figure 5, the brand "oswalds" with location "anaheim ca" is combined to form a simple query, and the search results are conveniently supplemented with a knowledge graph including the simple description we want– a gift shop. This information helps us (or the classifier) understand that "oswalds" is a gift shop and therefore should be classified as retail trade. Since the source code structure is identical for each returned Google search, it is easy for us to pinpoint where to find the text description. Without going into too many details (see `make_google_search.py` in the `feature_engineering` directory), we are able to supplement each transaction example in the train and test sets with a short description (when Google provides it) by simply querying the brand names and location. We use the `grequests` library for making asynchronous HTTP requests, `requests-ip-rotator` and Amazon Web Services (AWS) for rotating through IP addresses (required for quick web-scraping), and BeautifulSoup (`bs4`) for parsing HTML data. Table 2 gives three more examples of our results along with their transaction categories.

4 Word Clustering

In the previous sections, we have identified and engineered a total of 5 features: *log-norm amount*, *merchant code*, *merchant category*, *brand names*, and *description* (discarding *location* after the fea-

Train index	Brand name	Description	Transaction category
120	ironside	bar	Entertainment
11034	simply mac	electronics store in peachtree corners, georgia	Retail Trade
32471	kosama	fitness center in clive, iowa	Health and Community Services

Table 2: Three examples of transaction data, including the brand name, description via web-scraping Google's knowledge graph, and transaction category.

ture engineering step). To recap, log-norm amount and merchant code are numeric, and the rest are currently unstructured text. In order to feed qualitative attributes to a model, we must strategically encode this data into feature vectors; the process of doing so is called a bag of words (BoW) model, a very common term in the world of natural language processing. In this section, we outline the way in which we convert text to number-filled vectors for both (1) visualization purposes, and (2) our machine learning baseline experiment. After we finish discussing our feature encoding scheme, we will move to visualization techniques that help uncover more information about the task at hand.

4.1 Feature Encoding Pipeline

The goal of this step is to process strings of unstructured text and transform them into interpretable numerical vectors that possess reasonable dimensionality and preserve the meaningful semantics of the original text. Before applying any kind of vectorization, we clean the unstructured text by lowercasing the whole string, removing punctuation, and using `nltk`'s tokenization function to split the text into words. Then, we apply the Porter stemming algorithm [7] to remove suffixes from words, which helps reduce the size of the dictionary. Having done so, we vectorize our qualitative data by introducing a $m \times n$-sized matrix, with m being the number of examples in the dataset, and n being the size of the dictionary (i.e., number of unique words in the dataset). In the matrix (denoted as C), each entry $C_{i,j}$ equals the number of times word j shows up in example i. This vectorization is implemented with Scikit-Learn's `CountVectorizer` class. Then, we transform the matrix further by applying the widely-used *term frequency - inverse document frequency* (TF-IDF) function with Scikit-Learn's `TfidfVectorizer` class. Although we will skip the details of TF-IDF, it is worth noting that the transformation is used as a way to normalize the measure of "word importance" across all dataset examples of varying text length. For an illustration of this pipeline, please refer to the analytic process flow diagram in Figure 1, and for more information on the code implementation, please see `cluster.py` in the `visuals` directory.

Now that we have established the text-to-feature encoding pipeline, we will briefly discuss its use on the three main text attributes. Either each attribute can undergo the vectorization separately and be combined at the end, or be combined at the beginning (i.e., concatenating the text strings) and undergo one vectorization. We will use the latter method for visualization purposes, but we find that the former method performs better as input for machine learning.

4.2 Visualization of the Feature Space

For the purposes of this subsection, we will ignore the *log-norm amount* and *merchant code* attributes since we are strictly dealing with text attributes. Furthermore, we encode the three text attributes by concatenation into one attribute, then transformation into 40,000 rows (one per train example) of vectorized features. As an ML practitioner, it is always helpful to visualize the feature space, but unfortunately the feature vectors possess a massive size of 36,121 (one column for each word in the dictionary). Therefore, we must utilize a dimensionality reduction scheme that best preserves the relationships between feature points in the high-dimensional space.

A recently-introduced technique receiving much attention is *t-distributed stochastic neighbor embedding* (t-SNE) [8], which models each high-dimensional point in lower (2/3) dimensional space in

such a way that synthesizes the similar groups of data and segregates the dissimilar ones. Since the t-SNE algorithm is only recommended for feature spaces of dimension 50 or less, we first use *truncated singular value decomposition* (T-SVD) to reduce the $40,000 \times 36,121$ sparse feature matrix down to $40,000 \times 50$. Figure 6 illustrates the 2D representation of the feature space after applying the t-SNE algorithm to the $40,000 \times 50$ compressed feature matrix. Since this algorithm is unsupervised (receives no labeling), it is exciting to see that the algorithm groups together examples that are similarly categorized. If anything, this visualization reassures that our high-dimensional feature representation contains enough information for our models to make intelligent classifications.

