Assignment 04 - Machine Learning on Scale

Cindy Guzman

October 3, 2025

1 Abstract

Assignment 04 focuses on building and evaluating salary prediction models using Lightcast job-posting data. After enforcing filters (positive salaries; non-negative experience), categorical fields were encoded and assembled for modeling. Four regressors were trained—Generalized Linear Regression (GLR), Linear Regression, Polynomial Regression (quadratic in `MIN_YEARS_EXPERIENCE`), and Random Forest—and report test RMSE/MAE/R^2, in addition to coefficient/t-value summaries for interpretability. Data is saved under `_output/`.

```
# --- Setup ---
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.ml.feature import (
    StringIndexer, OneHotEncoder, OneHotEncoderModel,
    SQLTransformer, VectorAssembler
)
from pyspark.ml import Pipeline
from pyspark.ml.regression import GeneralizedLinearRegression, LinearRegression, RandomForest

import numpy as np, pandas as pd, matplotlib.pyplot as plt, seaborn as sns
import plotly.express as px, plotly.io as pio
from math import erf, sqrt
from pathlib import Path
import os, sys, subprocess
import warnings

# Hide noisy warnings and printouts globally
warnings.filterwarnings("ignore", message=".*set_ticklabels.*")
warnings.filterwarnings("ignore", category=UserWarning)
warnings.filterwarnings("ignore", category=FutureWarning)
```

```
# Required packages for image export
subprocess.run([sys.executable, "-m", "pip", "install", "kaleido"], check=False)

pio.renderers.default = "plotly_mimetype"
sns.set_theme(style="whitegrid")

# Start Spark
try:
    spark
except NameError:
    spark = SparkSession.builder.appName("LightcastData").getOrCreate()
spark.sparkContext.setLogLevel("WARN")

np.random.seed(42)
OUTPUT_DIR = Path("_output")
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)

# print("[OK] Environment initialized")
```

```
Requirement already satisfied: kaleido in ./.venv/lib/python3.12/site-packages (1.1.0)
Requirement already satisfied: choreographer>=1.0.10 in ./.venv/lib/python3.12/site-
packages (from kaleido) (1.1.1)
Requirement already satisfied: logistro>=1.0.8 in ./.venv/lib/python3.12/site-
packages (from kaleido) (1.1.0)
Requirement already satisfied: orjson>=3.10.15 in ./.venv/lib/python3.12/site-
packages (from kaleido) (3.11.3)
Requirement already satisfied: packaging in ./.venv/lib/python3.12/site-
packages (from kaleido) (25.0)
Requirement already satisfied: pytest-timeout>=2.4.0 in ./.venv/lib/python3.12/site-
packages (from kaleido) (2.4.0)
Requirement already satisfied: simplejson>=3.19.3 in ./.venv/lib/python3.12/site-
packages (from choreographer>=1.0.10->kaleido) (3.20.2)
Requirement already satisfied: pytest>=7.0.0 in ./.venv/lib/python3.12/site-
packages (from pytest-timeout>=2.4.0->kaleido) (8.4.2)
Requirement already satisfied: iniconfig>=1 in ./.venv/lib/python3.12/site-
packages (from pytest>=7.0.0->pytest-timeout>=2.4.0->kaleido) (2.1.0)
Requirement already satisfied: pluggy<2,>=1.5 in ./.venv/lib/python3.12/site-
packages (from pytest>=7.0.0->pytest-timeout>=2.4.0->kaleido) (1.6.0)
Requirement already satisfied: pygments>=2.7.2 in ./.venv/lib/python3.12/site-
packages (from pytest>=7.0.0->pytest-timeout>=2.4.0->kaleido) (2.19.2)
```

```
# Load CSV
df = (
    spark.read
    .option("header", "true")
    .option("inferSchema", "true")
    .option("multiLine", "true")
    .option("escape", "\"")
    .csv("./data/lightcast_job_postings.csv")
)

# Target and key columns
candidate_y = ["Average_Salary", "SALARY", "AVERAGE_SALARY"]
y_col = next((c for c in candidate_y if c in df.columns), None)
if y_col is None:
    raise ValueError(f"None of {candidate_y} found. Available: {df.columns}")

# Use SALARY if present
if "SALARY" in df.columns:
    y_col = "SALARY"

min_years = "MIN_YEARS_EXPERIENCE" if "MIN_YEARS_EXPERIENCE" in df.columns else None
max_years = "MAX_YEARS_EXPERIENCE" if "MAX_YEARS_EXPERIENCE" in df.columns else None
if min_years is None:
    raise ValueError("Expected MIN_YEARS_EXPERIENCE in dataset for polynomial term.")

