**Notebook: Fraud Detection, Exercise**

Next, you'll approach the task of payment fraud detection! This is a real-world problem, with fraud accounting for billions of dollars worth of loss, worldwide. As you follow along with this lesson, you should work in the referenced SageMaker notebooks. We will present a solution to you, but please try to work on a solution of your own, when prompted. Much of the value in this experience will come from experimenting with the code, **in your own way**.

To open this notebook:

* *Navigate to your SageMaker notebook instance, in the*[*SageMaker console*](https://console.aws.amazon.com/sagemaker/)*, which has been linked to the main*[*Github exercise repository*](https://github.com/udacity/ML_SageMaker_Studies)
* *Activate the notebook instance (if it is in a "Stopped" state), and open it via Jupyter*
* *Click on the exercise notebook in the Payment\_Fraud\_Detection directory.*

You may also directly view the exercise and solution notebooks via the repository at the following links:

* [Exercise notebook](https://github.com/udacity/ML_SageMaker_Studies/blob/master/Payment_Fraud_Detection/Fraud_Detection_Exercise.ipynb)
* [Solution notebook](https://github.com/udacity/ML_SageMaker_Studies/blob/master/Payment_Fraud_Detection/Fraud_Detection_Solution.ipynb)

**The solution notebook is meant to be consulted if you are stuck or want to check your work.**

**Notebook Outline**

We'll go over the following steps to complete the notebook.

* Load in and explore payment transaction data
* Train a LinearLearner to classify the data
* Improve a basic model by accounting for class imbalance in the dataset and different metrics for model "success"

**Later: Delete Resources**

At the end of this exercise, and intermittently, you will be reminded to delete your endpoints and resources so that you do not incur any extra processing or storage fees!

# Instantiate a LinearLearner

Now that you've uploaded your training data, it's time to define and train a model! In the main exercise notebook, you'll define and train the SageMaker, built-in algorithm, LinearLearner.

### EXERCISE: Create a LinearLearner Estimator

You've had some practice instantiating built-in models in SageMaker. All estimators require some constructor arguments to be passed in.

*See if you can complete this task, instantiating a LinearLearner estimator, using only the*[*LinearLearner documentation*](https://sagemaker.readthedocs.io/en/stable/linear_learner.html)*as a resource.*

You'll find that this estimator takes in a lot of arguments, but not all are required. My suggestion is to start with a simple model, and utilize default values where applicable. Later, we will discuss some specific hyperparameters and their use cases.

### Instance Types

It is suggested that you use instances that are available in the free tier of usage: 'ml.c4.xlarge' for training and 'ml.t2.medium' for deployment.

Here is what the exercise code looks like in the main notebook:

*# import LinearLearner*

**from** sagemaker **import** LinearLearner

*# instantiate LinearLearner*

Try to complete this code on your own, and I'll go over one possible solution, next!

## Train your Estimator

After defining a model, you can format your training data and call .fit() to train the LinearLearner.

In the notebook, these exercises look as follows:

### EXERCISE: Convert data into a RecordSet format

Prepare the data for a built-in model by converting the train features and labels into numpy array's of float values. Then you can use the record\_set function to format the data as a RecordSet and prepare it for training!

*# create RecordSet of training data*

formatted\_train\_data = **None**

### EXERCISE: Train the Estimator

After instantiating your estimator, train it with a call to .fit(), passing in the formatted training data.

%%time

*# train the estimator on formatted training data*

Complete this code, and you may check out a solution, next!

