Machine Learning Project: Supervised Learning Models for Classifying Uber and Lyft Trips in the City of Boston, Massachusetts.

Arnoldo Fernando Chue Sánchez
Information Technologies for Sciences
National Autonomous University of Mexico
arnoldwork20@gmail.com

ABSTRACT

This work is the culmination of the 2024 Machine Learning course in the Bachelor's degree in Information Technologies for Sciences at the National Autonomous University of Mexico (UNAM). The course focused on classification algorithms for supervised learning.

Therefore, different models of algorithms studied in class are shown here for classifying Uber and Lyft trips in the city of Boston, Massachusetts.

KEYWORDS

Machile Learning, Supervised Learning, Models, Metrics

ACM Reference Format:
Arnoldo Fernando Chue Sánchez. 2024. Machine Learning Project: Supervised Learning Models for Classifying Uber and Lyft Trips in the City of Boston, Massachusetts.. In *Proceedings of National Autonomous University of Mexico.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/8888888.7777777

1 INTRODUCTION

Companies like Uber, Lyft, Didi, etc., use artificial intelligence all the time. In fact, without machine learning models, these companies would cease to function. However, there aren't many works that combine the data from these companies. This would be very useful because we could identify what truly sets each of these companies apart.

Now, how could supervised machine learning help us in this analysis? There are many approaches that could be taken, but the one presented below is as follows: creating classification algorithms that allow us to classify trips from these companies. If we manage to build models that identify which company is conducting a trip, it would mean that there are indeed differentiation factors for each company.

Therefore, throughout this work, models of supervised learning studied in class that allow us to better achieve the goal of classifying trips between Uber and Lyft will be presented and discussed. These models will range from classic algorithms to ensembles with the algorithms that have performed the best.

National Autonomous University of Mexico, National School of Higher Studies, Morelia Campus
© 2024 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-1234-5/17/07.
https://doi.org/10.1145/8888888.7777777

2 EXPLORATORY DATA ANALYSIS (EDA)

First, let's look at the dataset we'll be working with. The dataset is called "Uber and Lyft Dataset Boston, MA" and can be found at https://www.kaggle.com/datasets/brllrb/uber-and-lyft-dataset-boston-ma. [Kaggle BM [n. d.]]

As we can see, we have two classes, Uber and Lyft. This already shows us that we have a binary classification problem. Furthermore, from the title itself, we can note that the data for these trips was collected from the city of Boston, Massachusetts. Before continuing to describe our dataset, below are all the columns it contains:

- id
- timestamp
- hour
- day
- month
- datetime
- timezone
- source
- destination
- cab_type
- product_id
- name
- price
- distance
- surge_multiplier
- latitude
- longitude
- temperature
- apparentTemperature
- short_summary
- long_summary
- precipIntensity
- precipProbability
- humidity
- windSpeed
- windGust
- windGustTime
- visibility
- temperatureHigh
- temperatureHighTime
- temperatureLow
- temperatureLowTime
- apparentTemperatureHigh
- apparentTemperatureHighTime
- apparentTemperatureLow

- apparentTemperatureLowTime
- icon
- dewPoint
- pressure
- windBearing
- cloudCover
- uvIndex
- visibility.1
- ozone
- sunriseTime
- sunsetTime
- moonPhase
- precipIntensityMax
- uvIndexTime
- temperatureMin
- temperatureMinTime
- temperatureMax
- temperatureMaxTime
- apparentTemperatureMin
- apparentTemperatureMinTime
- apparentTemperatureMax
- apparentTemperatureMaxTime

As we can see initially, we have 57 features with almost 700,000 instances to work with. However, we should not use all the features because some would only add noise to our models.

Analyzing the features, we first find that our class is in what is called cab_type. There, our class is clearly written as Uber or Lyft. Taking this into account, it is necessary to remove the columns product_id and name because they directly reference the class name. This is crucial because if we don't remove them, our models would use them as direct references to predict the classes [Kubat 2017], which is not what we want. Instead, we want the company to be predicted based on the other attributes of the dataset, such as weather variables, trip price, or destination.