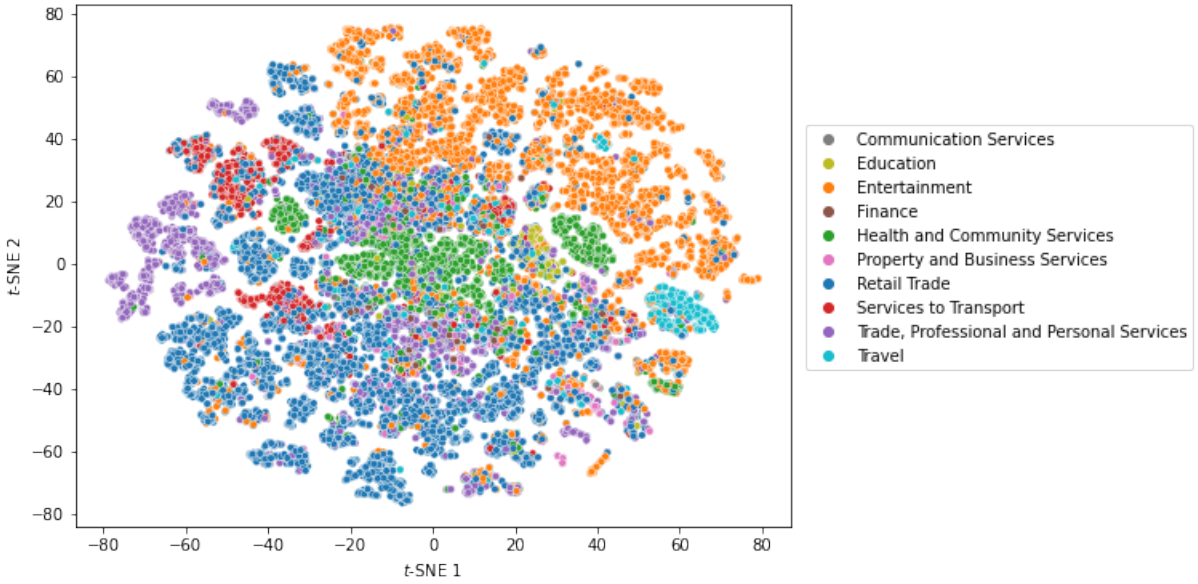

Figure 6: t-SNE clustering of the feature space introduced in subsection 4.1.

Indeed, the clustering of high-dimensional feature data is enlightening in itself, but can we engineer a better visualization by leveraging the useful information we have on word frequencies? What if instead of points, we use words to plot the t-SNE cluster, where each word is written in place of points that contain it, colored according to the category of most relevance, and sized based on its relative frequency within the category? With the data and resources we have, this is certainly possible, so that's why I went ahead and attempted to develop an algorithm that clusters words according to both t-SNE and high-dimensional data. In the next few paragraphs, I will describe my approach as well as my qualitative results.

The first step is to figure out which words in the dictionary belong to which category, and rank them with a score that measures how important they are to the category. In doing so, we use the $40,000 \times 36,121$ high-dimensional TF-IDF feature matrix, denoted T, to create a new $10 \times 36,121$ matrix F where each $F_{i,j}$ measures the importance of word j to the transaction category i. Specifically, we derive F as follows:

$$F_{i,j} = \left[\frac{1}{|K_i|} \sum_{k \in K_i} T_{k,j} \right] - \left[\frac{1}{40,000} \sum_{k=1}^{40,000} T_{k,j} \right], \tag{2}$$

where K_i is the set of examples categorized as i. Then each word is assigned to the transaction category (row) i^* that gives the highest score, where $i^*(j) = \text{argmax}_{i \in [10]} F_{i,j}$. For reference, Table 5 in the Appendix lists the highest-scoring words for each transaction category. Once the words are assigned to categories, the words are normalized between 0 and 1, where the highest-scoring word for each category is 1. Then, the word scores are scaled by category frequency in the dataset. After this step, we have our final scores for each word in the dictionary.

The next step involves the placement of words in the t-SNE cluster. To do this, we build on the `wordcloud` Python library, which randomly fits a group of words (sized by frequency) in a given shape. In their algorithm, they size and position the words in order of frequency (highest $\rightarrow$ lowest); we take the same approach, with our word scores (denoted s_w for word w) acting as the frequencies. However, the main difference is that the existing `wordcloud` algorithm finds all the open placements, then draws the word in a randomly-chosen placement, while we use our own algorithm to decide where the word should be placed, given the open placements. As a prerequisite to our word clustering algorithm, we assign each word a center point; specifically the center point p_w for word w is the dimension-wise *median* of all the 2D points in the t-SNE cluster that (1) belong to the same transaction category assigned to the word, and (2) contain the word in its text. We use median as our measure of centrality because it is more robust to outliers compared to the mean. With each word w possessing a score s_w and central point p_w, we can now introduce the word clustering algorithm demonstrated in Algorithm 1. The global parameters `fontmax`, `fontmin`, α, ϵ_1, and ϵ_2 are used for improving the appearance of the word cluster, with an example displayed in Figure 7.