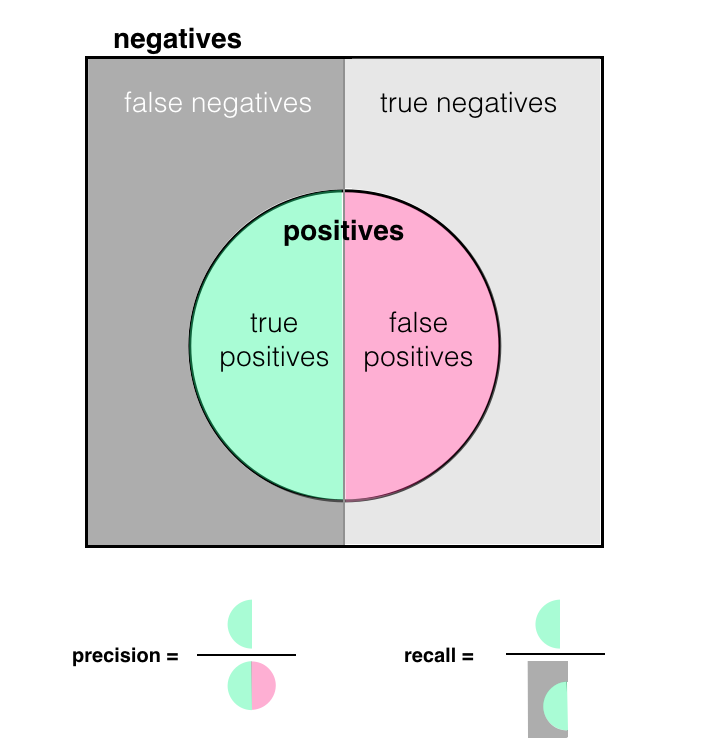
**Precision & Recall**

Precision and recall are just different metrics for measuring the "success" or performance of a trained model.

* **precision** is defined as the number of true positives (truly fraudulent transaction data, in this case) over *all* positives, and will be the higher when the amount of false positives is low.
* **recall** is defined as the number of true positives over true positives *plus* false negatives and will be higher when the number of false negatives is low.

Both take into account true positives and will be higher for high, positive accuracy, too.

I find it helpful to look at the below image to wrap my head around these measurements:



In many cases, it may be worthwhile to optimize for a higher recall or precision, which gives you a more granular look at false positives and negatives.

## Deploy an Endpoint and Evaluate Predictions

Finally, you are ready to deploy your trained LinearLearner and see how it performs according to various metrics. As you evaluate this model, I want you to think about:

* Which metrics best define success for this model?
* Is it important that we catch all cases of fraud?
* Is it important to prioritize a smooth user experience and never flag valid transactions?

The answers to these questions may vary based on use case!

In the main exercise notebook, you'll see the following instructions for deploying an endpoint and using it to make predictions:

### EXERCISE: Deploy the trained model

Deploy your model to create a predictor. We'll use this to make predictions on our test data and evaluate the model.

%%time

*# deploy and create a predictor*

linear\_predictor = **None**

### Evaluating Your Model

Once your model is deployed, you can see how it performs when applied to the test data. Let's first test our model on just one test point, to see the resulting list.

*# test one prediction*

test\_x\_np = test\_features.astype('float32')

result = linear\_predictor.predict(test\_x\_np[0])

print(result)

You should proceed with investigating and evaluating the model test results. And next, I will discuss the results I got after deploying.

### Shutting Down an Endpoint

*As always, after deploying a model and making/saving predictions, you are free to delete your model endpoint and clean up that resource.*

## Model Improvement: Accounting for Class Imbalance

We have a model that is tuned to get a higher recall, which aims to reduce the number of **false negatives**. Earlier, we discussed how class imbalance may actually bias our model towards predicting that all transactions are valid, resulting in higher **false negatives and true negatives**. It stands to reason that this model could be further improved if we account for this imbalance!

To account for class imbalance during training of a binary classifier, LinearLearner offers the hyperparameter, positive\_example\_weight\_mult, which is the weight assigned to positive (fraudulent data) examples when training a binary classifier. The weight of negative examples (valid data) is fixed at 1.

From the [hyperparameter documentation](https://docs.aws.amazon.com/sagemaker/latest/dg/ll_hyperparameters.html) on positive\_example\_weight\_mult, it reads:

*"If you want the algorithm to choose a weight so that errors in classifying negative vs. positive examples have equal impact on training loss, specify balanced."*

In the main exercise notebook, your exercises from defining to deploying an improved model looks as follows:

### EXERCISE: Create a LinearLearner with a positive\_example\_weight\_mult parameter

In addition to tuning a model for higher recall, you should add a parameter that helps account for class imbalance.

*# instantiate a LinearLearner*

*# include params for tuning for higher recall*

*# \*and\* account for class imbalance in training data*

linear\_balanced = **None**

### EXERCISE: Train the balanced estimator

Fit the new, balanced estimator on the formatted training data.

%%time

*# train the estimator on formatted training data*

### EXERCISE: Deploy and evaluate the balanced estimator

Deploy the balanced predictor and evaluate it. Do the results match with your expectations?

%%time

*# deploy and create a predictor*

balanced\_predictor = **None**

An important question here, when evaluating your model, is: **Do the results match with your expectations?** Much like in a scientific experiment it is good practice to start with a hypothesis that drives your idea for improving a model; if the trained model reacts in a different way than you expect (i.e. the model metrics are worse), it is worth revisiting your assumptions and approach.

Try to complete all these tasks, and if you get stuck, you can reference the solution video, next!