Continuing with the classes we will eliminate, there are columns like id or timestamp that would act as unique identifiers for each instance, so they must be removed because they do not provide useful information.

Another unnecessary column is timezone. This is because all values in the column are the same, so it only consumes memory space in the dataset without providing useful information.

The following eliminations might be somewhat controversial, but at an attribute engineering level, I will remove them because I consider them of lesser importance: the time variables associated with a weather factor: windGustTime, temperatureHighTime, temperatureLowTime, apparentTemperatureHighTime, apparentTemperatureLowTime, sunriseTime, sunsetTime, uvIndexTime, temperatureMinTime, temperatureMaxTime, apparentTemperatureMinTime, apparentTemperatureMaxTime.

The reason I don't consider them relevant for our study is because they are in timestamp format, making their interpretation somewhat confusing from the start. However, what I really consider useful and functional for the models are the weather factors. The time associated with them depends a lot on the general time of the trip. So, by keeping attributes about the time the trip was made, we retain that information and avoid repeating it with all the time variables associated with the weather. This way, we can remove them from our dataset.

Continuing with what I mentioned about keeping only the necessary information about the trip time, we have an attribute called datetime. To give it a better format and complete the time information we already have (hour, day, and month), we will extract two new variables from datetime: trip minutes and trip seconds. Once we have these two new columns, we can remove datetime, and we will have the columns we will work with.

Now we can move on to data imputation for missing values. In this case, only the price attribute has missing data, and there are not many. Therefore, I will replace them with their median. The reason I choose the median to replace missing data is to avoid altering the symmetry of the data distribution [Mendenhall et al. 2010]

Now, regarding attributes whose variables are qualitative, we need to map them to numerical values because of our machine learning library (Scikit Learn) [scikit-learn developers [n. d.]a]. However, if we directly assign a number to each of the values of the variables, we would be assigning them an order (even if they do not necessarily have one). In fact, the only attribute we can apply this kind of encoding to is our class attribute.

The cab_type attribute has Lyft and Uber as variables, which, when applied a LabelEncoder [scikit-learn developers [n. d.]e], are transformed into 0 for Lyft and 1 for Uber. Upon further analysis of how our classes are structured, we can see that approximately 55% of the total instances are from Uber and 45% from Lyft. So, they are almost balanced, or at least enough so that class balance is not a major concern.

After handling the columns of our classes, we found that columns like source, destination, short_summary, long_summary, and icon need to be encoded. However, this encoding should not imply establishing an order. Therefore, we will apply a OneHotEncoder [scikit-learn developers [n. d.]h] to these attributes.

By doing the aforementioned, although we remove these 5 columns, we are generating 50 new columns of 0s and 1s. Therefore, we end up with more than 90 attributes, and half of them would behave like a sparse matrix. [Bramer 2016]

With this, we have reached a final numerical dataset with which we could already train some models. For example, algorithms that work with decision trees or variants of Naive Bayes could already be used. However, other models like support vector machines or logistic regression require our data to be normalized or standardized. Therefore, even though some columns are already normalized, we will create an additional dataset that is standardized so that we can use it with the models that require it. [scikit-learn developers [n. d.]i]

Finally, we can create the training and testing sets for both datasets (numerical and standardized) [scikit-learn developers [n. d.]j]. Due to the enormous number of instances we have, I decided to allocate 80% of the data for training and 20% for testing. Even with 20% in testing, we still have a large number of instances that allow us to measure the model's performance.

With this data exploration and preprocessing [VanderPlas 2017] procedure completed, we can begin with machine learning engineering.

3 MODEL DEPLOYMENT

3.1 K-Nearest Neighbors

In the process of selecting the models we want to use, we often think about which algorithm to choose, but sometimes it's just as useful to think about which algorithm not to choose. With this mindset, I began working and quickly decided to discard the K-Nearest Neighbors (KNN) algorithm [Aggarwal 2015]. The reason is due to the number of attributes in our dataset: more than 90. With these numbers, the problem of distances in high dimensions becomes a serious issue because there is no way to differentiate them. If, despite this, you want to work with a KNN model, it might be convenient to use the cosine distance metric. This way, we avoid measuring distances in Euclidean geometries (avoiding the problem of dimensionality) and also avoid penalizing instances with attributes at zero. This last point is crucial considering that with the OneHotEncoder we have 50 attributes with 0 and 1. However, for the purposes of this work, I will not use KNN.

