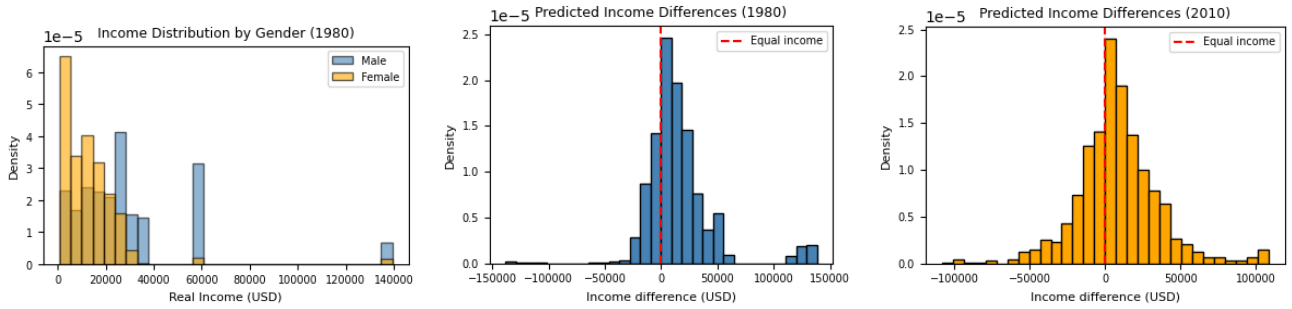
We selected **QRF (Quantile Regression Forest)** as our final model due to its strong performance and advantages. We chose a non-parametric model as any potential wrong assumption is worse than imposing no assumptions, but the assumption that similar circumstances as in the train set must hold in the test set [1]. This is particularly beneficial when the data is skewed, as we found real income to be, refer to Appendix 1 [2].

Besides QRF, we employed various models for estimating the distribution. For these models, data manipulation was used; we kept the processed data for generality.

We performed hyperparameter optimization through random search; the grid we used is described in Appendix 2. Since QRF builds upon random forests, we tune the hyperparameters commonly optimised in such models [3]. We are aware that the optimal set found is by chance. With 20 replications, we tried to ensure that the grid is explored sufficiently. For a selected parameter set, we run a 5-fold cross-validation procedure where we compute the CRPS on each fold and average the results at the end to obtain a reliable value for out-of-sample performance. We kept the hyperparameter set that yields the lowest average CRPS across the 5-folds.

We found the best hyperparameters to be: $n_{\text{estimators}} = 800$, max_depth $= 18$, min_samples_leaf $= 4$, and max_features $\approx$ "*sqrt*". These parameters achieved an average out-of-sample CRPS value of 8,134$. The PIT histogram is approximately uniform with minor deviations, indicating good calibration. A slight central hump suggests mild overdispersion, meaning the model slightly overestimates uncertainty [4].

Additionally, we trained another QRF model on only the year and gender. With this model, we can find the relation of income between gender over time. In 1980, females exhibited slightly higher real income inequality than males, as indicated by a higher Gini coefficient (0.432 vs. 0.418). This suggests that the distribution of female incomes was more dispersed relative to their mean income. Figure 1a shows the income distribution.



(a) 1980 Predicted income distribution by gender.

(b) Predicted income differences in 1980.

(c) Predicted income differences in 2010.

Figure 1: (a) Income distribution by gender in 1980, (b) predicted income differences in 1980, and (c) predicted income differences in 2010.

Figures 1b and 1c show histograms of 1,000 Monte Carlo–simulated income differences (male minus female) for 1980 and 2010, respectively, used to estimate the probability that males earned more than females. We find that in 1980, the empirical probability of 0.692. In 2010, we find an empirical probability of 0.610. The decline in the estimated probability suggests that the gender pay gap has narrowed. Nevertheless, as the model conditions only on gender, this finding cannot be attributed exclusively to gender effects, as other variables are not accounted for. Distribution histograms are found in the appendix.

References

[1] Kevin Dowd and David Blake. After var: The theory, estimation and insurance applications of quantile-based risk measures. Technical Report PI-0603, The Pensions Institute, Cass Business School, City University, London, UK, 2006.