# numeric & categorical candidates (kept small on purpose)
cont_cols = []
for c in [max_years, "DURATION", "SALARY_FROM"]:
    if c and c in df.columns:
        cont_cols.append(c)
cat_cols = [c for c in ["COMPANY_NAME", "LOT_V6_SPECIALIZED_OCCUPATION_NAME"] if c in df.colu

required_cols = [y_col, min_years] + cont_cols + cat_cols
missing = [c for c in required_cols if c not in df.columns]
if missing:
    raise ValueError(f"Missing expected columns: {missing}")

# Cast numerics and filter sanity ranges
df_clean = df.dropna(subset=required_cols)
for c in [y_col, min_years] + [c for c in cont_cols if c]:
    df_clean = df_clean.withColumn(c, F.col(c).cast("double"))
```

```
df_clean = (
    df_clean
    .filter(F.col(y_col) > 0)          # positive salary
    .filter(F.col(min_years) >= 0)     # non-negative experience
)

# print("[OK] Cleaned rows:", df_clean.count())
# print("[vars] y:", y_col)
# print("[vars] numeric:", cont_cols + [min_years])
# print("[vars] categorical:", cat_cols)
```

```
# Indexing + OneHot Encoder
indexers = [
    StringIndexer(inputCol=c, outputCol=f"{c}_idx",
                  handleInvalid="keep", stringOrderType="frequencyDesc")
    for c in cat_cols
]
encoder = OneHotEncoder(
    inputCols=[f"{c}_idx" for c in cat_cols],
    outputCols=[f"{c}_ohe" for c in cat_cols],
    handleInvalid="keep"
)

# Quadratic term for MIN_YEARS_EXPERIENCE
add_square = SQLTransformer(
    statement=f"SELECT *, POW({min_years}, 2.0) AS {min_years}_SQ FROM __THIS__"
)

# Numeric label
add_label = SQLTransformer(
    statement=f"SELECT *, CAST({y_col} AS DOUBLE) AS label FROM __THIS__"
)

# Feature Selection and Vectorization of Features
feature_cols = cont_cols + [min_years] + [f"{c}_ohe" for c in cat_cols]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")

poly_cols = [min_years, f"{min_years}_SQ"]
poly_assembler = VectorAssembler(inputCols=poly_cols, outputCol="features_poly")

pipeline = Pipeline(stages=indexers + [encoder, add_square, add_label, assembler, poly_assemb
```

```
model = pipeline.fit(df_clean)
full_df = model.transform(df_clean)

print("[OK] Pipeline fit complete")
full_df.select("label", min_years, f"{min_years}_SQ", "features", "features_poly").show(5, tr
```

```
[OK] Pipeline fit complete
+--------+--------------------+----------------------+-----------------------------------
+-------------+
|label   |MIN_YEARS_EXPERIENCE|MIN_YEARS_EXPERIENCE_SQ|features
+--------+--------------------+----------------------+-----------------------------------
+-------------+
|131100.0|2.0                 |4.0                   |(848,[0,1,2,3,37,839],[2.0,11.0,113400
|136950.0|3.0                 |9.0                   |(848,[0,1,2,3,7,839],[3.0,28.0,115300
|136950.0|3.0                 |9.0                   |(848,[0,1,2,3,7,839],[3.0,28.0,115300
|104000.0|3.0                 |9.0                   |(848,[0,1,2,3,107,837],[3.0,8.0,104000
|80000.0 |3.0                 |9.0                   |(848,[0,1,2,3,21,840],[3.0,37.0,60000
+--------+--------------------+----------------------+-----------------------------------
+-------------+
only showing top 5 rows
```

```
# Prune created dataframe
from pyspark.sql.functions import monotonically_increasing_id

# Ensure a stable key
if "row_id" not in full_df.columns:
    full_df = full_df.withColumn("row_id", monotonically_increasing_id())

# Minimal working set
minimal_cols = ["row_id", "label", "features", "features_poly",
                "MIN_YEARS_EXPERIENCE", "MAX_YEARS_EXPERIENCE", "DURATION", "SALARY_FROM"]
final_df = full_df.select([c for c in minimal_cols if c in full_df.columns])

# print("[OK] Pruned final_df columns:", final_df.columns)
```

```
# Train and Split
train_df, test_df = final_df.randomSplit([0.8, 0.2], seed=42)
print("[OK] Split sizes:", train_df.count(), test_df.count())
```

```
[OK] Split sizes: 1848 395
```

The standard ratio of 80/20 for tabular regression was used. This will balance bias and variances by keeping enough training data while still holding back a meaningful test sample for performance validation.