3.2 Decision Tree

In this way, we move on to a new candidate model: decision trees. [scikit-learn developers [n. d.]b] Decision trees are very versatile classifiers: they are not greatly affected by the number of dimensions, their interpretability is straightforward, data standardization is not required, etc. Therefore, we will use our dataset without standardization for a decision tree. However, it is important to consider the main problem with decision trees: it is very easy for them to overfit if we leave the parameters at their defaults, which can lead to pure leaf nodes during training that may cause variance error during testing.

Thus, we need to propose some parameters for tree construction. [scikit-learn developers [n. d.]c] Firstly, the metric for node purity. Here, choosing between entropy and Gini impurity, I opted for the Gini index simply by random choice.

Now, we need to focus on parameters that help us avoid overfitting. Firstly, I decided to set the maximum depth of the tree. This is one of the most useful parameters for manipulating model performance. In this case, knowing that we have more than 90 attributes that can be used to build the tree, setting the maximum depth to 20 is not excessive. Especially considering that for binary classification, the branching factor of the tree is 2. Therefore, giving it room to split nodes 20 times is not excessive.

Another parameter I want to customize is the minimum number of instances required to form a leaf. This is extremely useful to avoid leaves with few instances that cause overfitting. Additionally, this parameter works very well if the maximum depth fails. This is because the tree's depth cannot increase if the number of instances to form new leaves does not allow it. So, I will set the minimum instances to form leaves at 10. With so many instances in the dataset, nodes with 10 leaves should exhibit very high purity if the separation was done correctly throughout the tree.

By building the decision tree in this manner - using the Gini index, maximum depth of 20, and a minimum number of instances to form a leaf of 10 - we ended up with a tree of depth 20 and a total of 3822 leaves. As we can see in 1 he tree turned out to be so large that it's quite challenging to visualize.

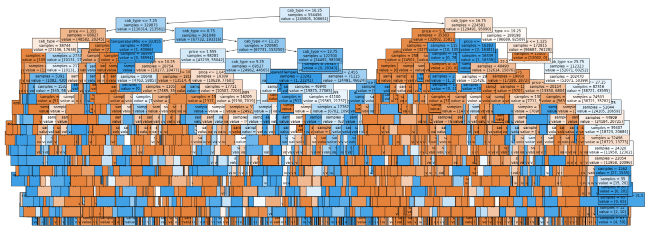

Figure 1: Plotting the result form of the Decision Tree after the training

However, its performance on all metrics with the test set was simply impressive 1:

Table 1: Metrics of the Decision Tree with the test set

Metric	Result
Accuracy	94.60%
Recall	94.58%
Precision	95.65%
F1	95.11%

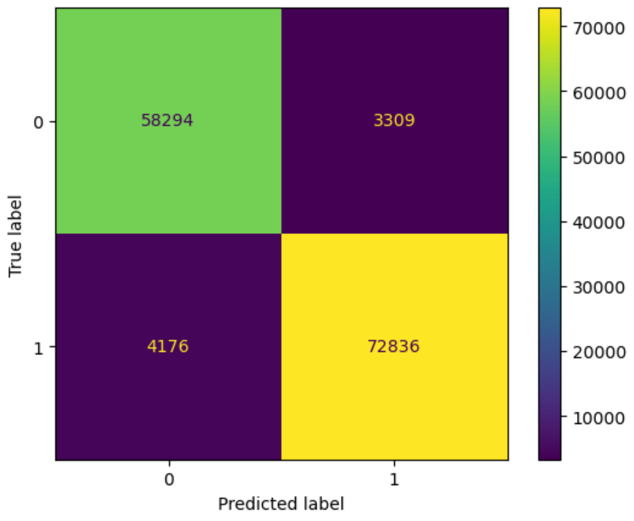

Figure 2: Confusion Matrix of Decision Tree Model

At a results level, we can see in 2 that it only misclassified 4176 Uber as Lyft and 3309 Lyft as Uber. If we focus on its successes, it correctly classified more than 58,000 Lyft and 72,000 Uber.

