Week 12 — Save and package your model for deployment

1. Saving your model for deployment as a pickle file

This week's work focused on preparing the final model from Week 11 for deployment. The assignment required us to serialize the model, organize the data needed for inference, and document the environment dependencies to ensure the model can be reproduced and executed in other systems. To enable deployment, we serialized the final Week 11 model (Lasso retrained on the fully cleaned Week 10 dataset), using Python's built-in pickle module. This allowed the model to be saved as a portable binary file and reloaded later without retraining. And the serialized model is stored at: Reports/w11_out/lasso_retrained_model.pkl

The saved file includes all learned parameters, coefficients, and hyperparameters. After serialization, we verified that the model could be successfully loaded and used to generate predictions on the Week 11 test dataset. This confirms that the model is fully functional outside the original training environment and ready for deployment in any compatible Python system.

To ensure smooth inference during deployment, the input data must match the feature structure used during training. Therefore, we exported all datasets in both .csv and .pkl formats. And all files are stored in the directory: Reports/w12_deploy/. Each dataset includes all feature columns plus a target column containing the true TSR values. To guarantee compatibility between the saved data and the serialized model, we also saved a feature schema file named: feature_schema.json. This ensures that any new data provided during deployment can be automatically aligned with the expected feature order. Missing features will be filled with zeros, and extra features will be removed.

2. Environment Dependencies

To ensure reproducibility and compatibility, we documented the full environment used for training and saving the model.

Operating System:

- JupyterHub
- Google Colab

Python Version:

- Python 3.12 on Google Colab
- Python 3.10+ on JupyterHub

Any Python version between 3.10 and 3.12 is appropriate for deployment.

Python Package Dependencies:

Core packages required for loading the model and running inference include:

- numpy (≈1.26)
- pandas (≈2.1)
- scikit-learn (1.3 or higher)
- scipy
- joblib
- matplotlib
- seaborn

Standard library modules used:

- pickle
- json
- pathlib

These packages ensure that both model serialization and data alignment can run smoothly.

Recommended Deployment Requirements:

numpy==1.26.4

pandas==2.1.4

scikit-learn==1.3.2

scipy==1.11.4

matplotlib==3.8.2

seaborn==0.13.1

joblib==1.3.2

3.Deployment mode

Our code and business setting both point toward batch deployment for the retrained Lasso model from week eleven. The target is total shareholder return for several listed real estate companies, and the predictors we engineered, such as lagged returns and rolling averages, change at daily or slower frequencies. Stakeholders do not need second by second forecasts. The practical use case is to refresh predictions once per trading day and feed a clean table of results into reports or portfolio tools.

The week twelve notebook already follows this design. We serialize the final Lasso model as a pickle file stored under Reports slash w11 underscore out slash lasso underscore retrained underscore model dot p k l. In the same notebook we copy the cleaned train, validation and test splits used in week eleven to a deployment folder named Reports slash w12 underscore deploy and we write a feature schema file that records the list of feature columns and the target name. We also define a helper function called load underscore for underscore inference that reads a table of new data, aligns columns to the schema, fills missing features with zeros and discards unexpected columns. This is ideal for a once per day batch job that loads a single table of fresh features for all firms, runs the model and stores a batch of predictions.

Batch deployment keeps infrastructure simple and matches the way our data arrives. Each day produces one consistent batch of predictions that can later be compared to realized returns. When monitoring signals drift or degradation, we can pause the next batch and retrain the model using the extra data.

4.Performance metrics for monitoring

In the week eleven retraining cell we evaluated the Lasso model using several statistical metrics. These include mean absolute error MAE, root mean squared error RMSE, mean absolute percentage error MAPE and R squared. For production monitoring we select MAE, RMSE and R squared as the core statistical metrics, while MAPE remains available for diagnostics.

MAE measures the typical size of prediction errors in units of total shareholder return. RMSE places more weight on large misses and highlights rare but extreme errors. R squared shows how much variation in future returns is explained by the model relative to a constant benchmark.

The assignment also asks for business and other metrics. From a business perspective we will track the share of predictions whose absolute error exceeds a tolerance that is meaningful for decision makers. If portfolio managers care most about avoiding errors larger than five percentage points, then the fraction of such large errors becomes a direct risk indicator. We will also follow directional accuracy, meaning the share of cases where the model correctly predicts whether total shareholder return will be above or below a benchmark. For the other category we continue the fairness work from week eleven by monitoring mean absolute error by firm or by metro region and the gap between the best and worst groups. This guards against situations where overall metrics look fine but one company or one city consistently receives poorer predictions.

5.Green, yellow and red thresholds

We define thresholds relative to the validation performance from week eleven. For mean absolute error, green status means that live MAE stays within ten percent of the validation value or lower. Yellow status starts once MAE rises to between ten and thirty percent higher than that baseline. Red status is triggered when MAE grows by more than thirty percent above baseline and the model should then be taken out of production until we retrain or redesign it.

For root mean squared error we use similar but slightly tighter bounds. Green means RMSE within ten percent of the validation level. Yellow corresponds to an increase between ten and twenty five percent. Red is any increase above twenty five percent or any sudden jump in a single monitoring period.

For R squared, green status means that the live value is no more than five percentage points lower than the validation R squared. Yellow means a drop between five and fifteen points. Red means a drop larger than fifteen points or a level very close to zero.