### Shutting Down the Endpoint

*Remember to delete your deployed, model endpoint after you finish with evaluation.*

# Model Design

Now that you've seen how to tune and balance a LinearLearner, it is your turn to put together all that you've learned and build a new model, based on a real, business problem. This exercise is meant to be more open-ended, so that you get practice with the steps involved in designing a model and deploying it. In this exercise you'll:

* Create a LinearLearner model, according to specifications
* Train and deploy the model
* Evaluate the results
* Delete the endpoint (after evaluation)

Here is what you'll see in the main exercise notebook:

### EXERCISE: Train and deploy a LinearLearner with appropriate hyperparameters, according to the given scenario

**Scenario:**

*A bank has asked you to build a model that optimizes for a good user experience; users should only ever have up to about 15% of their valid transactions flagged as fraudulent.*

This requires that you make a design decision: Given the above scenario, **what metric (and value)** should you aim for during training?

You may assume that performance on a training set will be within about 5-10% of the performance on a test set. For example, if you get 80% on a training set, you can assume that you'll get between about 70-90% accuracy on a test set.

**Your final model should account for class imbalance and be appropriately tuned.**

%%time

*# instantiate and train a LinearLearner*

*# include params for tuning for higher precision*

*# \*and\* account for class imbalance in training data*

%%time

*# deploy and evaluate a predictor*

*## IMPORTANT*

*# delete the predictor endpoint after evaluation*

In this case, I will not be walking through a detailed solution (and there are multiple ways to approach this task and come up with a solution), but you can see one example solution in the solution notebook and on the next page.

### Final Cleanup!

After completing these tasks, double check that you have deleted **all** your endpoints, and associated files. I'd also suggest manually deleting your S3 bucket, models, and endpoint configurations directly from your AWS console. You can find thorough cleanup instructions, [in the documentation](https://docs.aws.amazon.com/sagemaker/latest/dg/ex1-cleanup.html).

# One Possible Solution

To optimize for few false positives (misclassified, valid transactions), I defined a model that accounts for class imbalance and optimizes for a **high precision**.

Let's review the scenario:

*A bank has asked you to build a model that optimizes for a good user experience; users should only ever have up to about 15% of their valid transactions flagged as fraudulent.*

My thoughts: If we're allowed about 15/100 incorrectly classified valid transactions (false positives), then I can calculate an approximate value for the precision that I want as: 85/(85+15) = 85%. I'll aim for about 5% higher during training to ensure that I get closer to 80-85% precision on the test data.

%%time

*# One possible solution*

*# instantiate and train a LinearLearner*

*# include params for tuning for higher precision*

*# \*and\* account for class imbalance in training data*

linear\_precision = LinearLearner(role=role,

train\_instance\_count=1,

train\_instance\_type='ml.c4.xlarge',

predictor\_type='binary\_classifier',

output\_path=output\_path,

sagemaker\_session=sagemaker\_session,

epochs=15,

binary\_classifier\_model\_selection\_criteria='recall\_at\_target\_precision',

target\_precision=0.9,

positive\_example\_weight\_mult='balanced')

*# train the estimator on formatted training data*

linear\_precision.fit(formatted\_train\_data)

This model trains for a fixed precision of 90%, and, under that constraint, tries to get as high a recall value as possible. After training, I deployed the model to create a predictor:

%%time

*# deploy and evaluate a predictor*

precision\_predictor = linear\_precision.deploy(initial\_instance\_count=1, instance\_type='ml.t2.medium')

*INFO:sagemaker:Creating model with name: linear-learner-2019-03-11-04-07-10-993 INFO:sagemaker:Creating endpoint with name linear-learner-2019-03-11-03-36-56-524*

Then evaluated the model by seeing how it performed on test data:

print('Metrics for tuned (precision), LinearLearner.\n')

*# get metrics for balanced predictor*

metrics = evaluate(precision\_predictor,

test\_features.astype('float32'),

test\_labels,

verbose=**True**)

These were the results I got:

Metrics for tuned (precision), LinearLearner.

prediction (col) 0.0 1.0

actual (row)

0.0 85276 26

1.0 31 110

Recall: 0.780

Precision: 0.809

Accuracy: 0.999

As you can see, we still misclassified 26 of the valid results and so I may have to go back and up my aimed-for precision; the recall and accuracy are not too bad, considering the precision tradeoff.

Finally, I made sure to **delete the endpoint** after I was doe with evaluation.

*## IMPORTANT*

*# delete the predictor endpoint*

delete\_endpoint(precision\_predictor)

*Deleted linear-learner-2019-03-11-03-36-56-524*