The important thing here is not only that it achieved metrics close to 95% in all aspects 1. What is truly impressive is that it did so with the test set. If it were directly with the training set, excellent performance would be expected, but a drop would likely occur when changing the dataset. However, our results were with the testing set.

However, it's important to consider that our data covers only two months of 2018. Probably, if we were to put this model into production with current data, it would indeed experience a drop in its metrics, highlighting the overfitting we currently have.

Therefore, we will focus the development of the following models to accompany this decision tree with an ensemble. In other words, to create a model closer to reality, we cannot rely solely on our nearly perfect tree. This tree likely has variance error, but we cannot detect it with the current data. With this in mind, we have an objective: to create an ensemble of models that includes this tree but also prevents or protects it from overfitting. An ensemble is exactly what we need to manage biases between models.

3.3 Naive Bayes

For the next model, I'll opt for a Naive Bayes classifier. [scikit-learn developers [n. d.]g] This is solely to leverage the non-standardized dataset we used for training and testing the decision tree. Therefore, the variant of Naive Bayes we need is Gaussian [scikit-learn developers [n. d.]d]. This is because we not only have categorical variables but also some variables (especially meteorological ones) that are in floating-point numbers. Hence, Gaussian Naive Bayes is a good choice.

When building this model (in this case, it wasn't necessary to modify its parameters), we found some interesting results: 2

Table 2: Metrics of the Gaussian Naive Bayes Classifier with the test set

Metric	Result
Accuracy	58.72%
Recall	99.97%
Precision	57.42%
F1	72.95%

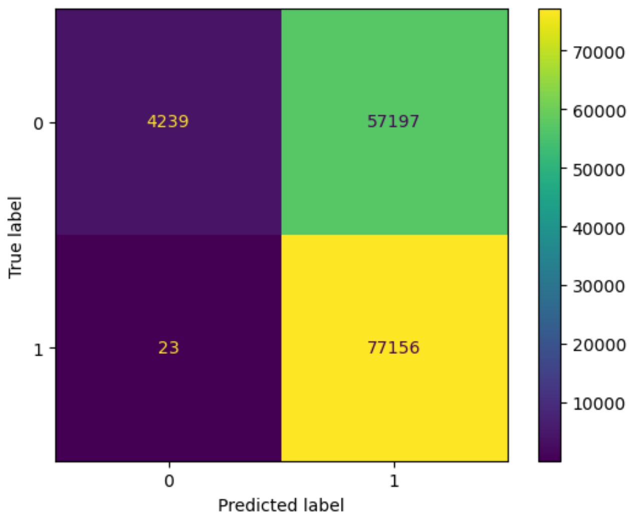

Figure 3: Confusion Matrix of Gaussian Naive Bayes Model

The first thing we should look at in the metrics 2 is that in terms of accuracy, it's a very poor classifier, just barely above random chance. However, what increases significantly is its recall: 99.97%. To interpret this, we have to consider that Uber is the class we consider as true (1). Therefore, we have a model that is unlikely to miss any Uber rides.

As we can see in 3, what harms the model are the more than 57,000 Lyft trips misclassified as Uber. This type I error (too many false positives) causes this classifier to suffer from bias error. Therefore, for the selection of the third model for the ensemble, we need a model that increases precision (at least classifies the Lyft trips correctly).

3.4 Logistic Regression

Continuing with the search for models for the ensemble, we then tried logistic regression [scikit-learn developers [n. d.]f]. Thinking about what we need to complement the ensemble, in addition to a model that classifies Lyft trips well, we need a model that doesn't overfit, meaning one of the main goals of this logistic regression is to learn from the data in a general way, not to specialize excessively.

Therefore, with this in mind, we proposed the following parameters for the regression. First, we will use L1 (Lasso) regularization with a strong regularization parameter C: 0.1. This is to avoid overfitting. By selecting L1 as our regularization, we need to change the default optimization algorithm to saga. Additionally, for the issue of assigning weights to the classes, we set them to the exact proportion in which they are found in the dataset: Uber 55% and Lyft 45%. Finally, we increased the maximum iterations value to 1000 to prevent the model from stopping training if it has not converged to a result.