[2] Francis Sahngun Nahm. Nonparametric statistical tests for continuous data: The basic concept and the practical use. *Korean Journal of Anesthesiology*, 69(1):8–14, 2016.

[3] Philipp Probst, Marvin N. Wright, and Anne-Laure Boulesteix. Hyperparameters and tuning strategies for random forest. *WIREs Data Mining and Knowledge Discovery*, 9(3), Jan 2019.

[4] Malte Tichy. Demystifying the probability integral transform, 2024. Accessed: 2025-10-11.

[5] Alexander Jordan, Fabian Krüger, and Sebastian Lerch. Evaluating probabilistic forecasts with `scoringRules`. *Journal of Statistical Software*, 90(12):1–37, 2019.

1 Appendix A

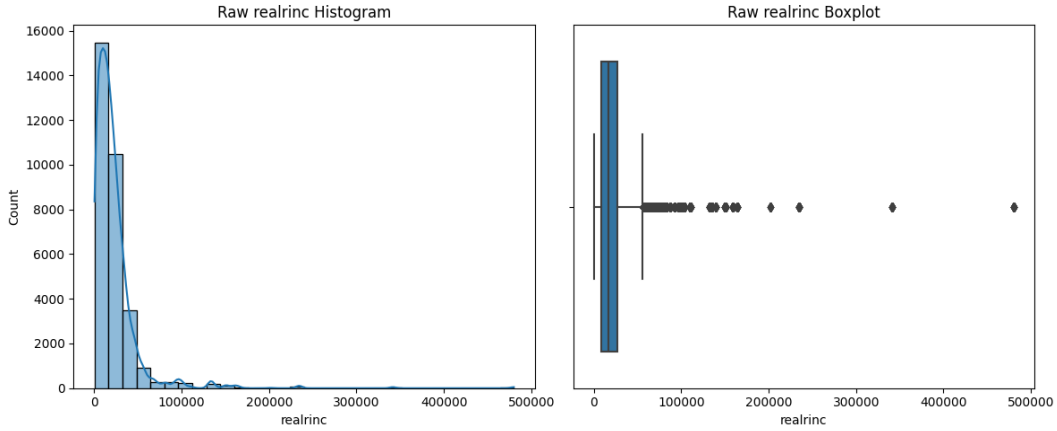


Figure 2: Plot of the income variable. We observe a skew on the right side in the data, this would mean that there is more mass on above the modal then below the modal.

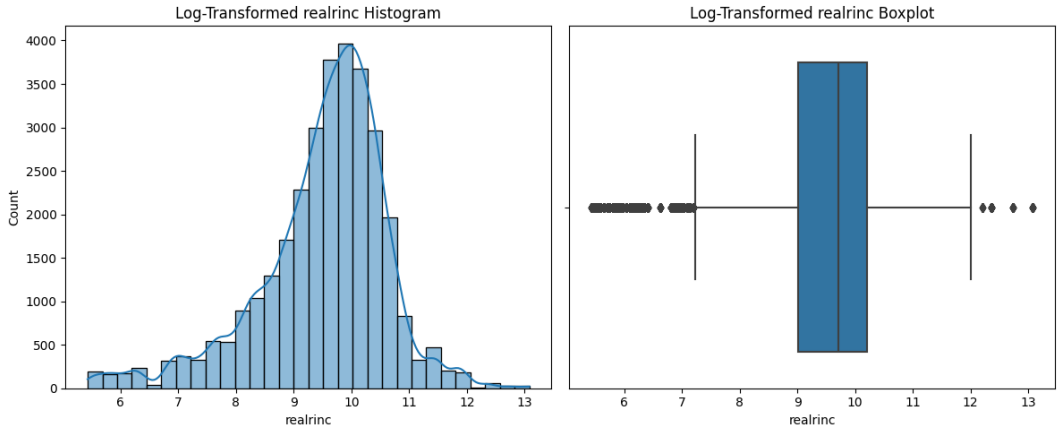


Figure 3: Plot of the log-income variable. We observe a skew on the left side in the data, this is due to the log transform.