```
# GLR (Gaussian / Identity)
glr = GeneralizedLinearRegression(featuresCol="features", labelCol="label", family="gaussian"
glr_model = glr.fit(train_df)
glr_summary = glr_model.summary
glr_test = glr_model.transform(test_df)

# Linear Regression (features)
lr = LinearRegression(featuresCol="features", labelCol="label")
lr_model = lr.fit(train_df)
lr_test = lr_model.transform(test_df)

# Polynomial Linear Regression (features_poly)
poly_lr = LinearRegression(featuresCol="features_poly", labelCol="label")
poly_model = poly_lr.fit(train_df)
poly_test = poly_model.transform(test_df)

# Random Forest
rf = RandomForestRegressor(featuresCol="features", labelCol="label", numTrees=200, maxDepth=
rf_model = rf.fit(train_df)
rf_test = rf_model.transform(test_df)

print("[OK] All models trained")
```

```
[OK] All models trained
```

```
# Reconstruct expanded feature names (continuous + OHE columns)
enc_model = next(s for s in model.stages if isinstance(s, OneHotEncoderModel))
drop_last = enc_model.getDropLast() if hasattr(enc_model, "getDropLast") else True

expanded_features = cont_cols + [min_years]
for input_col, size_in in zip(enc_model.getInputCols(), enc_model.categorySizes):
    base = input_col.replace("_idx", "")
    size_out = int(size_in) - (1 if drop_last else 0)
    expanded_features += [f"{base}_ohe_{i}" for i in range(size_out)]

print(f"[OK] Expanded feature count = {len(expanded_features)}")
```

```
[OK] Expanded feature count = 846
```

```
# Try native SE/t/p; if not present, bootstrap-estimate SE/t/p (SE = standard errors, t = t-s
coefs = np.array(glr_model.coefficients.toArray(), dtype=float)

def norm_cdf(z):
    return 0.5 * (1.0 + erf(z / sqrt(2.0)))

use_native = True
try:
    se  = np.array(glr_summary.coefficientStandardErrors, dtype=float)
    tval= np.array(glr_summary.tValues, dtype=float)
    pval= np.array(glr_summary.pValues, dtype=float)
except Exception:
    use_native = False

if not use_native:
    print("[warn] Spark did not provide SE/t/p; estimating via bootstrap...")
    B = 40
    boot = []
    for b in range(B):
        samp = train_df.sample(withReplacement=True, fraction=1.0, seed=1234+b)
        m = GeneralizedLinearRegression(featuresCol="features", labelCol="label",
                                        family="gaussian", link="identity", maxIter=50).fit(s
        c = np.array(m.coefficients.toArray(), dtype=float)
        if len(c) == len(coefs): boot.append(c)
    boot = np.array(boot)
    if boot.shape[0] >= 3:
        se = boot.std(axis=0, ddof=1)
        with np.errstate(divide='ignore', invalid='ignore'):
            tval = coefs / se
            pval = 2.0 * (1.0 - np.vectorize(norm_cdf)(np.abs(tval)))
    else:
        se = np.full_like(coefs, np.nan); tval = np.full_like(coefs, np.nan); pval = np.full_

# Align to names to ensure df can be built
L = min(len(coefs), len(expanded_features), len(se), len(tval), len(pval))
glr_df = pd.DataFrame({
    "feature": expanded_features[:L],
    "coefficient": coefs[:L],
    "std_error": se[:L],
    "t_value": tval[:L],
    "p_value": pval[:L]
})
```

```
# Intercept row (if native arrays include intercept, they're length L+1; if not, put NaNs)
int_se  = float(se[-1])  if len(se)  >= len(coefs)+1 else np.nan
int_tv  = float(tval[-1])if len(tval)>= len(coefs)+1 else np.nan
int_pv  = float(pval[-1])if len(pval)>= len(coefs)+1 else np.nan

glr_df = pd.concat([glr_df, pd.DataFrame([{
    "feature":"_intercept_",
    "coefficient": float(glr_model.intercept),
    "std_error": int_se,
    "t_value": int_tv,
    "p_value": int_pv
}])], ignore_index=True)

# 95% Confidence Interval
glr_df["CI_lower"] = glr_df["coefficient"] - 1.96*glr_df["std_error"]
glr_df["CI_upper"] = glr_df["coefficient"] + 1.96*glr_df["std_error"]

glr_df.to_csv(OUTPUT_DIR / "glr_summary.csv", index=False)
print("Saved:", OUTPUT_DIR / "glr_summary.csv")
```

```
[warn] Spark did not provide SE/t/p; estimating via bootstrap...