After completing the training and testing the model with the test data, we obtained the following results: 3

Table 3: Metrics of the Logistic Regression with the test set

Metric	Result
Accuracy	58.74%
Recall	99.95%
Precision	57.48%
F1	72.99%

As we can see in 4 and 3, we have practically the same results as with the Gaussian Naive Bayes classifier. We could adjust the class weights during training to try to rescue the model. However, I prefer to go directly for a more robust model that complements the ensemble better: a Support Vector Machine.

3.5 Support Vector Machine

For the support vector machine, I first decided to use one with a linear kernel, specifically the one available in Scikit-Learn as LinearSVC. This choice was made for training speed. While other models might achieve better accuracy, they would take much longer to train due to the kernel. Therefore, we prefer to use a linear one and manipulate multiple dimensions until our data is well classified.

Support vector machines are not affected by the high number of dimensions in our dataset. However, we do need to use standardized data to achieve convergence of the SVM hyperplane during training.

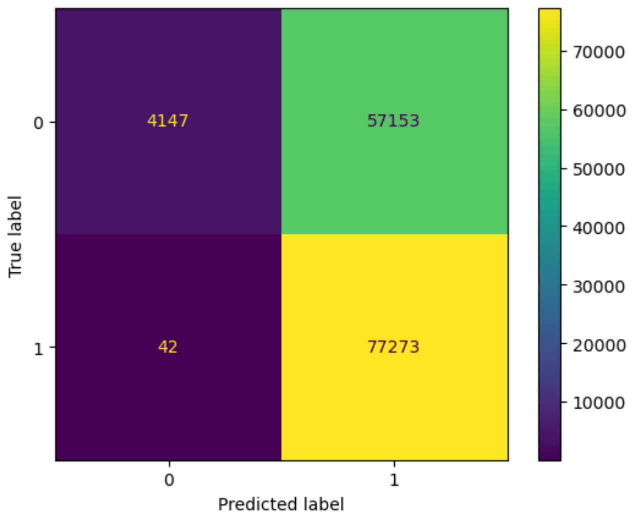

Figure 4: Confusion Matrix of Logistic Regression Model

Changing the focus a bit from logistic regression, we will now use L2 penalty (Ridge) to minimize but not eliminate attributes that generate noise. However, we will maintain the same regularization strength: C=0.1. Regarding class weights (a key component to reduce the marked Type I error), we will use balanced weights.

With these parameters and this new algorithm, we obtained the following results with the test set: 4

Table 4: Metrics of the first SVM

Metric	Result
Accuracy	58.30%
Recall	75.22%
Precision	60.08%
F1	66.80%

As we can see from both the table 4 and the image 5, we managed to increase precision slightly. However, we still have a quite pronounced Type I error, and now we also have an increased Type II error: we have many false negatives, meaning many Ubers are misclassified as Lyfts.

Here, it's clear that we need to improve one thing: our model must correctly identify which instances are Lyfts. In other words, we need the model to fit the data more closely because in this model, there is bias error.

Therefore, I will increase the parameter C to 1. This will reduce the regularization strength so that our SVM learns better. Additionally, we will again adjust the class weights. In this case, we will give more weight to Lyft at 60% and 40% to Uber to improve classification.

With these parameters, we obtained the following results: 5

If we look solely at the results in the table 5, metric-wise, the model is a disaster. However, examining the image 6, we see that we finally managed to reduce false positives with a model other than the decision tree. The problem now is that we have too many false negatives. While we now have issues with Type II error, we can incorporate this support vector machine into the ensemble.