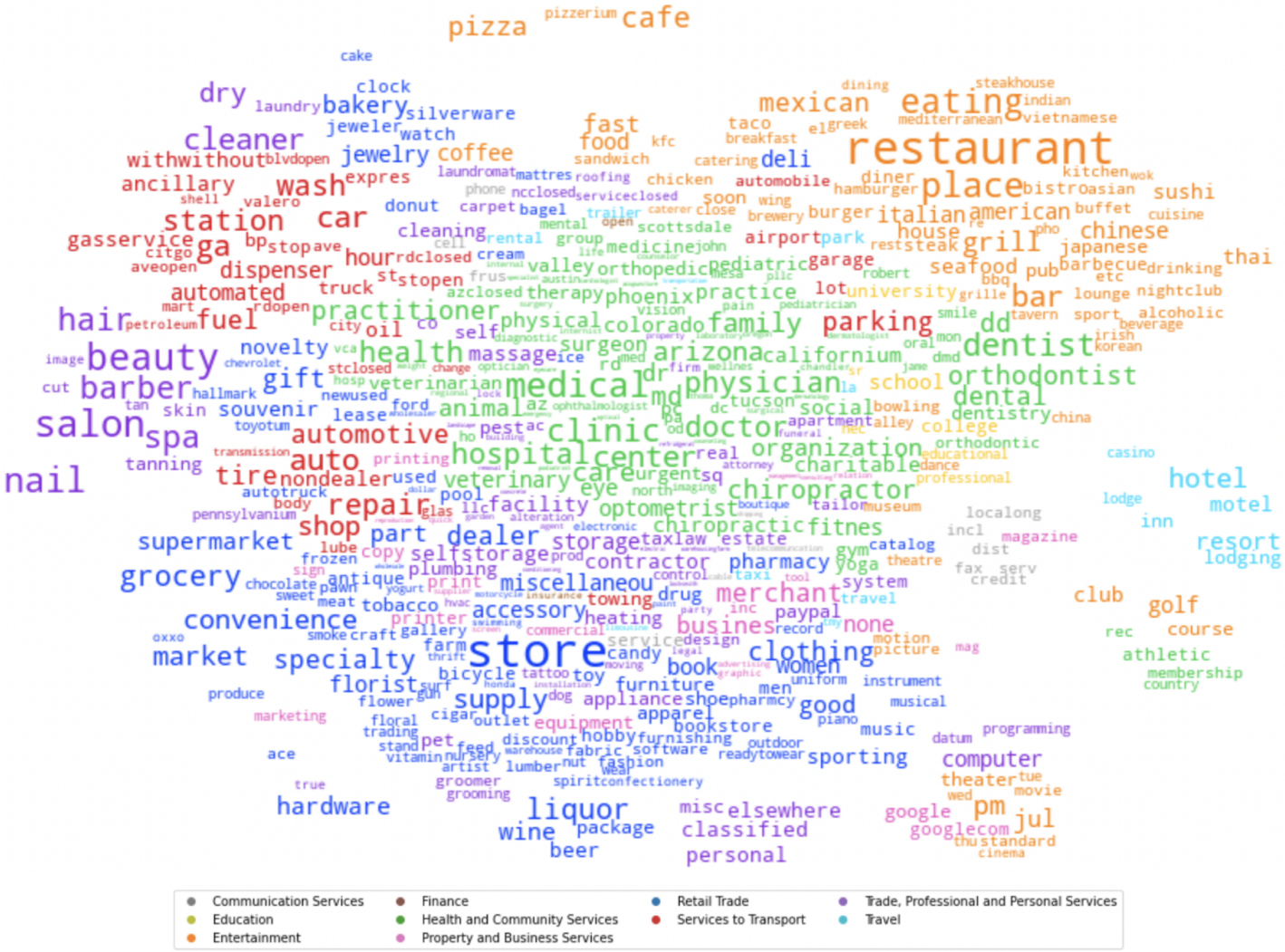

Figure 7: t-SNE **word** clustering of the feature space introduced in subsection 4.1. The parameters `fontmax` $= 30$, `fontmin` $= 4$, $\alpha = 0.25$, $\epsilon_1 = 100$, $\epsilon_2 = 0.25$ were used for this rendering.

Algorithm 1 Positioning words on the "word cloud"

procedure WORDCLUSTER(w, p_w, s_w; `fontmax`, `fontmin` , $\alpha, \epsilon_1, \epsilon_2$)
  w.`fontsize` ← `fontmax` $\times (s_w)^{\alpha}$ ▷ smaller α makes s_w less impactful
  while w.`fontsize` $\geq$ `fontmin` **do**
    `possible` ← open placements within ϵ_1 distance of point p_w
    `preferred` ← places in `possible` also within $\epsilon_1 \cdot \epsilon_2$ distance of p_w
    if `preferred` is nonempty **then**
      return random place in `preferred`
    else if `possible` is nonempty **then**
      return place in `possible` closest to p_w
    end if
    w.`fontsize` ← w.`fontsize` - 1
  end while
  return `null` ▷ no viable placement was found, so word w is not drawn
end procedure

Many more versions can be found in the `wordcloud_figures` directory under `visuals/data/`. For more information on the word scoring and clustering algorithms, please see `word_cluster_data.py` and `make_word_cluster.py` under the `visuals` directory.

To me, the word cluster visualization is quite fascinating and there is a lot to discuss. Since I'd like to keep this short, we will stick to a few main points. First, it is interesting to see how similar words are grouped together within the larger cluster of their transaction category. For example, in the big blue "Retail Trade" cluster, we observe a sub-cluster of *liquor*, *wine*, *beer*, and *package*, which all share the theme of alcoholic beverages; the same theme can be seen in the orange "Entertainment" cluster, where *drinking*, *alcoholic*, *beverage*, and *nightclub* are grouped together. However, what is possibly even more interesting is the overlapping nature between some categories. The red "Services to Transport" cluster is broken into two pieces, which is partially surrounded by automotive-themed "Retail Trade" words, such as *ford*, *lease*, *toyotum*, *used*, *autotruck*, *part*, and *dealer*. This particular overlap underscores the challenge of differentiating between car-related places, like auto parts stores (retail trade) versus auto body shops (services to transport). Lastly, it is also interesting to see how some blue retail words have migrated north, blending in with the orange entertainment words. A *deli*, for example, could very easily be confused with a restaurant, even though it is technically a store selling food. Similarly, *bakery*, *cake*, *donut*, and *bagel* are all technically retail since they are pre-made goods, but they could easily pass as items found in the entertainment category. Although more interesting observations could be made about the word cluster, it is now time to begin discussing the predictive modeling stage of the project.