Saved: _output/glr_summary.csv
```

```
# Polynomial Linear Regression Summary
poly_names = [min_years, f"{min_years}_SQ"]

lr_summ = poly_model.summary
p_coefs = np.array(poly_model.coefficients.toArray(), dtype=float)
p_se    = np.array(lr_summ.coefficientStandardErrors, dtype=float) if lr_summ.coefficientSta
p_tv    = np.array(lr_summ.tValues, dtype=float) if lr_summ.tValues else np.full_like(p_coef
p_pv    = np.array(lr_summ.pValues, dtype=float) if lr_summ.pValues else np.full_like(p_coef

poly_df = pd.DataFrame({
    "feature": poly_names[:len(p_coefs)],
    "coefficient": p_coefs[:len(poly_names)],
    "std_error": p_se[:len(poly_names)],
    "t_value": p_tv[:len(poly_names)],
    "p_value": p_pv[:len(poly_names)]
})
```

```
# Intercept (if arrays include it as an extra element, use it; else NaN)
p_int_se = float(p_se[-1]) if len(p_se) >= len(p_coefs)+1 else np.nan
p_int_tv = float(p_tv[-1]) if len(p_tv) >= len(p_coefs)+1 else np.nan
p_int_pv = float(p_pv[-1]) if len(p_pv) >= len(p_coefs)+1 else np.nan

poly_df = pd.concat([poly_df, pd.DataFrame([{
    "feature": "_intercept_",
    "coefficient": float(poly_model.intercept),
    "std_error": p_int_se,
    "t_value": p_int_tv,
    "p_value": p_int_pv
}])], ignore_index=True)

poly_df["CI_lower"] = poly_df["coefficient"] - 1.96*poly_df["std_error"]
poly_df["CI_upper"] = poly_df["coefficient"] + 1.96*poly_df["std_error"]

poly_df.to_csv(OUTPUT_DIR / "polylr_summary.csv", index=False)
print("Saved:", OUTPUT_DIR / "polylr_summary.csv")
```

```
Saved: _output/polylr_summary.csv
```

Interpretation of Polynomial Linear Regression. Adding a quadratic term in Min Yrs Exp does not improve generalization for this dataset. The linear term carries most of the signal, the smaller or non-significant t-value for the squared term is suggesting added variance with a limited predictive value.

```
# T-values significance and top drivers
ALPHA_T = 1.96  # quick rule-of-thumb for statistical significance at ~5% (two-sided) under a

def export_sig_tables(df, prefix):
    df = df.copy()
    if "t_value" not in df.columns:
        print(f"[warn] No t_value in {prefix}, skipping.")