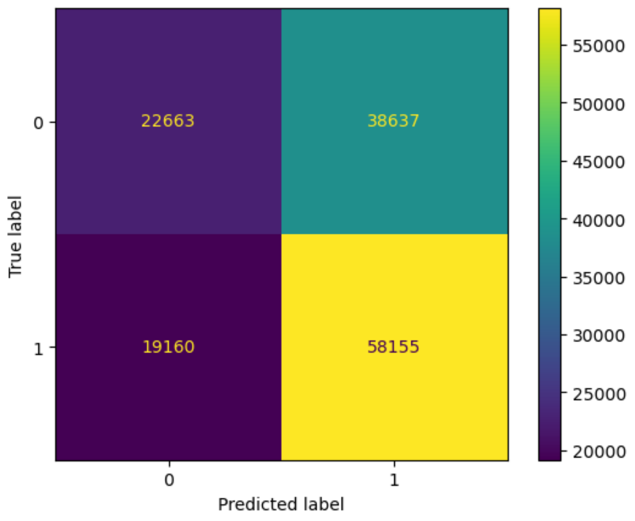

Figure 5: Confusion Matrix of the first SVM

Table 5: Metrics of the second SVM

Metric	Result
Accuracy	48.78%
Recall	26.47%
Precision	59.11%
F1	36.57%

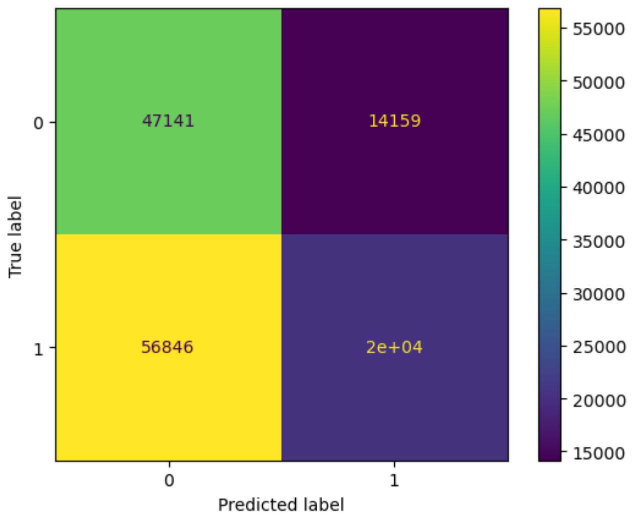

Figure 6: Confusion Matrix of the second SVM

3.6 Ensemble

Let's remember that the idea for this ensemble was to incorporate two models that would prevent overfitting of the decision tree. In this case, it's the Gaussian naive Bayes classifier and the second SVM. These two models suffer from Type I and Type II errors respectively. Therefore, when combined, they can cushion their biases when working together with the tree. This means that the tree both remains balanced and also helps to break ties between the other two models when they conflict. In other words, the tree has the tie-breaking vote, the other two models generally won't fail when they agree, and if there's a conflict among the three, they can generate a response that is often accurate.

Once we've explained the philosophy of our ensemble, let's move on to the technical part. The first thing is that we're going to use a simple majority voting with the 3 models, this voting parameter is set as hard in Scikit Learn. Now, an important factor to consider is that the tree and the Gaussian Naive Bayes were trained with unstandardized data, and the support vector machine was given the same data but standardized. Therefore, when creating the ensemble, we must create a pipeline for the SVM: first data standardization, then training.

This is how we managed to create the ensemble, producing the following results with the test data: 6