5 Machine Learning Baselines

Up to this point, we have reviewed the preluding components of the data science life cycle, including attribute selection, feature engineering, and data exploration. In doing so, we witnessed the transformation of three unstructured text attributes into one high-dimensional, sparse feature matrix. On the way here, the intriguing technique of *word clustering* caught our attention, confirming that our data has the potential to facilitate the learning of well-performing models. In this section, we begin exploring the predictive modeling stage, which includes the architectural choices, hyperparameter tuning, and evaluation of machine learning and/or deep learning models. Specifically, we begin with the implementation of 6 total machine learning algorithms– three of which we consider basic, and the other three we consider a strong representative subset of the current boosting algorithm literature.

For each ML algorithm, we use three different datasets. The first dataset only includes three of the original attributes: the *log-norm amount*, *merchant code*, and text of *brand names*. The log-norm amount is stored in one column, while the merchant code takes up two columns– one with the original data (with the missing values imputed with zeros), and the other simply being a missing value indicator column (1 - missing, 0 - filled). The text of brand names is encoded into

features of length 28,918, the size of the brand name dictionary. As a result, the total dataset can be represented by a $40,000 \times 28,921$ feature matrix. The second dataset is similarly structured, but also includes the *merchant category* and *description* text attributes; these are concatenated with the brand names before transforming into 36,121-length feature vectors. The final dataset contains every aforementioned attribute, but keeps the feature encodings separate between the three text attributes. Therefore, the final combined feature vector is the largest of the three, with length 43,079 (includes the 2 numerical attributes, and one dictionary for each text attribute).

ML Algorithm		Amount + Code + Brand		+ Category + Description		+ Separate Bags of Words	
Type	Name	Acc.	F1	Acc.	F1	Acc.	F1
Basic	Naive Bayes	69.7 ± 0.6	53.7 ± 3.7	81.2 ± 0.6	62.9 ± 2.8	82.6 ± 0.7	67.6 ± 3.7
	KNN	75.8 ± 2.3	65.6 ± 3.2	79.7 ± 1.4	69.8 ± 3.6	80.7 ± 1.7	70.0 ± 2.9
	Random Forest	77.8 ± 1.4	60.8 ± 4.1	81.2 ± 0.9	68.7 ± 4.1	82.8 ± 0.4	72.2 ± 3.2
Boosting	XGBoost	78.1 ± 1.4	**72.3 ± 1.7**	82.9 ± 0.8	72.5 ± 3.3	83.0 ± 0.8	**75.0 ± 2.6**
	LightGBM	**78.8 ± 1.9**	70.2 ± 3.1	**83.6 ± 0.9**	**73.1 ± 1.6**	**83.5 ± 0.8**	**75.0 ± 3.1**
	CatBoost	74.1 ± 1.7	57.6 ± 1.8	81.5 ± 1.3	71.2 ± 2.3	82.4 ± 0.8	72.9 ± 2.1

Table 3: Results for the 6 ML algorithms. Each combination of algorithm and dataset was run on seeds 1-5, and the mean ± std is reported. The top-performing results are **boldfaced**.

As mentioned before, we consider a total of 6 ML algorithms. The first algorithm is Multinomial Naive Bayes, which is typically a popular choice for text classification tasks. However, the input must be between 0 and 1, so the Naive Bayes algorithm only trains on the text attribute encodings for each dataset. The second algorithm is K-Nearest Neighbors (KNN) which is typically used for simpler tasks, but can serve to be a good baseline for more complex tasks. The last basic algorithm is Random Forest, which builds an ensemble of decision trees and classifies via majority vote system; this model has more capacity than the previous two, so we expect this one to perform the best out of the basic algorithms. The next three in consideration are boosting algorithms, whose primary objective is to convert a group of weak learners (e.g., decision trees) into strong learners by focusing on their weaknesses– a seemingly-meta approach to machine learning. Specifically, we consider XGBoost [1], LightGBM [4], and CatBoost [3], all of which have recently received high praise for performing well in a variety of ML competitions. Since the performance of ML algorithms often relies on careful choice of hyperparameters, we defined hyperparameter search spaces for each of the 6 algorithms, which can be found in `algo_params.py` under the `machine_learning` directory. Furthermore, we traverse the hyperparameter spaces with Bayesian optimization, implemented with the `hyperopt` library. After a period of optimization, the best hyperparameters are saved for each algorithm, and they are later used with the models to train on 5 random seeds of the experiment.

Overall, we make a few important observations. First, the addition of two engineered attributes– *merchant category* and *description*– certainly helped the models improve their accuracy. In fact, each ML algorithm improved by at least 3% accuracy, with Naive Bayes and CatBoost even improving by 11.5% and 7.4%, respectively. This indicates to us that our feature engineering proved to be worthwhile; the inclusion of merchant category and/or description enhanced the predictability of the dataset to a noticeable extent. Second, it appears that keeping the three feature encodings separate for each text attribute was the right decision in terms of model performance. With the exception of LightGBM (barely), every other algorithm showed improvement with the separation of feature encodings. This would actually make sense because the separation allows the model to distinguish between each attribute, and possibly establish a "hierarchy of trust"; for instance, perhaps the model will find that the merchant category text is more reliable than the description text, or vice-versa. Without this separation, only one "bag" of words is created, and there is no way to differentiate between the three attributes. Lastly, it is clear that XGBoost and LightGBM are the two best-performing algorithms for this task, with LightGBM clearly being the top choice.