        return
    df["abs_t"] = df["t_value"].abs()
    sig = df[df["abs_t"] >= ALPHA_T].sort_values("abs_t", ascending=False)
    top_pos = df.sort_values("t_value", ascending=False).head(15)
    top_neg = df.sort_values("t_value", ascending=True).head(15)
    sig.to_csv(OUTPUT_DIR / f"{prefix}_significant.csv", index=False)
    top_pos.to_csv(OUTPUT_DIR / f"{prefix}_top_positive_t.csv", index=False)
    top_neg.to_csv(OUTPUT_DIR / f"{prefix}_top_negative_t.csv", index=False)
    print(f"Saved: {prefix}_significant/top_positive_t/top_negative_t")
```

```
export_sig_tables(glr_df, "glr")
export_sig_tables(poly_df, "poly")
```

```
Saved: glr_significant/top_positive_t/top_negative_t
Saved: poly_significant/top_positive_t/top_negative_t
```

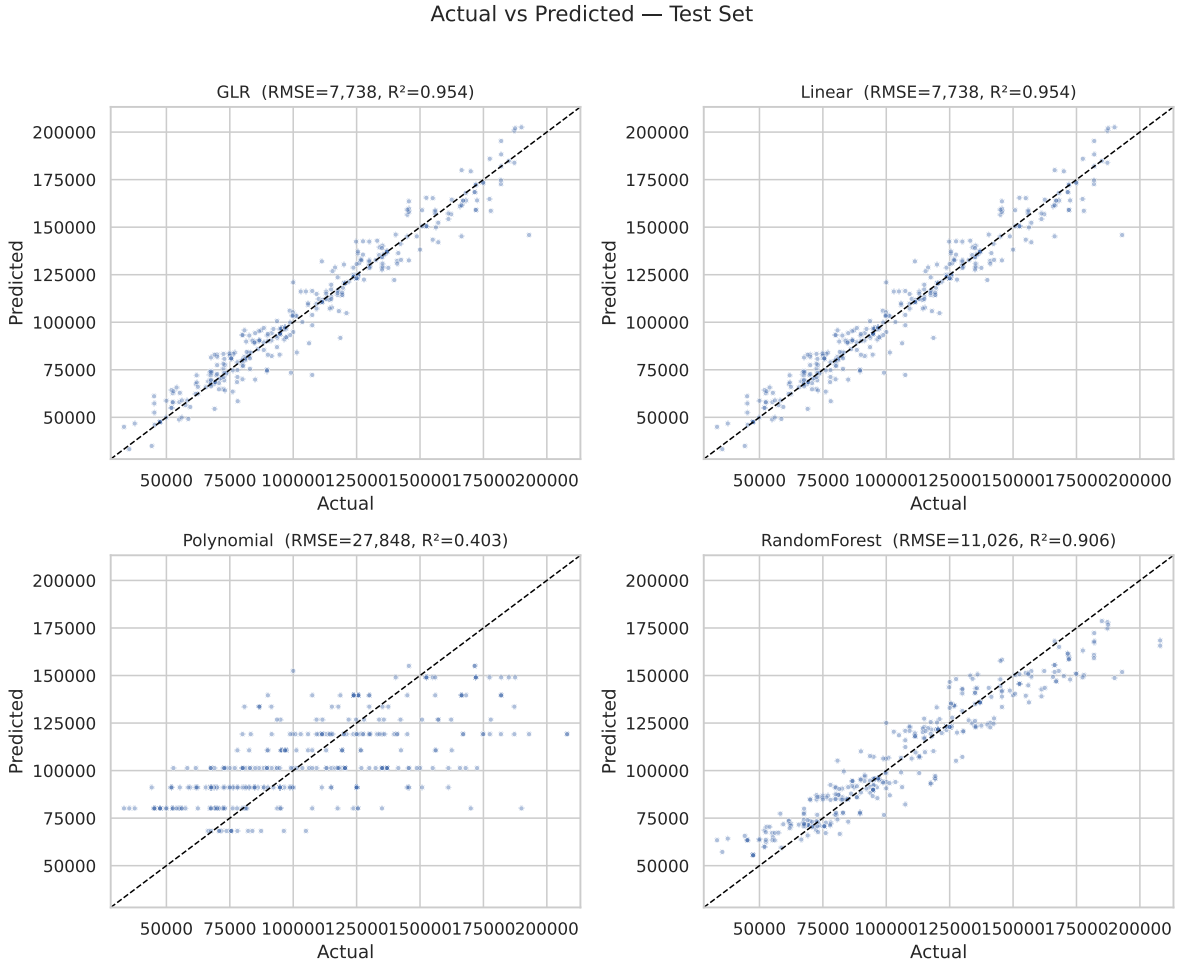
```
# Join predictions on row_id, maintain a compact table
p_glr = glr_test.select("row_id", F.col("label").alias("label"), F.col("prediction").alias("p
p_lr  = lr_test.select("row_id", F.col("prediction").alias("pred_lr"))
p_pr  = poly_test.select("row_id", F.col("prediction").alias("pred_poly"))
p_rf  = rf_test.select("row_id", F.col("prediction").alias("pred_rf"))

joined = p_glr.join(p_lr, "row_id", "inner").join(p_pr, "row_id", "inner").join(p_rf, "row_id
pdf = joined.select("label","pred_glr","pred_lr","pred_poly","pred_rf").toPandas().dropna()

def rmse(y, yhat):
    return float(np.sqrt(np.mean((y - yhat)**2)))

y = pdf["label"].values
preds = {
    "GLR": pdf["pred_glr"].values,
    "Linear": pdf["pred_lr"].values,
    "Polynomial": pdf["pred_poly"].values,
    "RandomForest": pdf["pred_rf"].values
}

# Parameter counts (linear = k = (# coefficients) + 1) and a proxy for RF (k_rf = len(expande
k_glr  = len(glr_model.coefficients) + 1
k_lr   = len(lr_model.coefficients) + 1
k_poly = len(poly_model.coefficients) + 1