For the business metric of large errors, green status means that fewer than ten percent of predictions have an absolute error larger than five percentage points. Yellow means between ten and twenty percent. Red means more than twenty percent.

For directional accuracy, green means that the model correctly predicts the sign of the return at least sixty percent of the time. Yellow covers directional accuracy between fifty five and sixty percent. Red is any sustained period where directional accuracy falls below fifty five percent.

For the fairness metric across firms or cities, we focus on the ratio between the highest and lowest group level MAE. Green means that this gap remains smaller than twenty percent of the overall MAE. Yellow means a gap between twenty and forty percent. Red means a wider gap than forty percent, which would trigger a closer investigation into data quality for the affected groups and possibly group specific modeling.

6. Risk-mitigation strategies for green, yellow, and red status

Once the model has been deployed, the monitoring thresholds only tell us *when* performance changes, but they do not tell us *how* to respond. Therefore, it is necessary to define a clear mitigation strategy for each alert level so that the model remains reliable in real-world use.

In the green stage, all monitored metrics remain within the expected variation range established during Week 11. Although no intervention is required, this stage still needs routine monitoring. The main responsibility in the green stage is to make sure the data pipeline continues to run smoothly: the incoming feature tables must retain the expected column names and distributions, the feature-schema alignment must be applied correctly, and the predictions must be logged consistently. Even small technical issues, such as missing columns or delayed data uploads, can cause later problems, so the green stage is mostly about maintaining stable operations and consistently comparing daily predictions to realized values.

When metrics reach the yellow stage, the model is showing early warning signals. Errors rise above tolerance, R-squared drops noticeably, or fairness gaps begin to widen. At this point, we do not immediately remove the model from production, but we must start deeper diagnostics. Yellow-stage actions include checking for data quality issues such as abnormal missing values, unusual shifts in macroeconomic variables, or a sudden change in the TSR distribution. We also re-examine subgroup errors, especially for companies that historically showed higher volatility. During this stage, we prepare a shadow-retrained model using the latest data, but we do not deploy it yet. This allows rapid replacement if conditions worsen.

The red stage means the model is no longer trustworthy. Sudden spikes in RMSE, a collapse in directional accuracy, or extremely large fairness gaps indicate serious model degradation. In this stage, the model should be temporarily removed from production. A full retraining process must begin immediately, following the same steps used in Week 10 and Week 11: data cleaning, smoothing, feature alignment, bias mitigation, and full evaluation. Red-stage risk mitigation also includes investigating whether the problem came from data drift, concept drift, or data-pipeline failures. Only after thorough validation and stability checks should the new version be promoted back to production.

This multi-level plan ensures that the deployed model remains safe, predictable, and responsive to real-world changes.

7. Retraining frequency and rationale

Retraining is essential for keeping the model aligned with financial market behavior. Because TSR (Total Shareholder Return) is sensitive to changing market conditions, macroeconomic cycles, and company-specific shifts, a model trained only once cannot remain accurate indefinitely.

A reasonable retraining frequency for this use case is every 1–3 months. This schedule aligns with the pace of financial reporting and allows enough new observations to accumulate so the model can detect structural updates. Monthly or quarterly retraining also prevents the Lasso coefficients from becoming stale, especially for companies whose business conditions change more rapidly.

In addition to scheduled retraining, the monitoring thresholds described above support event-triggered retraining. If the model enters the yellow or red zone, retraining should begin immediately regardless of the calendar. This hybrid approach—scheduled plus performance-triggered—ensures the model remains both stable and responsive.

Before any retrained model replaces the production version, it must pass the full evaluation pipeline: accuracy metrics, group-level fairness checks, and residual analysis. This guarantees that updates improve performance instead of introducing new risks.

8. Data drift, concept drift, and mitigation plan

Once deployed, the model must deal with changes in the real world that were not present in the training data. These changes fall into two categories: data drift and concept drift, and both can negatively impact predictions if not monitored properly.

Data drift happens when the distribution of the input features changes over time. For example, if treasury yields or unemployment rates shift to ranges that were not present in the Week 10 dataset, then the model receives inputs it was not trained on. Even if the relationship between the features and TSR stays the same, unfamiliar input distributions cause the model to behave unpredictably.
To detect data drift, we will compare incoming feature distributions with historical distributions using

statistical checks such as mean and variance differences or KS tests. When drift is detected, mitigation steps include adjusting normalization rules, recalibrating distributions, or retraining the model with the updated data.

Concept drift is more serious. It happens when the relationship between features and TSR changes, even if the feature distributions remain stable. This is common in financial markets: a factor that used to predict TSR may stop working when market regimes shift. Examples include major policy changes, macroeconomic shocks, or shifts in investor expectations.
Concept drift is harder to detect because the model's inputs look normal, but its predictions suddenly become wrong. Monitoring sudden drops in directional accuracy, rising RMSE, or changes in company-level fairness is the most reliable way to identify concept drift. The primary mitigation method is retraining with the latest data so that the model learns the updated relationships.

In practice, both data drift and concept drift will be managed together through routine drift detection, scheduled retraining, and automatic escalation when the model enters the yellow or red zones. This ensures the model remains aligned with real-world dynamics and avoids performance collapse.