The best LightGBM model achieved an accuracy of 83.6%, and another one achieved an F1 score of 75.0%. In the limited world of ML algorithms, these are impressive scores, but we will soon see that deep transformer models can easily surpass these baselines. For more information on the ML baseline experiments, please see `main.py` under the `machine_learning` directory.

6 Deep Transfer Learning

In the previous section, the hyperparameter tuning, training, and evaluation of 6 machine learning algorithms on 3 datasets gave us key insight into the classification task at hand. Not only did it confirm to us the important role merchant category and description play in the separability of the dataset, but it also provided us a pair of baseline metrics to beat: 83.6% for accuracy, and 75.0% for F1 score. Indeed, these metrics are impressive considering the difficulty of the task and relatively-large number of classes, but we still have one important solution to try: deep transfer learning, a method responsible for achieving state-of-the-art results in just about every natural language processing (NLP) task to exist. In general, transfer learning is the process of training a model to perform a certain task, then applying the learned instance of the model to perform a different task. Shifting focus from one task to another, the model is said to "transfer" its embedded knowledge to the new task, helping the model adapt to its new environment rather quickly. In this section, we consider the use of two transformer models, BERT [2] and XLNet [9], that were pre-trained to perform simple NLP tasks, like predicting the next word in a sentence, or evaluating the consistency of two sentences. Now applying the models to the new task of text classification, we, the practitioners, are responsible for *fine-tuning* the models to fit on our own train set. However, it is first important to understand the architecture of each transformer (similaries and differences) before diving deeper into optimization strategy. On this topic, Figure 8 illustrates the setup of each transformer as a text classifier, with BERT on the left and XLNet on the right.

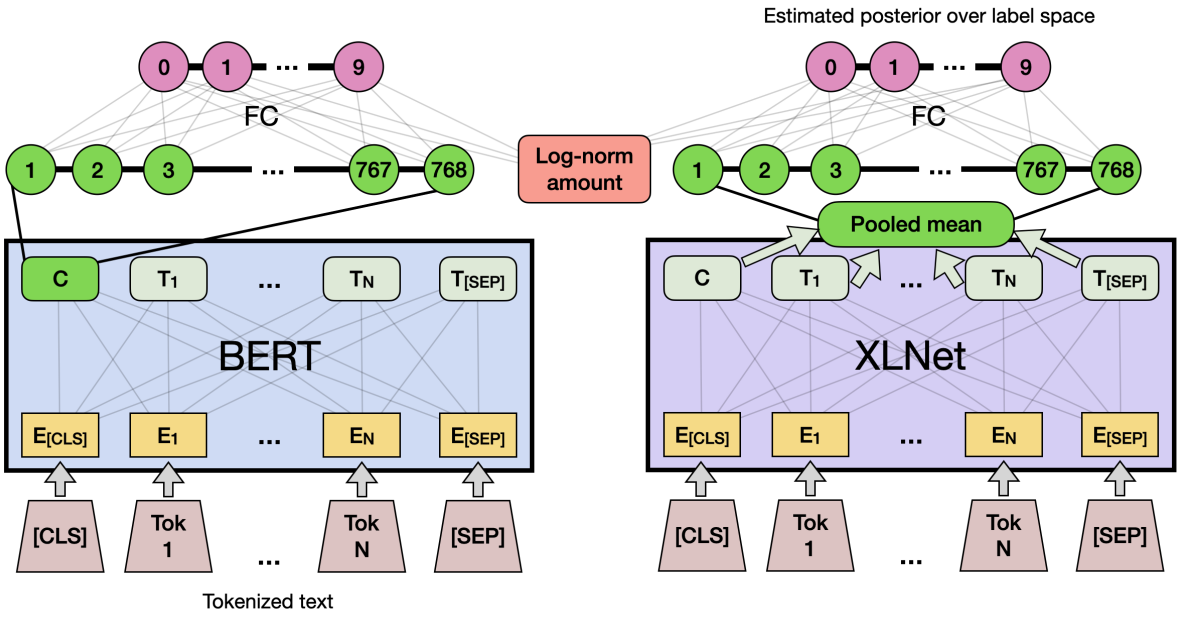

Figure 8: Architectures of each transformer in consideration. Both receive the input similarly, but differ in the fully-connected classification layer.

First, each transformer comes with its own *tokenizer*– a function that converts a string of unstructured text into machine-friendly data. We call these tokens $1...N$, and they are fed into the network to produce N token encodings $E_1...E_N$. Since batches in `pytorch` must contain entries of

the same length, we truncate or pad each input to 50 tokens for both models. After this step, the inner-workings of BERT and XLNet are very structurally different and not the focus of this report; in general, XLNet is a descendant of BERT, and we find that it takes longer to compute due to its larger capacity. Both models return a set of vectorized outputs corresponding to the input length, and from here, BERT and XLNet differ in how they form the first classification state from the output vectors. In one case, BERT simply makes the head of the output layer (C) the first classification state. Conversely, XLNet utilizes the entire set of output vectors by pooling the element-wise mean; since each output vector has a length of 768, the pooled classification state is also of length 768.