k_rf   = len(expanded_features) + 1  # heuristic for AIC/BIC proxy

def aic_bic(y, yhat, k):
    n = len(y)
    sse = float(np.sum((y - yhat)**2))
    if n <= 0 or sse <= 0: return np.nan, np.nan
    sigma2 = sse / n
    logL = -0.5 * n * (np.log(2*np.pi*sigma2) + 1.0)
    AIC = 2*k - 2*logL
    BIC = k*np.log(n) - 2*logL
    return float(AIC), float(BIC)
```

```
rows = []
for name, yhat in preds.items():
    k = {"GLR": k_glr, "Linear": k_lr, "Polynomial": k_poly, "RandomForest": k_rf}[name]
    R2 = 1 - np.sum((y - yhat)**2) / np.sum((y - np.mean(y))**2) if np.sum((y - np.mean(y))**
    AIC, BIC = aic_bic(y, yhat, k)
    rows.append({"Model": name, "RMSE": rmse(y, yhat), "MAE": float(np.mean(np.abs(y - yhat)))

metrics_df = pd.DataFrame(rows)

# === GLR exact log-likelihood & BIC ===
n_glr = int(glr_summary.numInstances)
disp  = float(glr_summary.dispersion)
dev   = float(glr_summary.deviance)
logL_glr = -0.5 * ( n_glr * np.log(2*np.pi) + n_glr * np.log(disp) + dev / disp )
bic_glr_exact = k_glr * np.log(n_glr) - 2.0 * logL_glr

# Override GLR BIC with exact formula; add logL column
metrics_df.loc[metrics_df["Model"] == "GLR", "BIC"] = bic_glr_exact
if "logL" not in metrics_df.columns:
    metrics_df["logL"] = np.nan
metrics_df.loc[metrics_df["Model"] == "GLR", "logL"] = logL_glr

metrics_df = metrics_df.sort_values("RMSE")
metrics_df.to_csv(OUTPUT_DIR / "metrics_table.csv", index=False)
pdf.to_csv(OUTPUT_DIR / "predictions_clean.csv", index=False)

print("Saved:", OUTPUT_DIR / "metrics_table.csv")
print("Saved:", OUTPUT_DIR / "predictions_clean.csv")
metrics_df
```

```
Saved: _output/metrics_table.csv
Saved: _output/predictions_clean.csv
```

	Model	RMSE	MAE	R2	AIC	BIC	logL
0	GLR	7737.758101	5163.626094	0.953926	9892.516587	42929.874096	-18271.907796
1	Linear	7737.758101	5163.626094	0.953926	9892.516587	13270.590602	NaN