2 Appendix B

n_estimators	{50, 200, 300, 800}
max_depth	[4, 20]
min_samples_leaf	[2, 20]
max_features	{"sqrt", log_2}

Table 1: The search grid we did hyperparameter search over. The parameter search was a random search with 20 trials. We used the seed 1738 for reproducibility.

n_estimators	800
max_depth	18
min_samples_leaf	4
max_features	”sqrt”

Table 2: The optimal set found through random search hyperparameter optimisation.

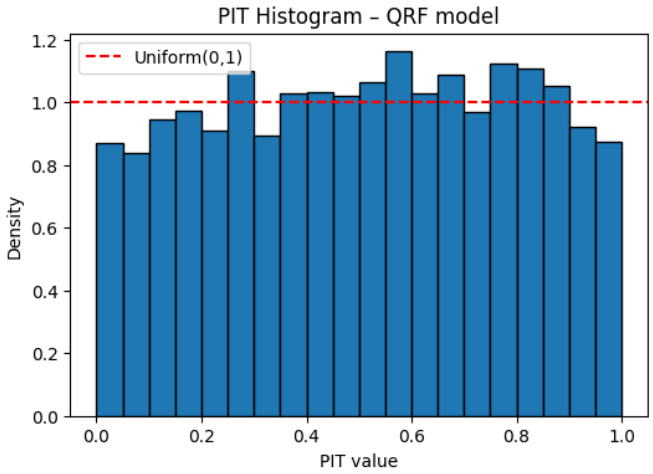


Figure 4: PIT Plot of our trained QRF on the 80/20 split. The distribution of PIT values shows a slight deviation from uniformity.

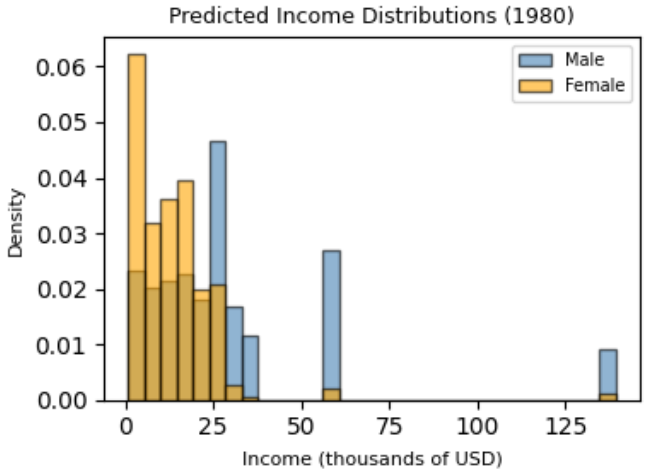


Figure 5: The histogram of the simulated distribution for the year 1980, for males and females

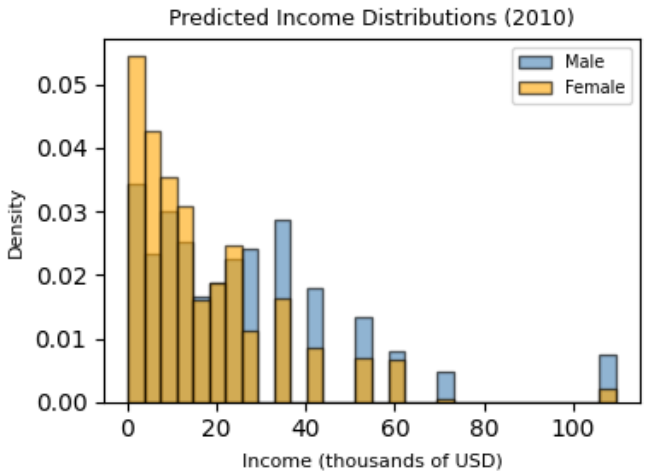


Figure 6: The histogram of the simulated distribution for the year 2010, for males and females

3 Appendix C

In this part, we explain the Python class and some functions we have used.