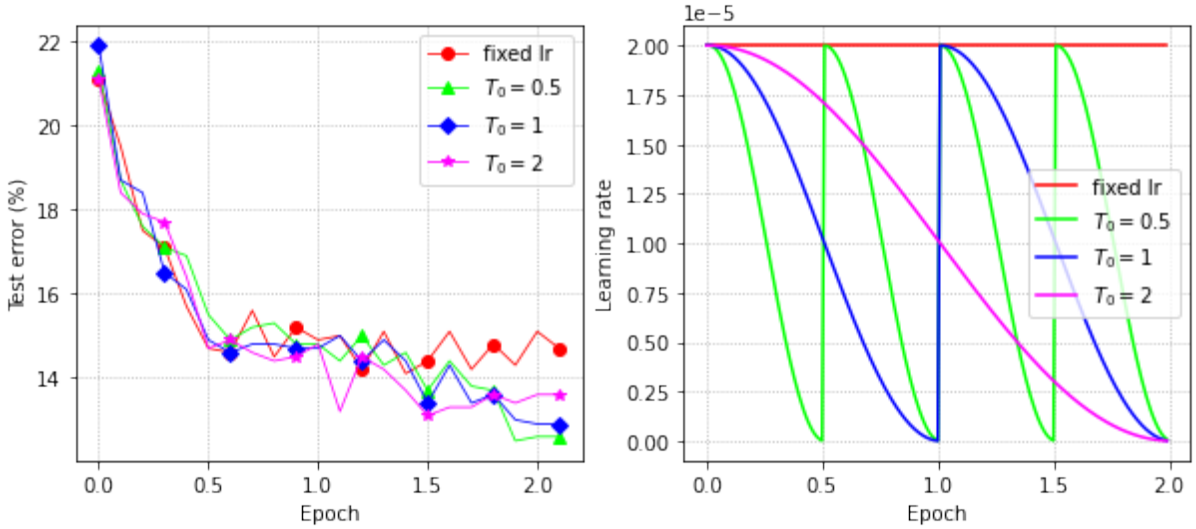

Figure 9: Validation error at each checkpoint during **BERT** training for each measure of T_0 (left) and the realized learning rates during training for each T_0 (right).

Next, the classification state undergoes a fully-connected layer to arrive at the final state of 10 units. Before doing so, we concatenate the *log-norm amount* attribute to the end of each first classification state, yielding a new total length of 769 units. The final classification state represents the estimated posterior distribution $\hat{p}_{Y|X=x}$ conditioned on the input x. Therefore, each unit corresponds to a transaction category; for simplicity, we organize the categories alphabetically, assigning 0 to communication services, 1 to education, and so on. When making predictions, we classify by selecting the category with the highest valued unit. During training, we use dropout with 0.1 probability before the linear classification layer, and cross entropy loss to minimize the expected

	Fixed lr		$T_0 = 0.5$	
Transformer	Acc.	F1	Acc.	F1
BERT	87.3 ± 1.2	78.8 ± 2.7	**87.9 ± 0.6**	**79.0 ± 4.7**
XLNet	86.4 ± 1.1	74.1 ± 2.9	86.8 ± 0.8	76.6 ± 3.2
	$T_0 = 1$		$T_0 = 2$	
Transformer	Acc.	F1	Acc.	F1
BERT	87.7 ± 0.7	**79.0 ± 3.8**	87.6 ± 0.9	78.8 ± 5.0
XLNet	86.7 ± 1.1	76.3 ± 3.6	87.0 ± 1.3	77.0 ± 3.3

Table 4: Results for the two transformers for each interval of warm restart. Each combination of transformer and T_0 was run on seeds 1-3, and the mean ± std is reported. The top-performing results are **boldfaced**

KL Divergence between the ground-truth posterior distribution and estimated poster distribution. For optimization, we use AdamW [6] with an initial learning rate of 2×10^{-5} and weight decay of 0.01. Additionally we use a cosine annealing learning rate scheduler with warm restarts [5], and tune the parameter T_0– the number of iterations before the next warm restart– to maximize the validation accuracy of the models, as shown in Figure 9 for BERT (XLNet is shown by Figure 13 in the Appendix) and Table 4. We train with 2 epochs, which is sufficient for convergence.

Analyzing the results, we see that the most frequent level of warm restarts ($T_0 = 0.5$) achieves the best accuracy and F1 score for BERT– 87.9% and 79.0%, respectively. However, the other values of T_0 are not too far behind, indicating that any degree of learning rate decay serves to be useful in optimization. This is highlighted by the fact that a constant learning rate leads to a noticeably smaller accuracy for both BERT and XLNet. On the other hand, XLNet's accuracy and F1 score is maximized with a cosine annealing decay and no warm restarts ($T_0 = 2$). However, like BERT, XLNet performs well across all tested variations of learning rate decay. Overall, we see a significant improvement in both accuracy and F1 score in the transition from conventional ML algorithms to deep transfer learning; building on this, we also observe a clear difference between BERT and XLNet, where BERT's best instance outperforms XLNet's by 0.9% in accuracy and 2% in F1 score. Therefore, we select BERT with $T_0 = 0.5$ and the aforementioned train settings to be our final classifier. For more information on the deep transfer learning experiments, please see `main.py` and `train.py` under the `deep_learning` directory, as well as the `bert.py` and `xlnet.py` under the `deep_learning/transformer_models/` directory

7 Conclusion

In this work, we develop a solution for the task of transaction categorization, specifically with the dataset provided by the 2022 Wells Fargo Campus Analytics Challenge. Overall, we achieve a few noteworthy contributions, including the engineering of two attributes responsible for improvement in ML performance, development of a word clustering algorithm that helps the practitioner better understand the relationships between words across categories, and design of a high-performing classifier using deep transfer learning and state-of-the-art optimization techniques. The strengths have been identified, but the limitations are just as important. For example, the final classifier file is nearly half a gigabyte, and the training time is significantly higher that that for any of the tested ML algorithms. Furthermore, 88% accuracy appears to be good given the challenging task at hand, but it can always be improved. Unfortunately, the synthetic data contains a noticeable degree of label noise (I estimate around 10%), so the evaluation metrics are not completely accurate. After labeling 200 random examples myself, I observed that my final classifier is about 95% consistent with my answers, while the "ground truth" labeling is only around 90% consistent. Although I didn't have much time to explore this, a potential avenue would be to work on robustification techniques against the threat of label noise. One popular technique is the use of robust loss functions that un-twist the corrupted ground-truth posterior distribution.