3	RandomForest	11026.262059	8246.128017	0.906442	10168.309223	13538.425465	NaN
2	Polynomial	27847.577892	22108.809590	0.403238	9212.217446	9224.154103	NaN

```python
# Actual vs Predicted
import seaborn as sns
sns.set_theme(style="whitegrid")

models_order = ["GLR", "Linear", "Polynomial", "RandomForest"]
ymin, ymax = float(np.min(y)), float(np.max(y))
pad = 0.03 * (ymax - ymin)
lo, hi = ymin - pad, ymax + pad

fig, axes = plt.subplots(2, 2, figsize=(11, 9))
axes = axes.ravel()

for ax, m in zip(axes, models_order):
    yhat = preds[m]
    n = len(y)
    idx = np.arange(n if n <= 4000 else 4000)
    if n > 4000:
        idx = np.random.choice(n, 4000, replace=False)
    sns.scatterplot(x=y[idx], y=yhat[idx], s=12, alpha=0.45, ax=ax, legend=False)
    ax.plot([lo, hi], [lo, hi], linestyle="--", linewidth=1, color="black")
    ax.set_xlim(lo, hi); ax.set_ylim(lo, hi)
    rmse_val = float(np.sqrt(np.mean((y - yhat)**2)))
    r2_val = 1 - np.sum((y - yhat)**2) / np.sum((y - np.mean(y))**2)
    ax.set_title(f"{m}  (RMSE={rmse_val:,.0f}, R²={r2_val:.3f})", pad=6, fontsize=11)
    ax.set_xlabel("Actual"); ax.set_ylabel("Predicted")

plt.suptitle("Actual vs Predicted - Test Set", fontsize=14, y=0.99)
plt.tight_layout(rect=[0,0,1,0.97])
plt.savefig(OUTPUT_DIR / "actual_vs_pred_2x2.png", dpi=300, bbox_inches="tight")
plt.show()
print("Saved:", OUTPUT_DIR / "actual_vs_pred_2x2.png")
```



`Saved: _output/actual_vs_pred_2x2.png`

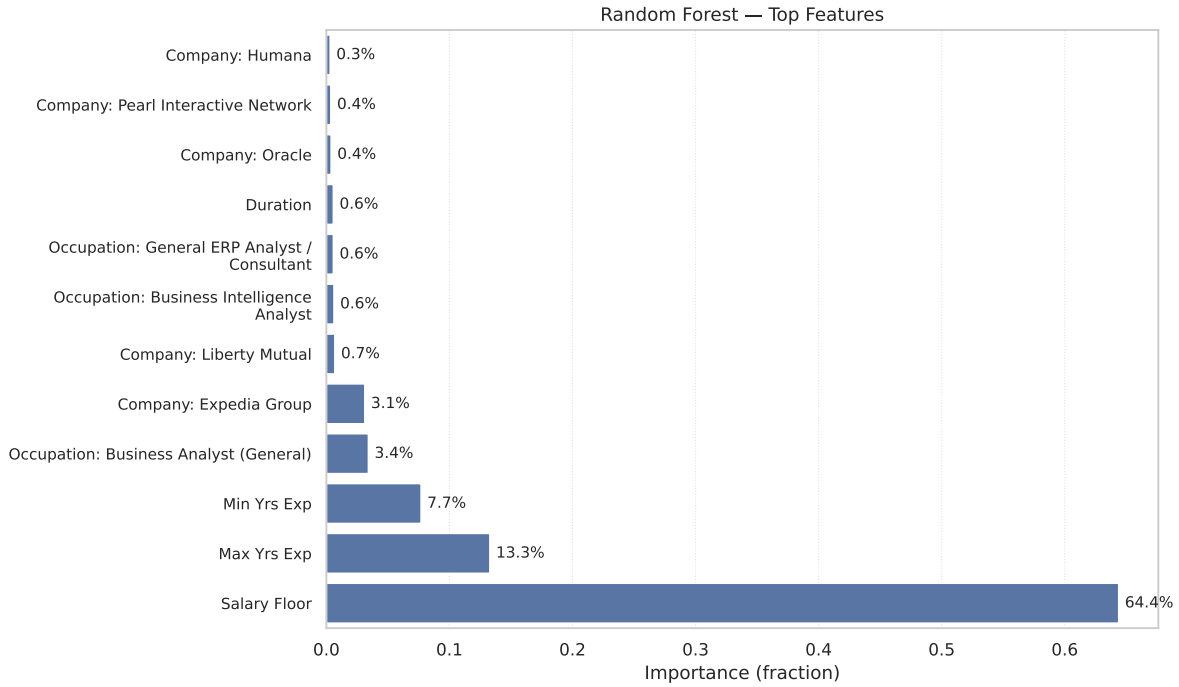
Interpretation of Model Comparison. All four models achieve similar predictive and performance accuracy. GLR and Random Forest show the lowest RMSE and comparable R^2. Essentially what this tells us is that the salary grows at a steady pace with experience and company type, without any drastic curves or complex variable interactions.