`_weighted_quantile(x, q, w)` computes a weighted quantile of the data x at quantile level q using sample weights w.

```
def _weighted_quantile(x, q, w):
    x = np.asarray(x); w = np.asarray(w)
    s = np.argsort(x); x, w = x[s], w[s]
    cw = np.cumsum(w) / np.sum(w)
    return np.interp(q, cw, x)
```

The code snippet below is where we create the class QRF, as we did not use a library directly. The class contains functions fit, predict, quantiles, and sample. In the code, the class acts as the variable model. When model.function of QRF.function is called, it uses this class and its built-in function.

```
class QRF:
    """
    Quantile Regression Forest using leaf-weight aggregation over training targets.
    - predict_quantiles(): weighted quantiles of training y
    - sample(): draws from empirical conditional distribution using leaf weights
    """
    def __init__(self, **rf_params):
        self.model = RandomForestRegressor(**rf_params)
        self.leaf_maps_ = None
        self.y_train_ = None
        self.n_train_ = None

    def fit(self, X, y):
        X = pd.DataFrame(X).reset_index(drop=True)
        y = pd.Series(y).reset_index(drop=True).values
        self.model.fit(X, y)
```

```
        self.y_train_ = y
        self.n_train_ = len(y)

        # For each tree: map leaf_id -> indices of training samples in that leaf
        self.leaf_maps_ = []
        for tree in self.model.estimators_:
            leaves = tree.apply(X)
            m = {}
            for lid in np.unique(leaves):
                m[lid] = np.where(leaves == lid)[0]
            self.leaf_maps_.append(m)
        return self

    def _weights_for(self, Xq):
        Xq = pd.DataFrame(Xq)
        n_q = len(Xq)
        W = np.zeros((n_q, self.n_train_), dtype=float)
        n_trees = len(self.model.estimators_)
        for t, tree in enumerate(self.model.estimators_):
            lids = tree.apply(Xq)
            mp = self.leaf_maps_[t]
            for i, lid in enumerate(lids):
                idx = mp[lid]
                W[i, idx] += 1.0 / (n_trees * len(idx))
        # each row sums to 1
        return W

    def predict_quantiles(self, X, quantiles):
        qs = np.asarray(quantiles).ravel()
        W = self._weights_for(X)
        out = np.empty((len(X), len(qs)), dtype=float)
        for i in range(len(X)):
            out[i] = [_weighted_quantile(self.y_train_, q, W[i]) for q in qs]
        return out

    def sample(self, X, n_samples=1000, rng=None):
        if rng is None:
            rng = np.random.default_rng(1738)
        W = self._weights_for(X)
        draws = np.empty((len(X), n_samples), dtype=float)
        for i in range(len(X)):
            draws[i] = rng.choice(self.y_train_, size=n_samples, p=W[i])
        return draws
```

This function calculates the CRPS from samples. This means that we do not use a continuous CRPS. The calculations are inspired by [5]. Where we use the formula

$$CRPS = E_F|X_1 - y| - \frac{1}{2}E_{F,F}|X_1 - X_2| \tag{1}$$

This is approximately:

$$\approx \frac{1}{S}\sum_{s=1}^{S}|y_s - y| - \frac{1}{2S^2}\sum_{s=1}^{S}\sum_{s'=1}^{S}|y_s - y_{s'}| \tag{2}$$

```python
def crps_from_samples(y_true, samples):
    y_true = np.asarray(y_true).reshape(-1, 1)
    samples = np.asarray(samples)
    n, S = samples.shape
    A = np.mean(np.abs(samples - y_true), axis=1)
    xs = np.sort(samples, axis=1)
    k = np.arange(1, S + 1, dtype=float)
    weights = 2 * k - S - 1
    B = (2.0 / (S**2)) * np.sum(xs * weights, axis=1)
    return A - 0.5 * B
```

This function uses the argument model, QRF in this instance, the factors, and a series of alphas. It returns the quantiles of the sample that is predicted through the trees.

```python
def predict_quantiles_qrf(model, X, alphas):
    return model.predict_quantiles(X, alphas)
```

This function uses the uniform distribution to map a value between 0 and 1 to predict a value in the predictive distribution.So we find values, these are qvalues, then we interpolate the values through the random samples of the uniform. Where uniform values must be seen as a mapping of the empirical interpolated CDF.

```python
def inverse_cdf_sample_from_quantiles(qvals, alphas, n_samples, rng):
    qvals = np.asarray(qvals)
    alphas = np.asarray(alphas).ravel()
    n, m = qvals.shape
    U = rng.uniform(alphas[0], alphas[-1], size=(n, n_samples))
    out = np.empty((n, n_samples), dtype=float)
    for i in range(n):
        out[i] = np.interp(U[i], alphas, qvals[i])
    return out
```