References

[1] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[3] Anna Veronika Dorogush, Andrey Gulin, Gleb Gusev, Nikita Kazeev, Liudmila Ostroumova Prokhorenkova, and Aleksandr Vorobev. Fighting biases with dynamic boosting. *CoRR*, abs/1706.09516, 2017.

[4] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *NIPS*, 2017.

[5] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with restarts. *CoRR*, abs/1608.03983, 2016.

[6] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[7] Martin F. Porter. An algorithm for suffix stripping. *Program*, 40:211–218, 1980.

[8] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[9] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019.

A Appendix

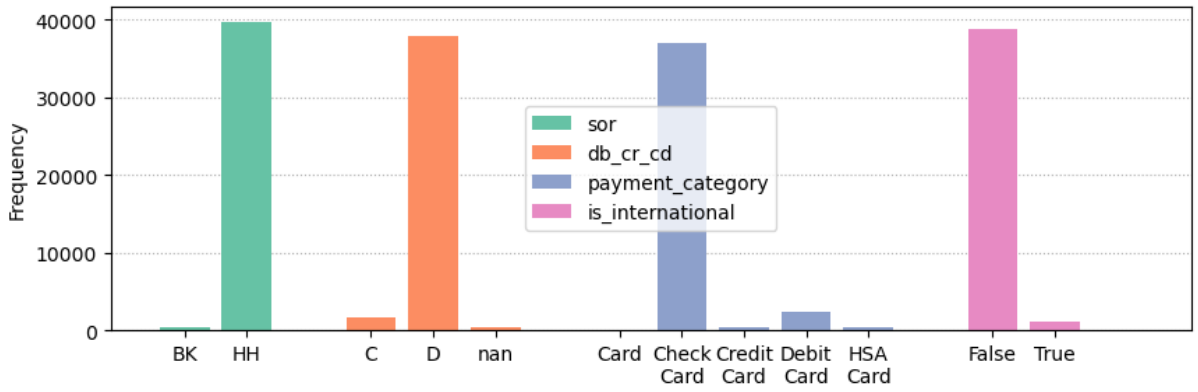

Figure 10: Distributions of four attributes that we chose to discard, reflecting very little diversity.

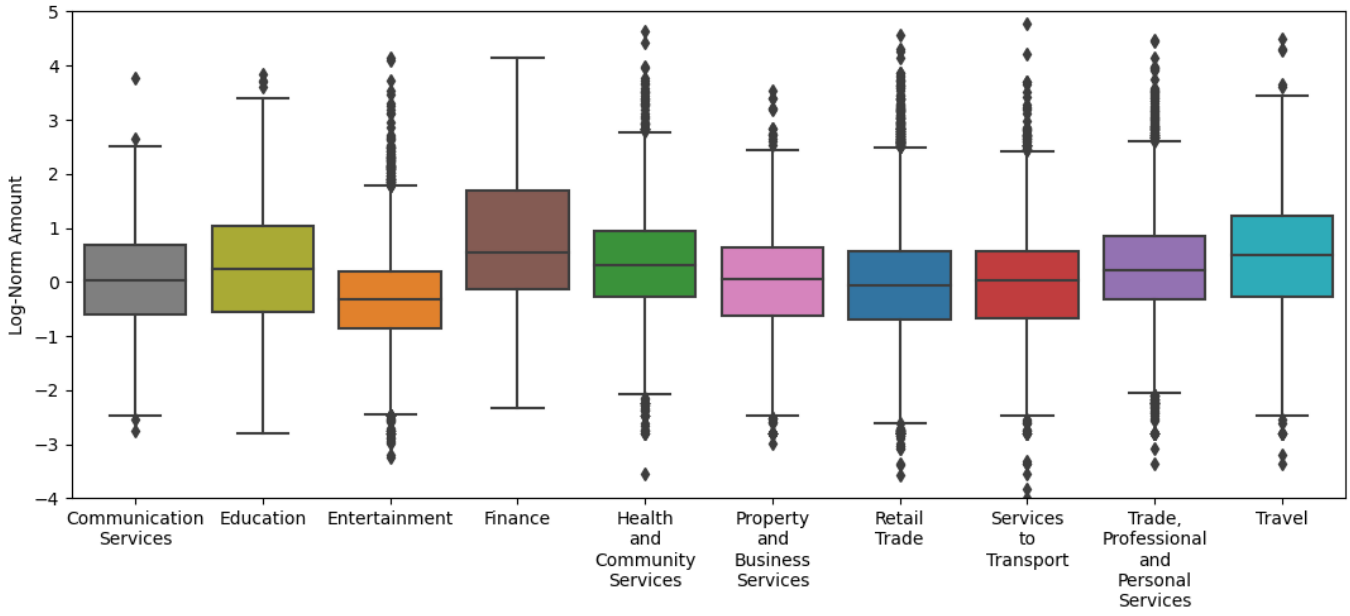

Figure 11: Distributions of the *log-norm amount attribute* conditioned across the category space.