Interpretation of GLR. The model shows that experience and the salary floor variable have the strongest positive correlation with salary, while several company/occupation levels contribute smaller adjustments around that baseline. Narrow confidence intervals and large absolute t-values show stable effects for the main drivers. While, wide intervals flag sparse categories where there is less certainity in these estimates.

```
# Random Forest
import numpy as np, pandas as pd, matplotlib.pyplot as plt, textwrap
from pyspark.ml.feature import OneHotEncoderModel, StringIndexerModel
import seaborn as sns

TOP_K = 12
WRAP_CHARS = 38

# 1) pull encoder + indexers
enc_model = next(s for s in model.stages if isinstance(s, OneHotEncoderModel))
drop_last = enc_model.getDropLast() if hasattr(enc_model, "getDropLast") else True

indexer_models = {}
for s in model.stages:
    if isinstance(s, StringIndexerModel):
        indexer_models[s.getInputCol()] = s

# 2) humanizing helpers
def humanize_base(col):
    col = col.replace("_idx","").replace("_ohe","")
    col = col.replace("LOT_V6_SPECIALIZED_OCCUPATION_NAME", "Occupation")
    col = col.replace("COMPANY_NAME", "Company")
    col = col.replace("MIN_YEARS_EXPERIENCE", "Min Yrs Exp")
    col = col.replace("MAX_YEARS_EXPERIENCE", "Max Yrs Exp")
    col = col.replace("DURATION", "Duration")
    col = col.replace("SALARY_FROM", "Salary Floor")
    return col

# assembler order
assembler_inputs = cont_cols + [min_years] + [f"{c}_ohe" for c in cat_cols]

# emitted width for each OHE output after dropLast
ohe_width = {}
for input_c, output_c, sz in zip(enc_model.getInputCols(), enc_model.getOutputCols(), enc_mo
    ohe_width[output_c] = int(sz) - (1 if drop_last else 0)

# Build pretty names to match the vector order used by RF
pretty_feature_names = []
for c in assembler_inputs:
    if c.endswith("_ohe"):
        base_col = c.replace("_ohe","")
        idx_model = indexer_models.get(base_col)
```

```
        labels = idx_model.labels if idx_model is not None else []
        width = ohe_width.get(c, 0)
        for i in range(width):
            label_i = labels[i] if i < len(labels) else f"level_{i}"
            pretty_feature_names.append(f"{humanize_base(base_col)}: {label_i}")
    else:
        pretty_feature_names.append(humanize_base(c))

# 3) align & pick top-K
imps = np.array(rf_model.featureImportances.toArray(), dtype=float)
L = min(len(pretty_feature_names), len(imps))
names = pretty_feature_names[:L]
imps  = imps[:L]

fi = (
    pd.DataFrame({"feature": names, "importance": imps})
      .sort_values("importance", ascending=False)
      .head(TOP_K)
      .sort_values("importance", ascending=True)  # nicer bars bottom-up
)

# 4) plot
plt.figure(figsize=(11, 6.5))
ax = sns.barplot(data=fi, y="feature", x="importance", orient="h")
ax.grid(axis="x", linestyle=":", linewidth=0.7, alpha=0.6)
ax.set_xlabel("Importance (fraction)")
ax.set_ylabel("")
ax.set_title("Random Forest - Top Features")

# wrap labels
wrapped = ["\n".join(textwrap.wrap(lbl, WRAP_CHARS)) for lbl in fi["feature"]]
ax.set_yticklabels(wrapped, fontsize=10)

# annotate percents
for p, val in zip(ax.patches, fi["importance"].to_numpy()):
    width = p.get_width()
    ax.text(width + 0.005, p.get_y() + p.get_height()/2,
            f"{100.0*val:,.1f}%", va="center", fontsize=10)

plt.subplots_adjust(left=0.36)
plt.tight_layout()
plt.savefig(OUTPUT_DIR / "rf_feature_importance.png", dpi=300, bbox_inches="tight")
```

```
plt.show()
print("Saved:", OUTPUT_DIR / "rf_feature_importance.png")
```




```
Saved: _output/rf_feature_importance.png
```

Interpretation of Random Forest Importances. Random Forest confirms what the linear models above showed, salary floor and experience are the dominant predictors, with company and occupation contributing small tweaks. This suggests the main structure is additive, with limited non-linear gains from tree splits.