This function returns the predictive samples of the QRF model. It uses the function samples from the model, the tree model of QRF. This is sampled an amount of times.

```python
def get_predictive_samples_qrf(model, X, n_samples=200, rng=None):
    if rng is None:
        rng = np.random.default_rng(1738)
    try:
        # Use QRF's own sampling method
        draws = np.asarray(model.sample(X, n_samples=n_samples))
        if draws.shape == (len(X), n_samples):
            return draws
    except Exception:
        raise RuntimeError("QRF sampling failed. Check the model's .sample() method.")

```

This function does a K-fold estimation and estimates the average CRPS for each fold.

```
def cv_crps_for_params(X, y, qrf_cls, params, K=5, S=200, seed=1738):
    if not isinstance(X, (pd.DataFrame, pd.Series)):
        X = pd.DataFrame(X)
    y = pd.Series(y).reset_index(drop=True)
    kf = KFold(n_splits=K, shuffle=True, random_state=seed)
    rng = np.random.default_rng(seed)
    scores = []
    for tr, va in kf.split(X):
        Xtr, Xva = X.iloc[tr], X.iloc[va]
        ytr, yva = y.iloc[tr], y.iloc[va]
        model = qrf_cls(**params)
        model.fit(Xtr, ytr)
        samples = get_predictive_samples_qrf(model, Xva, n_samples=S, rng=rng)
        scores.append(np.mean(crps_from_samples(yva.values, samples)))
    return float(np.mean(scores))
```

This function uses a predetermined search field for the hyper parameters. And returns a dictionary with the random found values.

```
def sample_qrf_params(rng):
    n_estimators = int(rng.choice([300, 500, 800, 1200]))
    max_depth = int(rng.integers(6, 31))
    min_samples_leaf = int(rng.integers(1, 21))
    mf_choice = rng.choice(['sqrt', 'log2', 'float'])
    max_features = float(rng.uniform(0.3, 0.9)) if mf_choice == 'float' else mf_choice
    return dict(
        n_estimators=n_estimators,
        max_depth=max_depth,
        min_samples_leaf=min_samples_leaf,
        max_features=max_features,
        bootstrap=True,
        random_state=1738,
        n_jobs=-1
    )
```

This function retrieves the the randomly generated hyperparameters and then estimates the CRPS score using the aforementioned functions. It only stores the best average CRPS that is found through the K-fold estimation, it then stores these values.

```
def random_search_qrf(X, y, qrf_cls, n_trials=60, K=5, S=200, seed=1738,
↪   verbose=False):
    rng = np.random.default_rng(seed)
    recs, best = [], {"score": np.inf, "params": None}
    for t in range(1, n_trials + 1):
        params = sample_qrf_params(rng)
        score = cv_crps_for_params(X, y, qrf_cls, params, K=K, S=S, seed=seed)
        recs.append({**params, "cv_crps": score})
        if score < best["score"]:
            best = {"score": score, "params": params}
```

```
        if verbose:
            print(f"[{t}/{n_trials}] CV-CRPS={score:.6f} | best={best['score']:.6f}")
    df = pd.DataFrame.from_records(recs).sort_values("cv_crps")
    return df, best
```

This function is used for the prediction challenge. We use the X_test set and the found best parameters, then try to estimate y_test. This is then stored in the appropriate file.

```
def fit_best_and_write_predictions(X, y, X_test, qrf_cls, best_params, n_samples=1000,
↪   seed=1738):
    if not isinstance(X, (pd.DataFrame, pd.Series)):
        X = pd.DataFrame(X)
    if not isinstance(X_test, (pd.DataFrame, pd.Series)):
        X_test = pd.DataFrame(X_test)
    model = qrf_cls(**best_params)
    model.fit(X, y)
    rng = np.random.default_rng(seed)
    draws = get_predictive_samples_qrf(model, X_test, n_samples=n_samples, rng=rng)
    return model, draws
```