Table 6: Metrics of the Ensemble with the test set

Metric	Result
Accuracy	85.46%
Recall	94.95%
Precision	81.86%
F1	87.93%

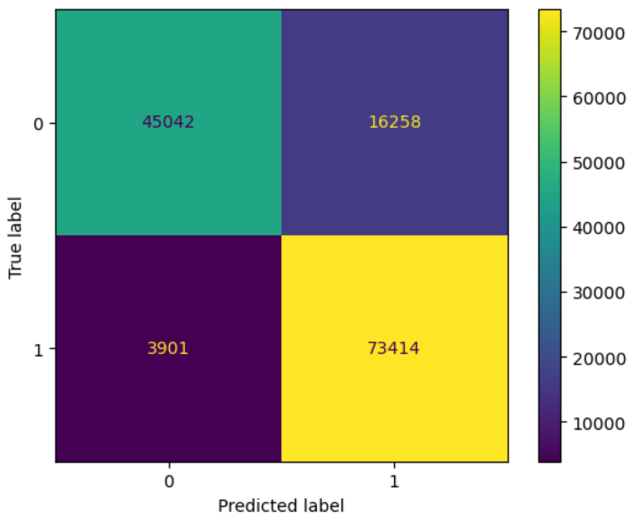

Figure 7: Confusion Matrix of the Ensemble

As we can see in 6, we have a very good model and we have met the ensemble's objectives. Firstly, looking at the metrics: we achieved a solid 85.46% accuracy. This already makes it a quite functional model. Additionally, we managed to maintain a very high recall, nearly 95%. Recall here is interpreted as not missing instances that are Uber. However, the great achievement of the ensemble was raising the precision above 80% without sacrificing recall. In the individual models, we could raise recall above 90% at the expense of precision, especially in classifying Lyft. With this ensemble, we cushion both type I and type II errors. Finally, let's recall the other objective for creating the ensemble: to avoid overfitting of the decision tree. Remember, the tree achieved metrics above 95% on the test set. This made it an exceptional model, but likely it wouldn't perform as well with new data. By covering each model's biases with the ensemble, our model has a higher chance of functioning correctly with different data. It also maintains balanced and stable metrics.

While our ensemble has met our project objectives, I would like to perform an additional verification on the model. Because it may be functioning correctly, but compared to other ensemble combinations, will it still be competitive or are there better ensembles out there?

3.7 Random Forest

To conduct this comparison test, I wanted to follow the same philosophy of building an ensemble based on the decision tree from our first model: using the Gini impurity index, maximum depth of 20, and a minimum of 10 instances to form a leaf. Therefore, to use this same tree in an ensemble, there is a robust and direct algorithm: random forest.

At a technical level, we will simply set the parameters described earlier for each tree. Additionally, to have a diverse forest, 1000 trees were generated. Let's see then if, when giving the training set to this random forest, the 1000 trees manage to surpass our decision tree model, Naive Bayes, and SVM.

Table 7: Metrics of Random Forest with the test set

Metric	Result
Accuracy	83.68%
Recall	95.68%
Precision	79.32%
F1	86.74%

As we can see in 7, we had very similar results than in 6. In terms of metrics, the Random Forest managed to increase the recall slightly, but at the cost of decreasing all other metrics. In terms of misclassified instances, in the figures 7 and 8 we can see that the original ensemble misclassified 16,258 Lyft as Uber and 3,901 Uber as Lyft. On the other hand, the random forest misclassified 19,277 Lyft as Uber and 3,339 Uber as Lyft. The nearly 600 better-classified Ubers that the Random Forest managed to achieve came at the expense of making slightly more mistakes with about 3,000 Lyft. Given the number of well-classified Ubers we already have, it is preferable to find a model that also classifies Lyft well. This was essentially the only thing missing to have an optimal model for this project. Thus, demonstrating that our ensemble of decision tree, Naive Bayes, and Support Vector Machine is the winner.

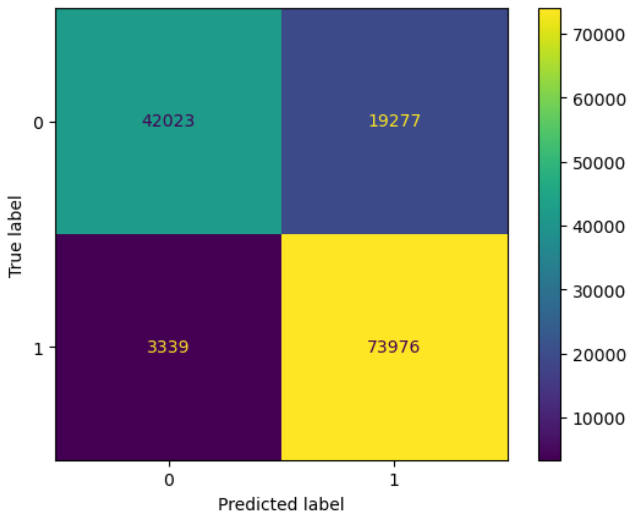

Figure 8: Confusion Matrix of Random Forest Model

4 CONCLUSIONS AND FUTURE WORK

Data science involves many stages. As seen throughout this article, it's not just about creating models with known algorithms that make predictions. The process must start with data cleaning. The proper selection of attributes and their processing is essential for models to achieve their maximum performance. On the other hand, doing machine learning is not just about knowing how models are executed. It's necessary to understand each parameter to adjust them to the needs of our project. In this work, we saw that parameters not only serve to achieve better metrics but also to avoid problems like variance error or biases towards certain classes.