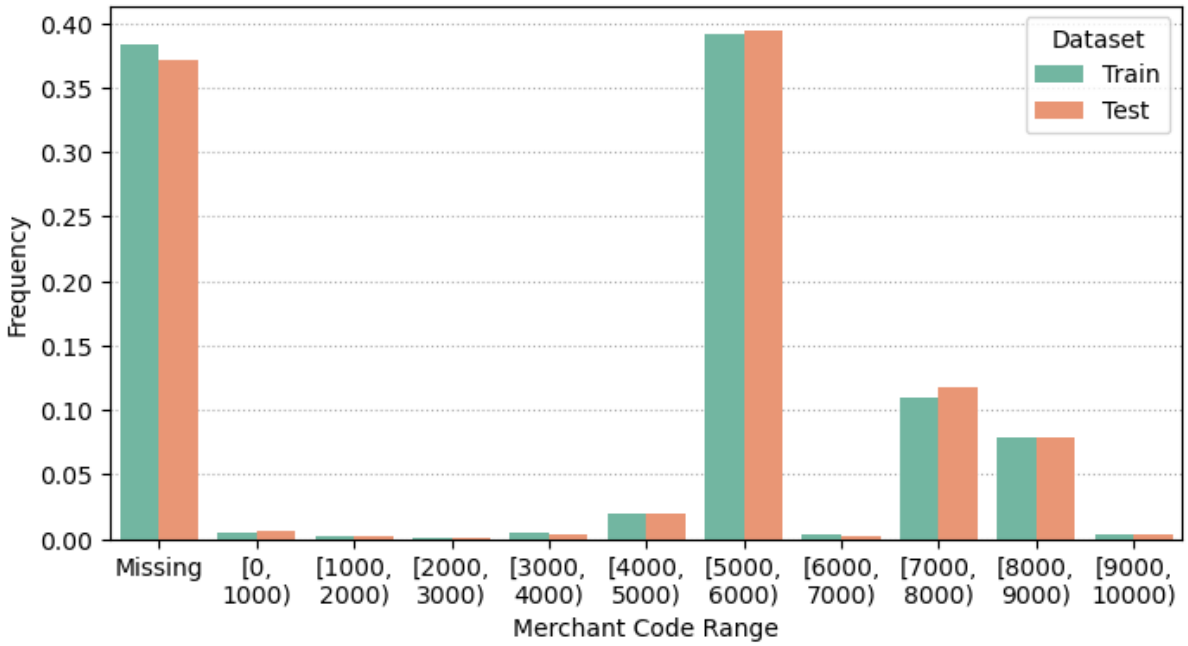

Figure 12: Nearly-identical distributions of merchant code range space for each train and test set.

Transaction Category	Highest-scoring words
Communication Services	service, fri, telecommuncation, serv, localong
Education	school, university, college, educational, nec
Entertainment	restaurant, place, eating, bar, cafe
Finance	insurance, open, agency, financial, near
Health and Community Services	medical, dentist, clinic, center, hospital
Property and Business Services	merchant, continuitysubscription, business, none, google
Retail Trade	store, liquor, grocery, market, convenience
Services to Transport	auto, car, ga, station, repair
Trade, Professional and Personal Services	salon, beauty, nail, barber, hair
Travel	hotel, resort, motel, inn, lodging

Table 5: Highest-scoring words for each of the 10 transaction categories.

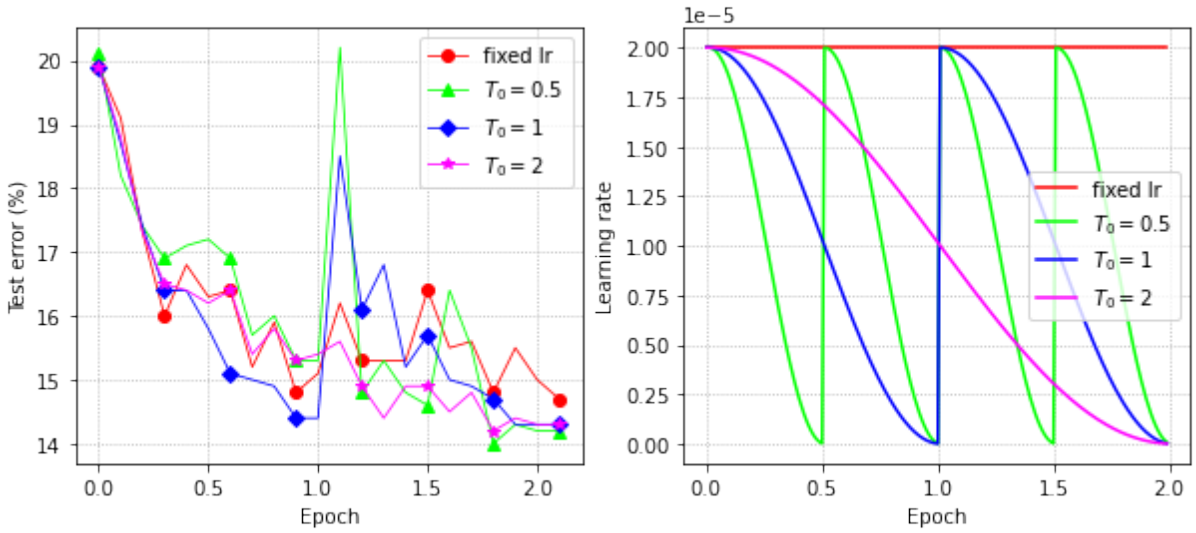

Figure 13: Validation error at each checkpoint during **XLNet** training for each measure of T_0 (left) and the realized learning rates during training for each T_0 (right).