Furthermore, we have the brilliant idea of ensembles: covering weaknesses and exploiting the strengths of each model by interacting with each other. This is why data scientists are needed: projects involve more than just initializing models and training them with raw data. To generate useful information, one must know how to optimally mine data.

For future work, there are two clear paths. The first is to put our models into production to see how they perform with data from other cities or seasons. Being able to identify trips from these transportation companies would be very useful for city security matters, route optimization for companies, and studying people's transportation in the city. Therefore, generating models that help us mine data from this industry has many useful applications for the future.

The second option for future work is to improve this work before putting it into production. An interesting objective would be to find the optimal configuration to get the best Random Forest. Another path for model development with this dataset would be to try more robust ensembles like XGBoost, Gradient Boosting, or LightGBM. A good objective for developing these models would be to raise all metrics above 90%, but without overfitting the model.

ACKNOWLEDGMENTS

The author would like to thank the National Autonomous University of Mexico (UNAM) and the Institute of Ecosystems and Sustainability Research (IIES) for the Diploma in Machine and Deep Learning Applied to Big Data. Especially to Dr. Sergio Rogelio Tinoco Martínez and Dr. Heberto Ferreira Medina for teaching the diploma and for providing feedback on the data analysis of this project, along with some of the models.

The author also wants to express gratitude to the Bachelor's degree in Information Technologies for Sciences for continuously preparing him in his development as a data scientist. Specifically, the author wants to acknowledge the work of Dr. Marisol Flores Garrido, Dr. Luis Miguel García Velázquez, and Mr. Rafael Pérez Estrada in the Machine Learning course. Without them, this work would not be possible.

REFERENCES

Charu C. Aggarwal. 2015. *Data Mining: The Textbook.* Springer.

Max Bramer. 2016. *Principles of Data Mining.* Springer.

Kaggle BM. [n. d.]. Kaggle: Uber and Lyft Dataset Boston, MA. https://www.kaggle.com/datasets/brllrb/uber-and-lyft-dataset-boston-ma. [Online; accessed 21-May-2024.

Miroslav Kubat. 2017. *An Introduction to Machine Learning.* Springer.

William Mendenhall, Robert J. Beaver, and Barbara M. Beaver. 2010. *Introducción a la probabilidad y estadística.* CENGAGE Learning.

scikit-learn developers. [n. d.]a. scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/. [Online; accessed 21-May-2024.

scikit-learn developers. [n. d.]b. scikit-learng: Decision Trees. https://scikit-learn.org/stable/modules/tree.html#tree. [Online; accessed 21-May-2024.

scikit-learn developers. [n. d.]c. scikit-learng: DecisionTreeClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier. [Online; accessed 21-May-2024.

scikit-learn developers. [n. d.]d. scikit-learng: GaussianNB. https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html#sklearn.naive_bayes.GaussianNB. [Online; accessed 21-May-2024.

scikit-learn developers. [n. d.]e. scikit-learng: LabelEncoder. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html#sklearn.preprocessing.LabelEncoder. [Online; accessed 21-May-2024.

scikit-learn developers. [n. d.]f. scikit-learng: LogisticRegression. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html. [Online; accessed 21-May-2024.

scikit-learn developers. [n. d.]g. scikit-learng: Naive Bayes. hhttps://scikit-learn.org/stable/modules/naive_bayes.html. [Online; accessed 21-May-2024.

scikit-learn developers. [n. d.]h. scikit-learng: OneHotEncoder. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html. [Online; accessed 21-May-2024.

scikit-learn developers. [n. d.]i. scikit-learng: StandarScaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html. [Online; accessed 21-May-2024.

scikit-learn developers. [n. d.]j. scikit-learng: train$_test_split$. . [Online; accessed 21-May-2024.

Jake VanderPlas. 2017. *Python Data Science Handbook.* O'Reilly.